JAVA I/O

Objectives

- Understand the basic principles of stream I/O in java

- Understand how the Decorator design pattern is used in Java I/O libraries

- Understand the File class

What is a Design Pattern?

- A solution to a recurring problem
- A model or abstraction
- Design, not implementation
- Not a primitive building block

Why Use Design Patterns?

- Speed
  - A lot of optimization work may have gone into a design pattern
- Quality
  - Don’t reinvent the wheel
- Functionality
  - A design pattern is a pre-existing solution to a problem

- And…
  - Flexibility?
  - Extensibility?
  - Reusability?

Classes of Design Patterns

- Creational Patterns
  Abstract the construction of objects

- Structural Patterns
  Define a specific data structure

- Behavioral Patterns
  Define the behavior of a program

Design Patterns Reference

- Design Patterns; Gamma, Helm, Johnson, and Vlissides; Addison Wesley, 1995.

- Design Patterns in Java Tutorial from tutorialspoint.

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns

- Adaptor
- Bridge
- Composite
- Decorator ⬅
- Facade
- Flyweight
- Proxy

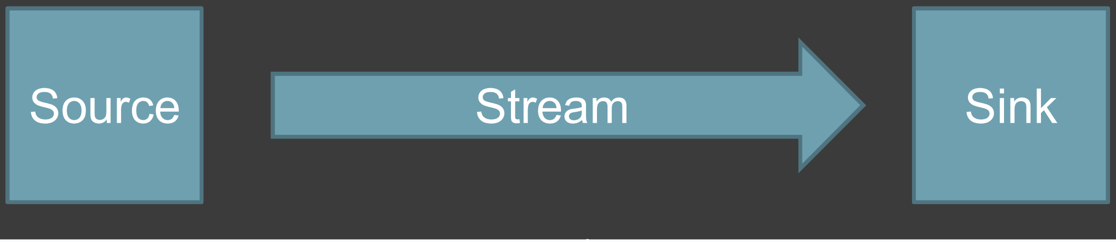
Structural Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Composite
- Mediator

- Momento
- Observer
- Statte
- Strategy
- Template Method
- Visitor

The Stream Model

- All Data is viewed as either a *source* or a *sink*



Source	Sink
File	Program's memory
Program's memory	File
String	Object
Object	String
Network Connection	
Laboratory Equipment	
Phone	

The Decorator Pattern

- Augments the functionality of an object
- Decorator object wraps another object
  - The Decorator has a similar interface
  - Calls are relayed to the wrapped object ...
  - ... but the Decorator can interpolate additional actions
- Example: BufferedOutputStream adds buffering to OutputStream

The Stream Model

- Use different streams for different jobs
  - stdin
  - stdout
  - stderr
  - socket (newtwork connection)
  - Write your own
- Streams are ubiquitous
- Streams form the fundamental I/O paradigm in Java

abstract class OutputStream

- Key methods
  - abstract void write() throws IOException
  - void write(byte[] b) throws IOException
  - void close() throws IOException

OutputStream: Some Concrete Sublcasses

- class ByteArrayOutputStream
  - Sink is byte[]
- class FileOutputStream
  - Sink is file
- class PipedOutputStream
  - Sink is a pipe to another thread
- class FilterOuputStream
  - Sink is another stream
- class ObjectOutputStream

OutputStream Example

```
ByteArrayOutputStream outStream   =
    new ByteArrayOutputStream( 2048 );

PrintStream printStream = new PrintStream(outStream, true);
printStream.println( "R. E. Cipient" );
printStream.print( 116 );
printStream.println( " Forrest Ave." );
printStream.println( "Los Cruces, NM 11234" );

System.out.println( outStream );
try
{
    outStream.close();
}
catch ( IOException exc )
{
    System.exit( 1 );
}
```

FileOutputStream Example

```
PrintStream printStr     = null;
FileOutputStream outStr = null;
try
{
    outStr = new FileOutputStream( "temp.txt" );
    printStr = new PrintStream( outStr );
    …
}
catch ( IOException exc )
{
    exc.printStackTrace();
}
...
```

