The Supercomputing People

Session 8

The java.io Package

Contents

java.io.File

File class

- It describes the properties of the file.
  - E.g., file size, check read-only or updatable, named pipe, get more information such as access permissions, date, time

- Does not specify how information is retrieved from or stored in file.

- A directory in Java is treated as a file with an additional property where a list of filenames can be examined.

java.io.File extends java.lang.Object

Constructors in File class

- **File**(File parent, String child)
  - Creates a new File instance from a parent abstract pathname and a child pathname string.
- **File**(String pathname)
  - Creates a new File instance by converting the given pathname string into an abstract pathname.
- **File**(String parent, String child)
  - Creates a new File instance from a parent pathname string and a child pathname string.
- **File**(URI uri)
  - Creates a new File instance by converting the given file: URI into an abstract pathname.

File Utilities

- **File Names**
  - String getName( )
  - String getPath( )
  - String getAbsolutePath( )
  - String getParent( )
  - boolean renameTo(File newName)

File Tests

- **File Tests**
  - boolean exits( )
  - boolean canWrite( )
  - boolean canread( )
  - boolean isFile( )
  - boolean isDirectory( )
  - boolean isAbsolute( )

General File Information and Directory Utilities

- **General File Information**
  - long lastModifies( )
  - long length( )
  - boolean delete( )

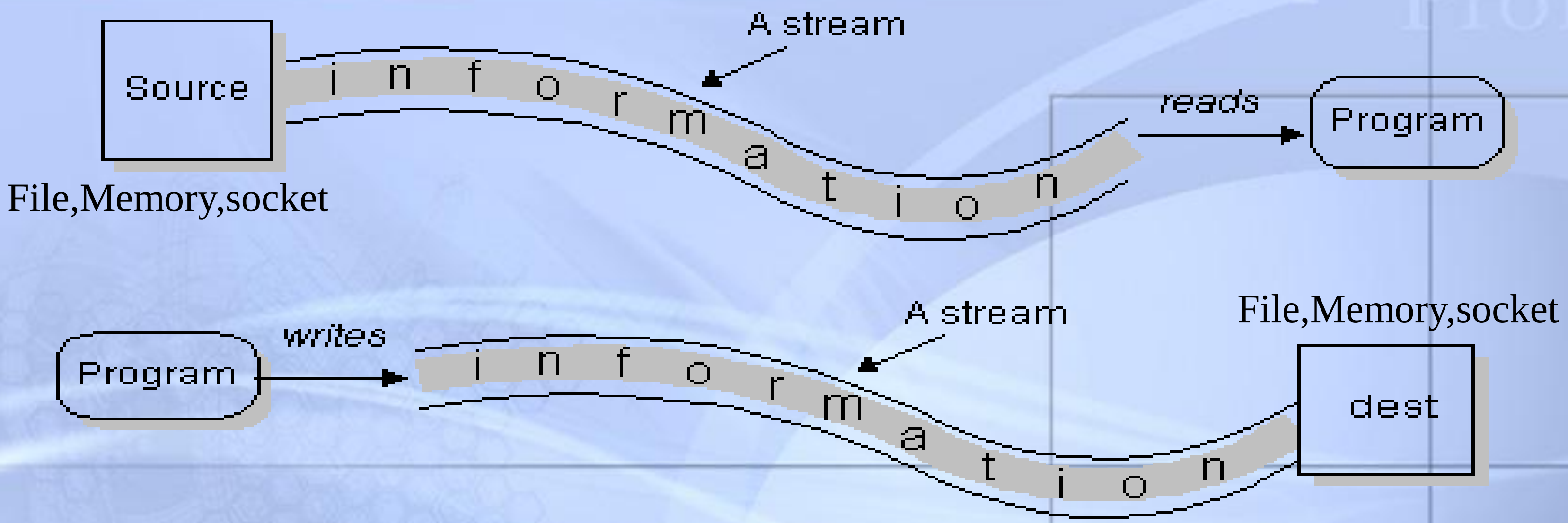
- **Directory Utilities**
  - boolean mkdir( )
  - String[ ] list( )

Example

```
import java.io.File;
class file {
    public static void main(String args[]) {
        File file1 = new File("file.txt");
        System.out.println("File: " + file1.getName() + (file1.isFile() ? " is a file" : " is
                              a named pipe"));
        System.out.println("Size: " + file1.length());
        System.out.println("Path: " + file1.getPath());
        System.out.println("Absolute Path: " + file1.getAbsolutePath());
        System.out.println("File was last modified: " + file1.lastModified());
        System.out.println(file1.exists() ? "File exists" : "File does not exist");
        System.out.println(file1.canRead() ? "File can be read from" :  "File cannot be read
      from");
      System.out.println(file1.isDirectory() ? "File is a directory" : "File is not a directory");
    } }
```

Stream Overview

Streams Overview



- A ***stream*** is a path of communication between a source of information and its destination.
- The java.io package defines I/O in terms of streams.
- The java.net package provides specific support for network I/O, based around the use of sockets, with an underlying stream or channel-based model.

Streams Overview

- Two Major parts in the package java.io :
  - **byte(8 bits) streams**
  - **character(16-bit UTF-16 characters) streams**
- I/O is either text-based or data-based (binary)
- Input streams or output streams → **byte stream**
- Readers or Writers → **character streams**

Streams Overview

- Five group of classes and interfaces in java.io
  - The general classes for building different types of byte and character streams.
  - A range of classes that define various types of streams – filtered, piped, and some specific instances of streams
  - The data stream classes and interfaces for reading and writing primitive values and strings.
  - For Interacting with files
  - For the object serialization mechanism

How to do I/O

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

1) *Open* the stream
2) *Use* the stream (read, write, or both)
3) *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.