Continued on next slide

Continued from previous slide

FileOutputStream Example

```
...
finally
{
    if ( outStr != null )
    {
        try
        {
            outStr.close();
        }
        catch ( IOException exc )
        {
            exc.printStackTrace();
        }
    }
}
```

Try-With-Resources

- Add parentheses to try
- Declare closeable resources in parentheses

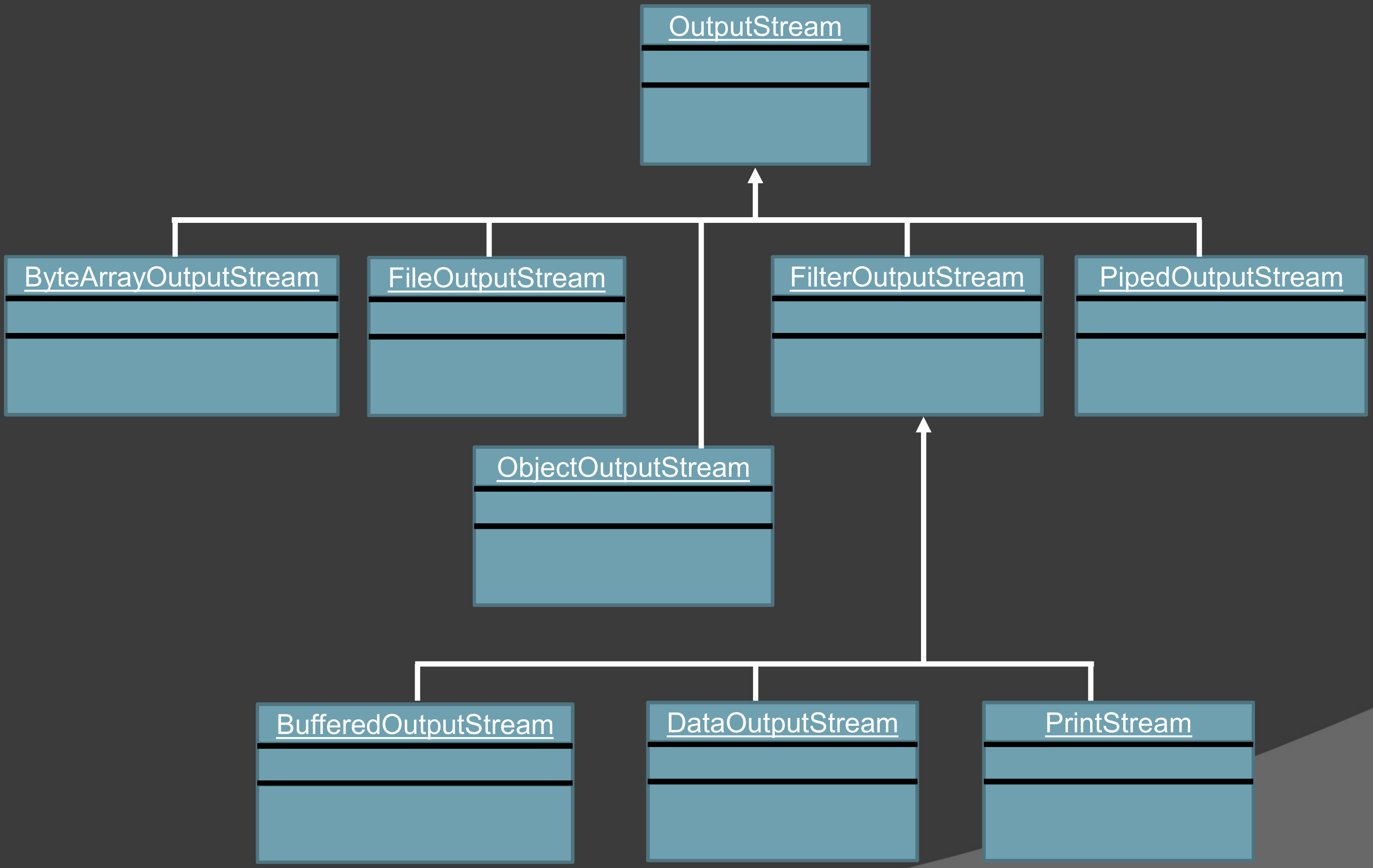
```
try ( Closeable resources declared here
{
    ...
}
catch ( IOException exc )
{
    ...
}
```