15

Using the Data Sink Streams

- **Sink Types**
  - Memory
  - Pipe
  - File

Memory Sink

- **Character Streams**
  - CharArrayReader, CharArrayWriter
  - StringReader, StringWriter

- **Byte Streams**
  - ByteArrayInputStream, ByteArrayOutputStream
  - StringBufferInputStream

Pipe Sink

- **Character Streams**
  - PipedReader,   PipedWriter

- **Byte Streams**
  - PipedInputStream, PipedOutputStream

File Sink

- **Character Streams**
  - FileReader, FileWriter

- **Byte Streams**
  - FileInputStream, FileOutputStream

Using the Processing Stream

- Buffering
- Converting between Bytes and Character
- Object Serialization
- Counting
- Printing
- Filtering
- Concatenation
- Data Conversion
- Peeking Ahead

Process: Buffering

- **CharacterStreams**
  - BufferedReader, BufferedWriter
- **Byte Streams**
  - BufferedInputStream, BufferedOutputStream

Process: Filtering

- **CharacterStreams**
  - FilterReader, FilterWriter

- **Byte Streams**
  - FilterInputStream,   FilterOutputStream

Process: Data Conversion

- **CharacterStreams**
  - None
- **Byte Streams**
  - DataInputStream,   DataOutputStream