- Resources automatically closed on completion

Try-With-Resources Example

```
try (
    FileOutputStream outStr =
        new FileOutputStream( "temp.txt" );
    PrintStream printStr = new PrintStream( outStr );
)
{
    for ( int inx = 0 ; inx < 10 ; ++inx )
    {
        printStr.print( "Line Number: " );
        printStr.println( inx );
    }
}
catch ( IOException exc )
{
    exc.printStackTrace();
}
```

Output Stream Hierarchy



Writers

Writer – Abstract superclass for all classes that write streams of type char. Always wrap an output stream

- BufferedWriter
- CharArrayWriter
- FileWriter
- FilterWriter
- OutputStreamWriter
- PipedWriter
- PrintWriter
- StringWriter

abstract class Writer

Principal Methods:

- write(char[]) – Writes the chars in the array
- write(int) – writes a single character
- write(String) – writes a string
- close() – Closes the stream

class BufferedWriter

Adds buffering to an input stream.

Principal methods:

- BufferedWriter(Writer) – Constructor
- NewLine() – Writes a line separator
- writer(char[]) – Writes an array
- write(int) – writes a single character

BufferedWriter Example

```
try(
    FileWriter fileWriter  =
        new FileWriter( "WriterTest.txt" );
    BufferedWriter bufWriter =
        new BufferedWriter( fileWriter );
)
{
    ...
}
catch ( IOException exc )
{
    exc.printStackTrace();
    System.exit( 1 );
}
```

class CharArrayWriter

Writes to a character array.

Principal methods:

- CharArrayWriter() – constructor
- toCharArray() – returns a copy of the buffer

```
CharArrayWriter writer  = new CharArrayWriter();
```

class FileWriter

Knows how to write to a file.

Principal methods:

- FileWriter(File) – constructor
- FileWriter(File,boolean*) – constructor
- FileWriter(String) – constructor
- FileWriter(String,boolean*) – constructor

```
FileWriter fileWriter  =
    new FileWriter( "WriterTest.txt" );
```

*True to append, false to overwrite

class OutputStreamWriter

Bridge between byte streams and char streams.

Principal methods:

- OutputStreamWriter(OutputStream) – constructor

```
// Not very useful
OutputStreamWriter writer =
   new OutputStreamWriter( System.err );
```

class PrintWriter

Formats objects as text and writes to text stream

Principal methods:

- PrintWriter(File) – constructor
- PrintWriter(OutputStream) – constructor
- PrintWriter(OutputStream, boolean*)
- PrintWriter(String fileName) – constructor
- PrintWriter(Writer) – constructor
- PrintWriter(Writer, boolean*)

*Autoflush

Continued on next slide

Continued from previous slide

class PrintWriter

- append(char) – append a character
- print(char[]) – prints an array of characters
- format(String, Object) – just like String.format()
- print(boolean) – prints a boolean
- print(X) – prints (X)
  - Many overloads; think “System.out.print()”
- println(X) – prints (X) followed by line separator
  - Many overloads; think “System.out.println()”

Continued on next slide

Continued from previous slide

class PrintWriter

- write(int) – writes a char
- write(char[]) – writes an array of chars
- write(String) – writes a string
- checkError() – checks the error state of stream