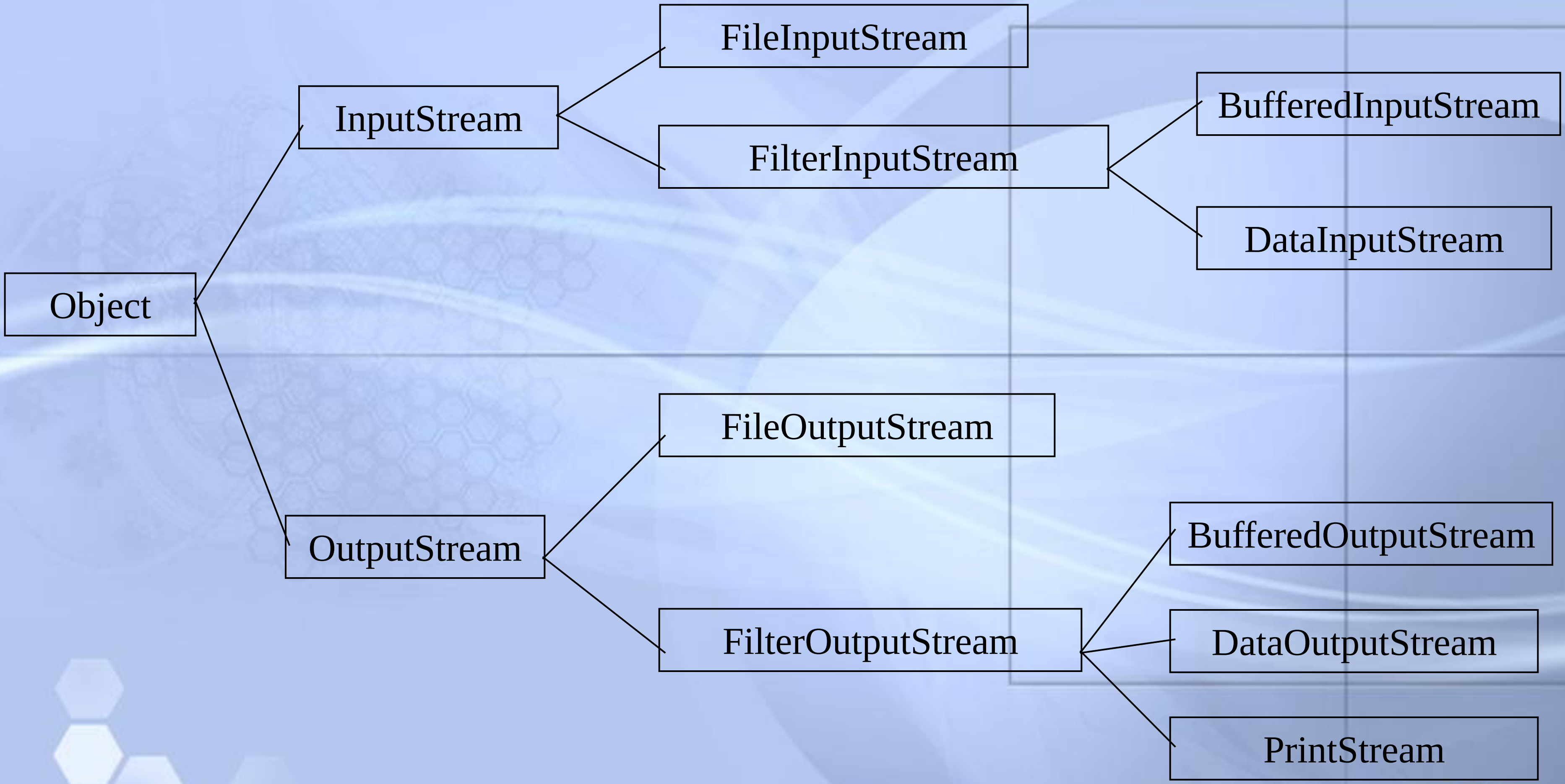
Process: Printing

- **CharacterStreams**
  - PrintWriter

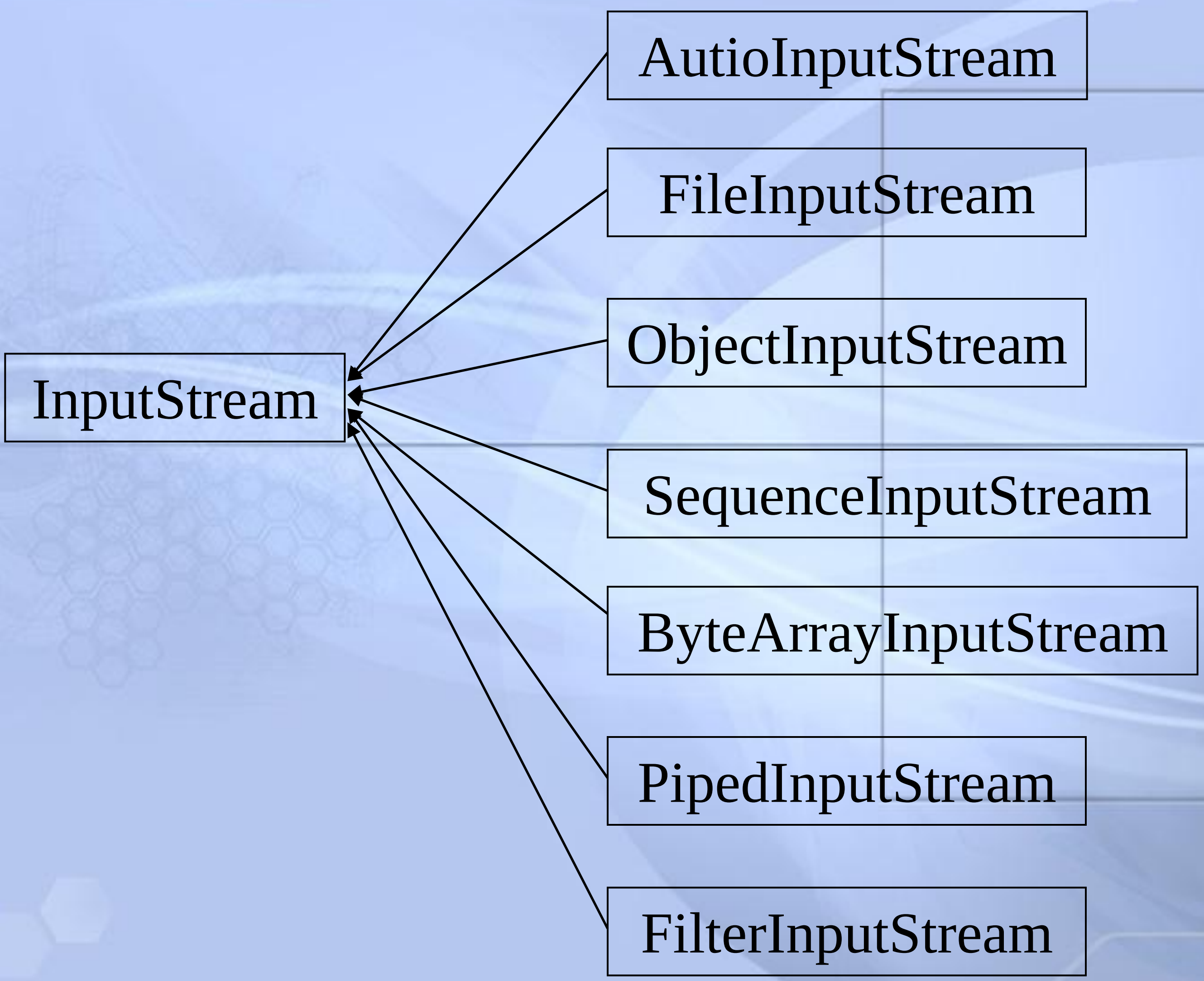
- **Byte Streams**
  - PrintStream

Byte Stream

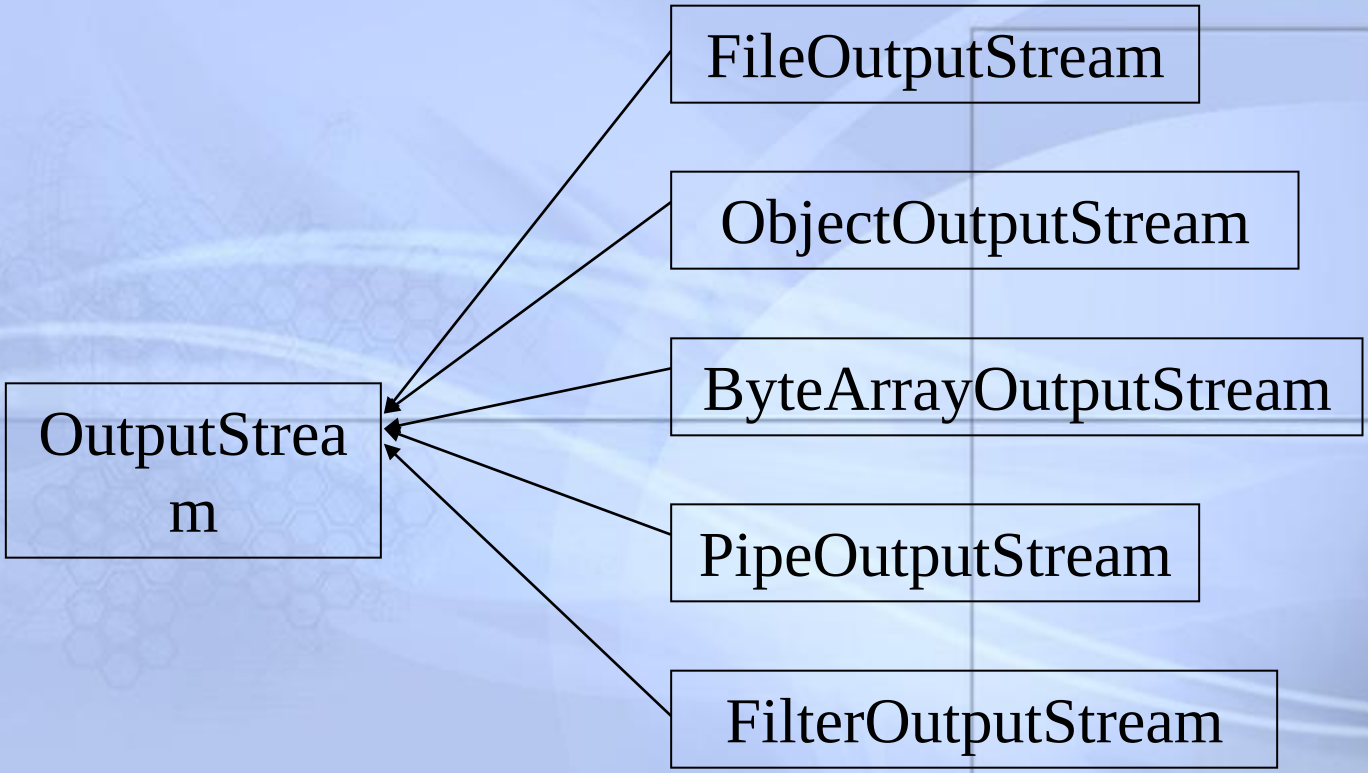
Byte Streams (Binary Streams)



Byte Streams



Byte Streams



InputStream Methods

- **The three basic read( ) methods**
  - int read( )
  - int read(byte[])
  - int read(byte[], int, int)

- **The other methods**
  - void close( )
  - int available( )
  - skip(long)
  - boolean markSupported( )
  - void mark(int)
  - void reset( )

OutputStream Methods

- **The three basic write( ) methods**
  - void write(int)
  - void write(byte[])
  - void write(byte[], int, int)

- **The other methods**
  - void close( )
  - void flush( )

Basic Stream Classes

- FileInputStream and FileOutputStream
- BufferedInputStream and BufferOutputStream
- DataInputStream and DataOutputStream
- PipedInputStream and PipeOutputStream

Example: Writing to a file using FileOutputStream

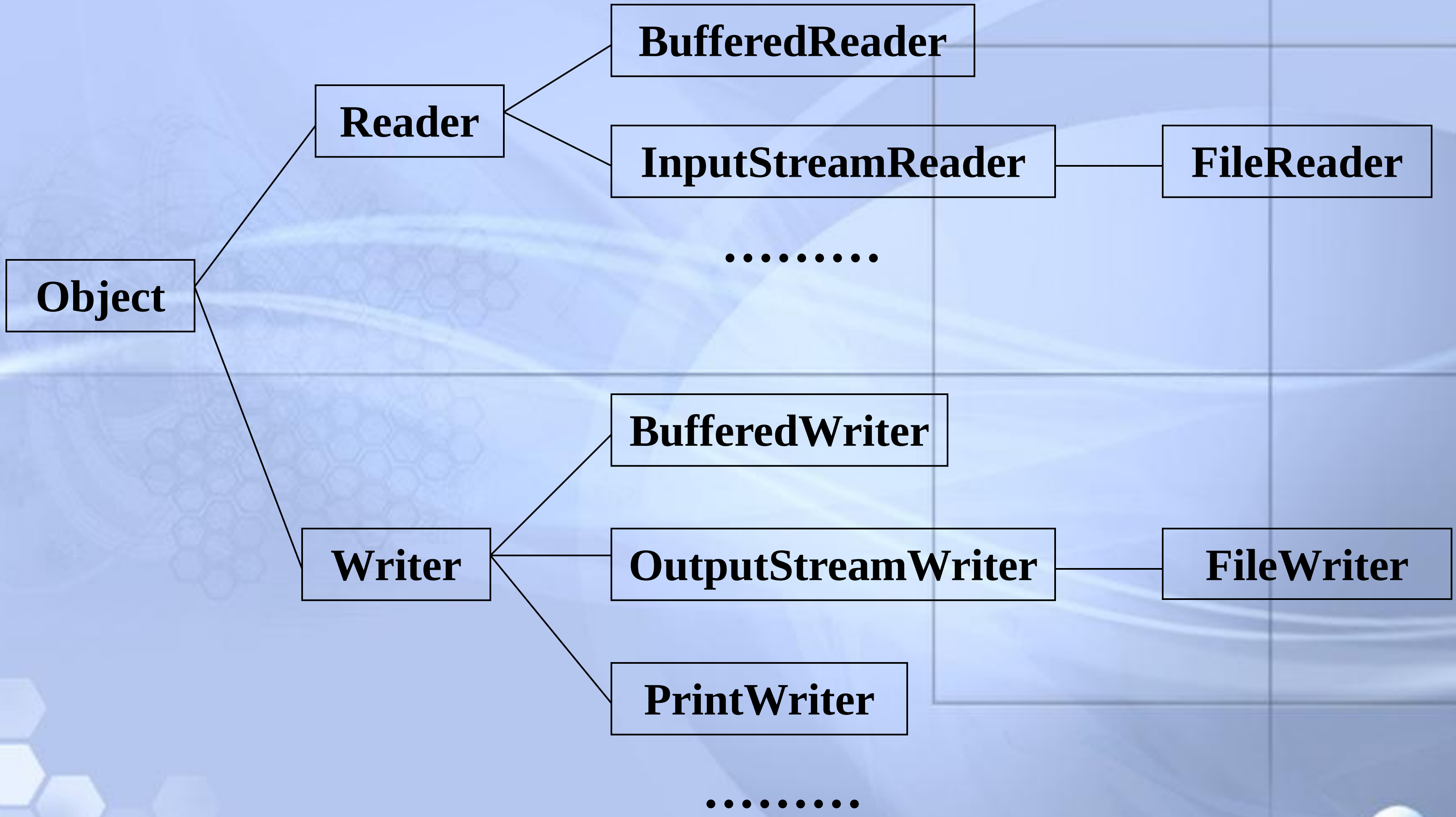
```
import java.io.*;
public class FTest {
    public static void main(String[] args){
    try{
            File file=new File("C:\\bbb.txt");
            FileOutputStream  fOut=new FileOutputStream(file);
            byte[] data={97,98,99,100,101};
            fOut.write(data);
            fOut.close();
    }catch(IOException e)
    {
            System.err.println(e.getMessage());
    }
}
```

Example: Reading from a file using FileInputStream

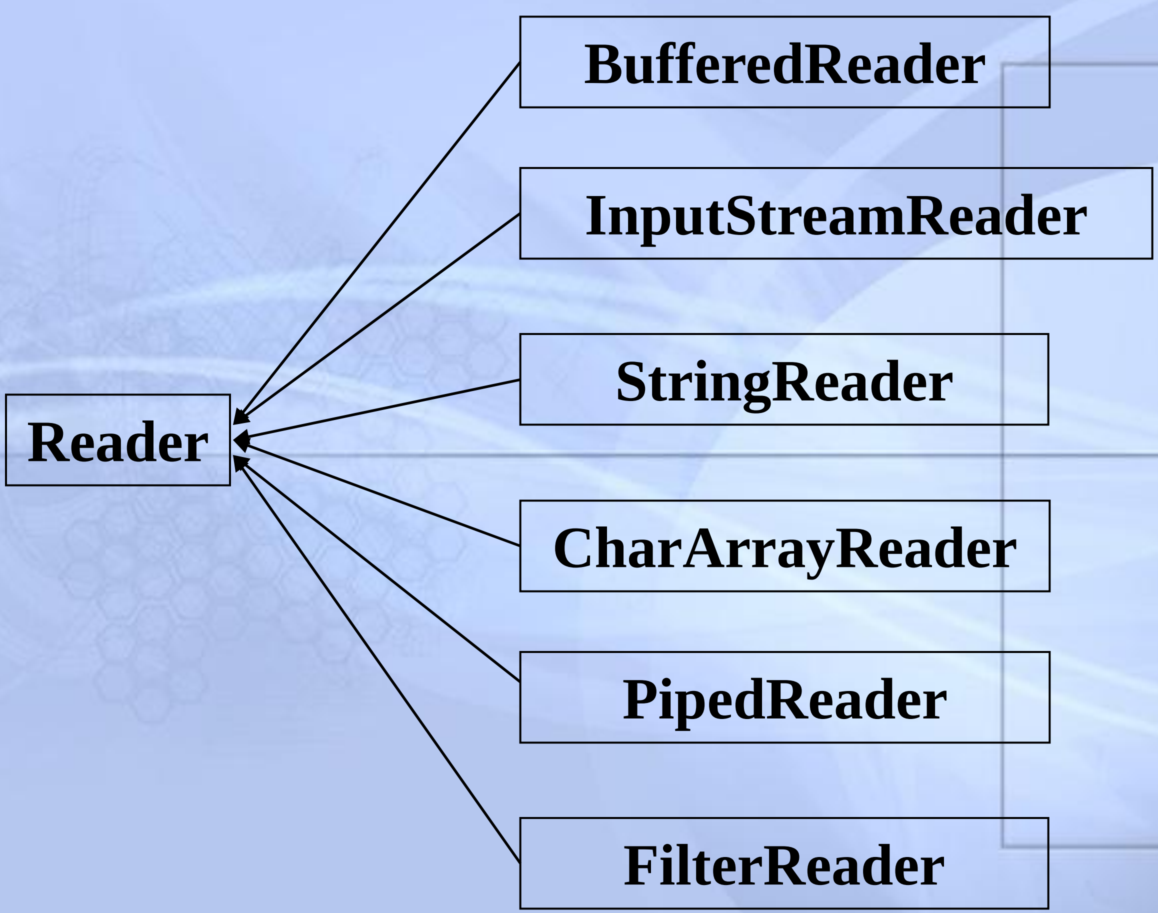
```
import java.io.*;
public class FTest {
    public static void main(String[] args){
    try{
            File file=new File("C:\\bbb.txt");
           FileInputStream fIn=new FileInputStream(file);
           byte[] data=new byte[(int)file.length()];
           fIn.read(data);
           fIn.close();
           for(int i=0; i<data.length; i++)
                       System.out.print((char)data[i]);
    }catch(IOException e)
    { System.err.println(e.getMessage()); }
}
```

Character Stream

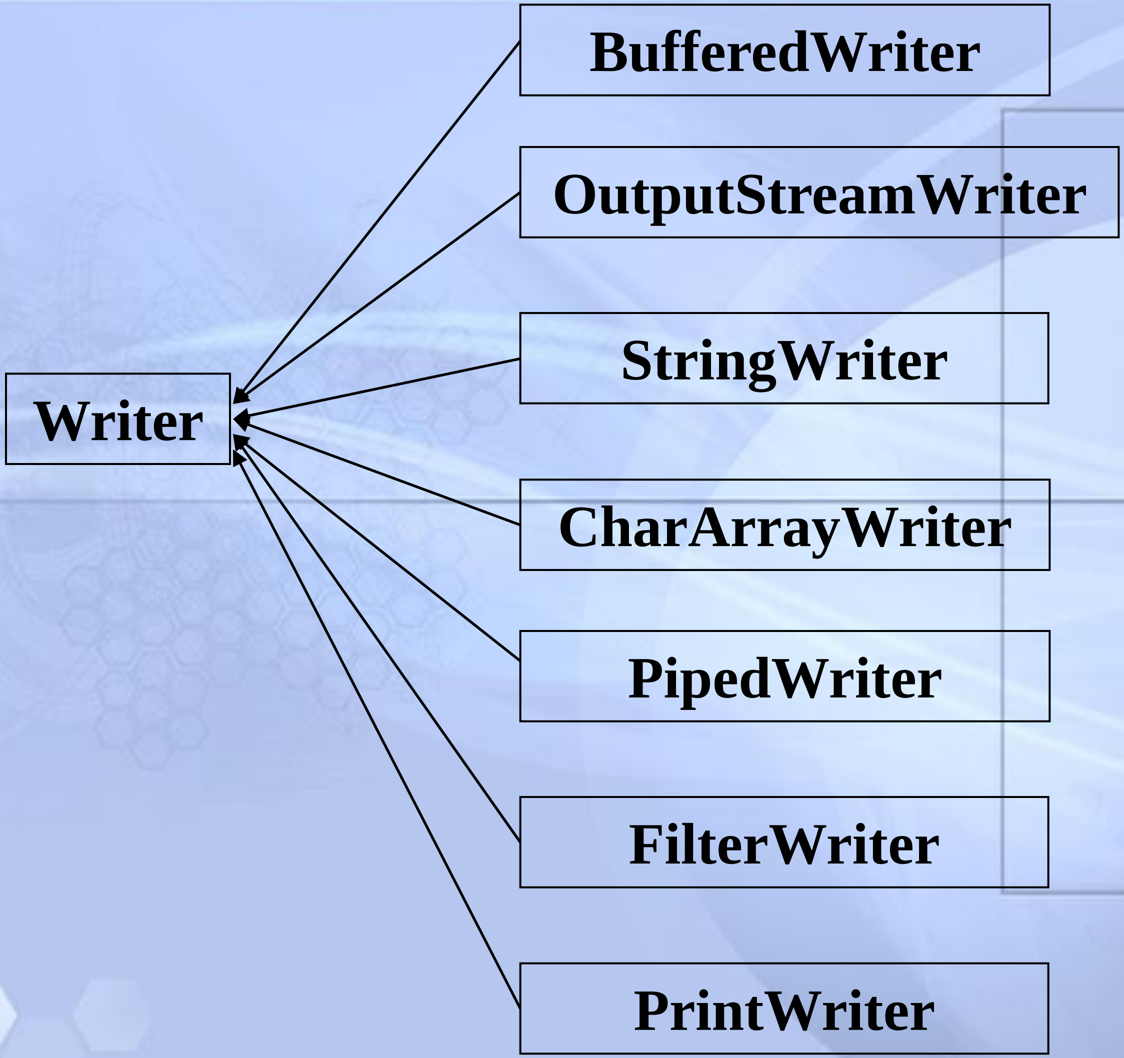
Character Streams



Character Streams



Character Streams



Opening a stream by using Reader

- A FileReader is a 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.

Example of using a stream

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

- The fileReader.read() method reads one character and returns it as an integer, or -1 if there are no more characters to read.
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)

Manipulating the input data

- Reading characters as integers isn't usually what you want to do.
- A BufferedReader will convert integers to characters; it can also read whole lines.
- The constructor for BufferedReader takes a FileReader parameter:

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

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

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

Closing

- A stream is an expensive resource.
- There is a limit on the number of streams that you can have open at one time.
- You should not have more than one stream open on the same file.
- You must close a stream before you can open it again.
- ***Always close your streams!***

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

Character Streams

```
import java.io.*;
public class CountSpace {
  public static void main(String[] args)
     throws IOException
  {
    Reader in;
    if (args.length == 0)
      in = new InputStreamReader(System.in);
    else
      in = new FileReader(args[0]);
    int ch;
    int total;
    int spaces = 0;
    for (total = 0; (ch = in.read()) != -1; total++) {
      if (Character.isWhitespace((char) ch))
        spaces++;
    }
    System.out.println(total + " chars "  + spaces + " spaces");  }
}
```

The abstract classes for reading and writing streams of characters are **Reader** and **Writer**.

The abstract class Reader provides a character stream analogous to the byte stream InputStream and the methods of Reader essentially mirror those of InputStream.

Run:
Java CountSpace CountSpace.java
Result:
520 characters 172 spaces

The conversion streams **InputStreamReader** and **OutputStreamWriter** translate between character and byte streams using either a specified character set encoding or the default encoding for the local system.

Using Writer and PrintWriter

- A FileWriter is a used to connect to a file that will be used for output:

```
FileWriter fileWriter = new FileWriter(fileName);
PrintWriter   printWriter =new PrintWriter(fileWriter);
String s="hello";
printWriter.println(s);
```

Closing stream

```
try {
            printWriter.flush( );
            printWriter.close( );

  }catch(Exception e)
   {

   }
```

Flushing the buffer

- When you put information into a buffered output stream, it goes into a buffer.
- The buffer may not be written out right away.
- If your program crashes, you may not know how far it got before it crashed.
- Flushing the buffer is forcing the information to be written out.

PrintWriter

- Buffers are automatically flushed when the program ends normally.

- Usually it is your responsibility to flush buffers if the program does not end normally.

- PrintWriter can do the flushing for you

  public PrintWriter(OutputStream out, boolean autoFlush)

Example: Writing to a file using FileWriter

```
import java.io.*;
public class FTest {
     public static void main(String[] args){
     try{
                File file=new File("C:\\bbb.txt");
               FileWriter fWriter=new FileWriter(file);
               char[] data={'a','b','c','d','e'}; // String data="abcde";
               fWriter.write(data);
               fWriter.close();
     }catch(IOException e)
     {
               System.err.println(e.getMessage());
     }
}
```

Example: Writing to a file using PrintWriter

```
import java.io.*;
public class FTest {
    public static void main(String[] args){
    try{
            File file=new File("C:\\bbb.txt");
           FileWriter fWriter=new FileWriter(file);
           PrintWriter pWriter=new PrintWriter(fWriter);
           // PrintWriter pWriter=new PrintWriter(fWriter,true); // append
           String data="abcde";
           pWriter.println(data);
           pWriter.close();
    }catch(IOException e)
    { System.err.println(e.getMessage()); }
}
```

Example: Reading from a file using FileReader

```
import java.io.*;
public class FTest {
    public static void main(String[] args){
    try{
            File file=new File("C:\\bbb.txt");
            FileReader fReader=new FileReader(file);
            char[] data=new char[(int)file.length()];
            fReader.read(data);
            fReader.close();
            for(int i=0; i<data.length; i++)
                    System.out.print(data[i]);
    }catch(IOException e)
    { System.err.println(e.getMessage()); }
}
```

Example: Reading from a file using BufferedReader

```
import java.io.*;
public class FTest {
    public static void main(String[] args){
    try{
             File file=new File("C:\\bbb.txt");
            FileReader fReader=new FileInputStream(file);
            BufferedReader bReader=new BufferedReader(fReader);
            String st=bReader.readLine();
            System.out.print(st);
            bReader.close();
    }catch(IOException e)
    { System.err.println(e.getMessage()); }
}
```

Data Byte Stream

The Data Byte Streams

- **DataInput** and **DataOutput**
- These interfaces define methods that transmit primitive types across a stream.
- provide Read / Write methods

Read	**Write**	**Type**
readBoolean	writeBoolean	boolean
readChar	writeChar	char
readByte	writeByte	byte
readShort	writeShort	short
readInt	writeInt	int
readLong	writeLong	long
readFloat	writeFloat	float
readDouble	writeDouble	double
readUTF	writeUTF	String(in UTF format)

Writing by using Data Byte Streams

```
public static void writeData(double[] data, String file) throws IOException
{
  OutputStream fOut = new FileOutputStream(file);
  DataOutputStream out = new DataOutputStream(fOut);

  out.writeInt(data.length)

  for(double d : data)
     out.writeDouble(d);

  out.close();
}
```

Writing by using Data Byte Streams

```
public static double[] readData(String file) throws IOException
{
  InputStream fin = new FileInputStream(file);
  DataInputStream in = new DataInputStream(fin);

  double[] data = new double[in.readInt()];

  for (int i = 0; i < data.length; i++)
    data[i] = in.readDouble();

   in.close();

  return data;
}
```

Reading Keyboard Input

```
import java.io.*;
public class Test{
    public static void main(String[] args){
        try{
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
                        String name=null;
                        int age=0;
                        System.out.print("Enter your name : ");
                        name=br.readLine();
                        System.out.print("Enter your age : ");
                        age=Integer.parseInt(br.readLine());
                        System.out.println("Your Name = " + name);
                        System.out.println("Your Age = " + age);
            }catch(IOException e)
            {System.err.println(e.getMessage());}
     }
}
```

Object Serialization

Serialization

- You can also read and write *objects* to files.
- Serialization: process of converting an object’s representation into a stream of bytes.
- Deserialization: reconstituting an object from a byte stream
- Support process of reading and writing objects

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

```
ObjectOutputStream objectOut =
  new ObjectOutputStream(
    new BufferedOutputStream(
      new FileOutputStream(fileName)));

objectOut.writeObject(serializableObject);

objectOut.close( );
```

Reading objects from a file

```
ObjectInputStream objectIn =
  new ObjectInputStream(
    new BufferedInputStream(
      new FileInputStream(fileName)));

myObject = (itsType)objectIn.readObject( );

objectIn.close( );
```

Example : Writing and Reading Date Object

```
FileOutputStream out = new
        FileOutputStream("theTime");

ObjectOutputStream s = new
            ObjectOutputStream(out);

s.writeObject("Today");
s.writeObject(new Date());
s.flush();

s.close();
```

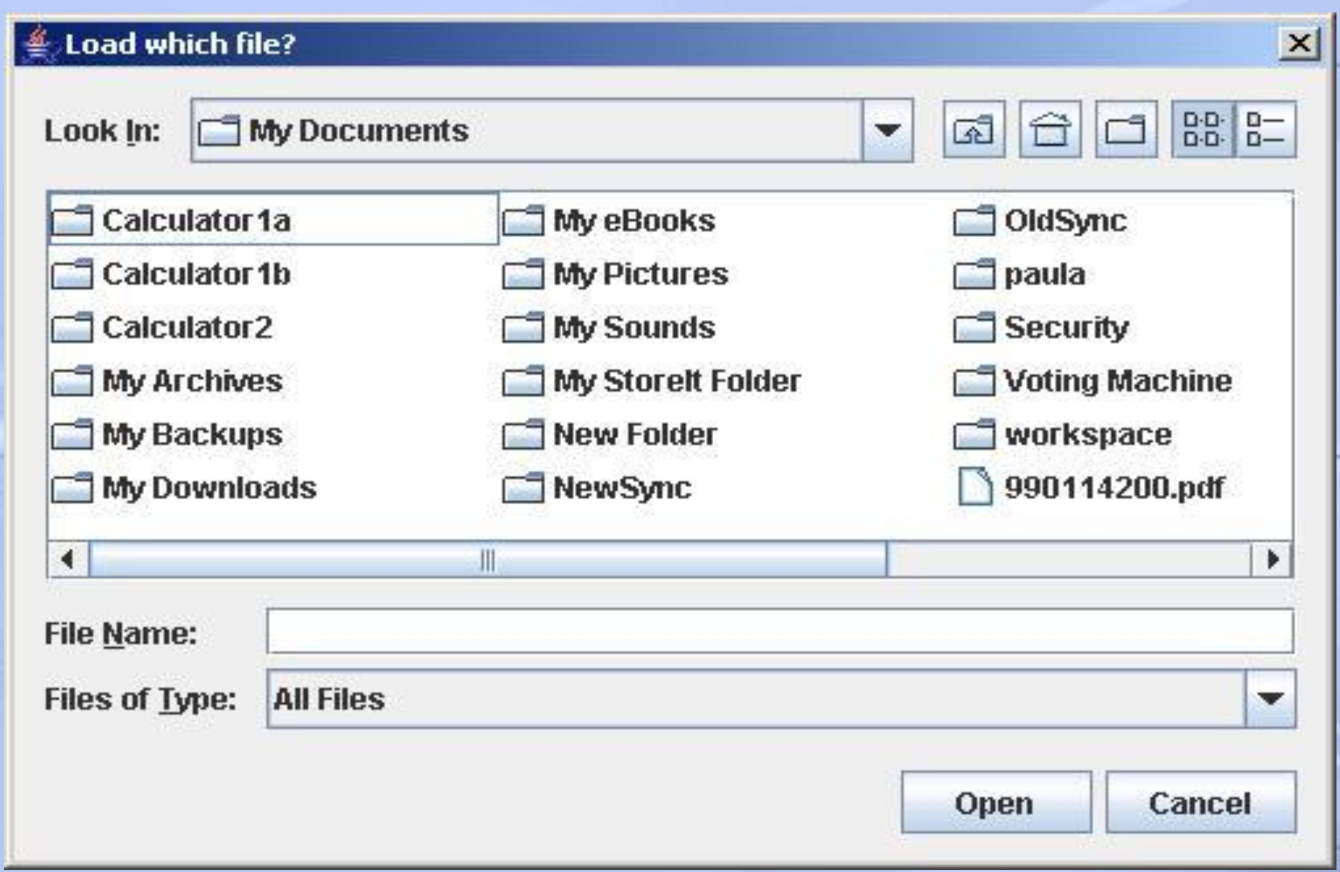
```
FileInputStream in = new
        FileInputStream("theTime");

ObjectInputStream s = new
         ObjectInputStream(in);

String today = (String)s.readObject();
Date date = (Date)s.readObject();

s.close();
```

JFileChooser

JFileChoosers

- The JFileChooser class displays a window from which the user can select a file.
- The dialog window is modal--the application cannot continue until it is closed.
- Applets cannot use a JFileChooser, because applets cannot access files.

Typical JFileChooser window



JFileChooser constructors

- JFileChooser()
  - Creates a JFileChooser starting from the user's directory

- JFileChooser(File ***currentDirectory***)
  - Constructs a JFileChooser using the given File as the path

- JFileChooser(String ***currentDirectoryPath***)
  - Constructs a JFileChooser using the given path

Useful JFileChooser methods

- int showOpenDialog(Component ***enclosingJFrame***);
  - Asks for a file to read; returns a flag
- int showSaveDialog(Component ***enclosingJFrame***);
  - Asks where to save a file; returns a flag
- Returned flag value may be:
  - JFileChooser.APPROVE_OPTION
  - JFileChooser.CANCEL_OPTION
  - JFileChooser.ERROR_OPTION

Useful JFileChooser methods (cont.)

File getSelectedFile()

– showOpenDialog and showSaveDialog return a flag telling what happened, but don't return the selected file.

– After we return from one of these methods, we have to ask the JFileChooser what file was selected.

– If we are saving a file, the File may not actually exist yet

Using a File

- Assuming that we have successfully selected a File:

  -
    ```
    File file = chooser.getSelectedFile();
    if (file != null) {
        String fileName = file.getCanonicalPath();
        FileReader fileReader = new FileReader(fileName);
        BufferedReader reader = new BufferedReader(fileReader);
     }
    ```

  -
    ```
    File file = chooser.getSelectedFile();
    if (file != null) {
        String fileName = file.getCanonicalPath();
        FileOutputStream stream = new FileOutputStream(fileName);
        writer = new PrintWriter(stream, true);
    }
    ```

Creating a Random Access File

- With the file name

myRAFile = new RandomAccessFile(String name, String mode);

- With a File object

myRAFile = new RandowAccessFile(File file, String mode);

- Example:

RandomAccessFile myRAFile;

myRAFile = new RandomAccessFile("db/stock.mdb","rw");

Random Access Files

- Long getFilePointer( )
  - return the current location of the file pointer.

- Void seek(long pos)
  - set the file pointer to the specified absolute position.

- Long length( )
  - return the length of the files.

The Supercomputing People

Corporate Profile

Thank you!