Continued on next slide

Continued from previous slide

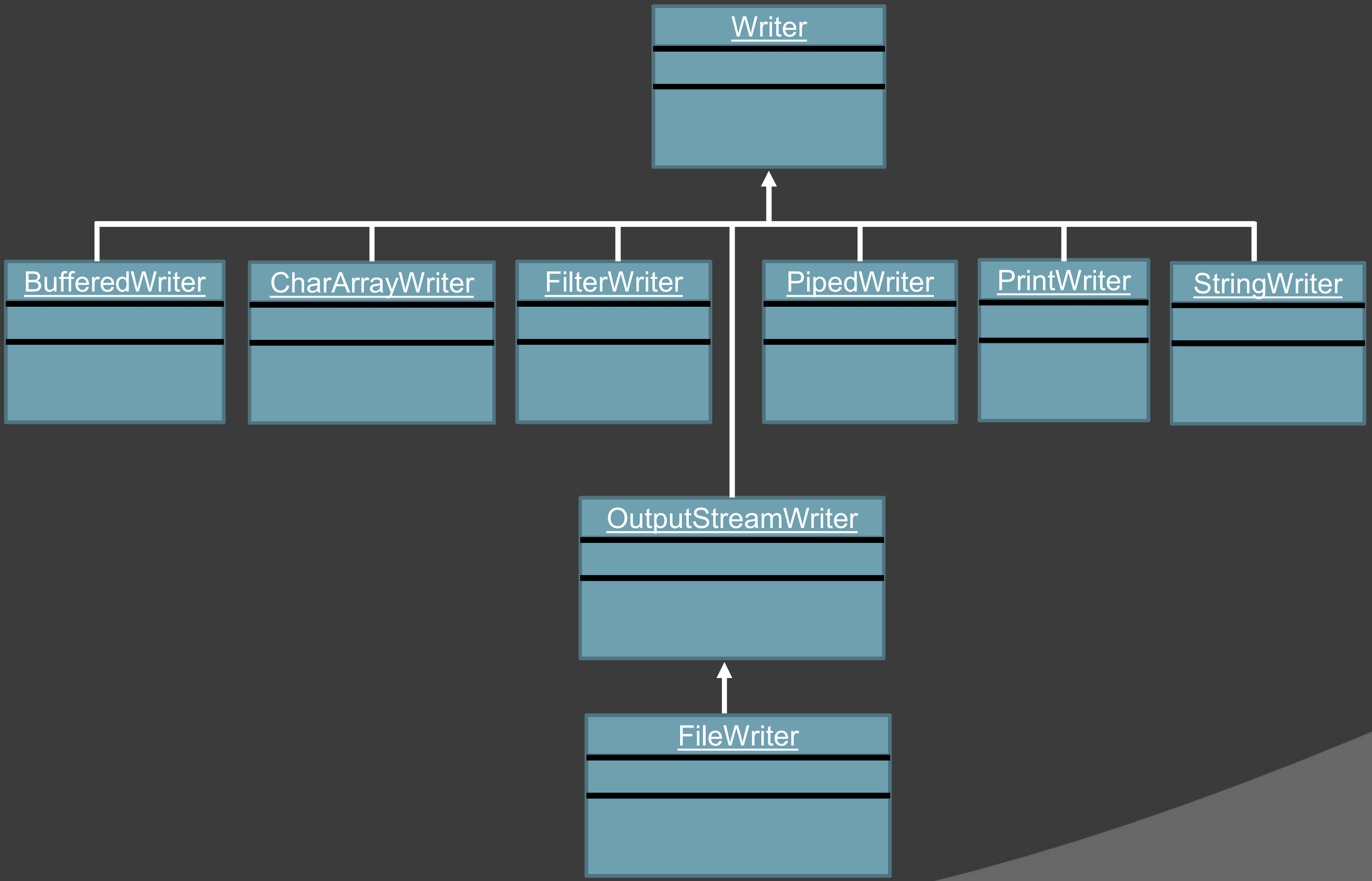
class PrintWriter

- Methods (except constructors) never throw an exception
- To check the state of the stream, use checkError()

see also: class PrintStream

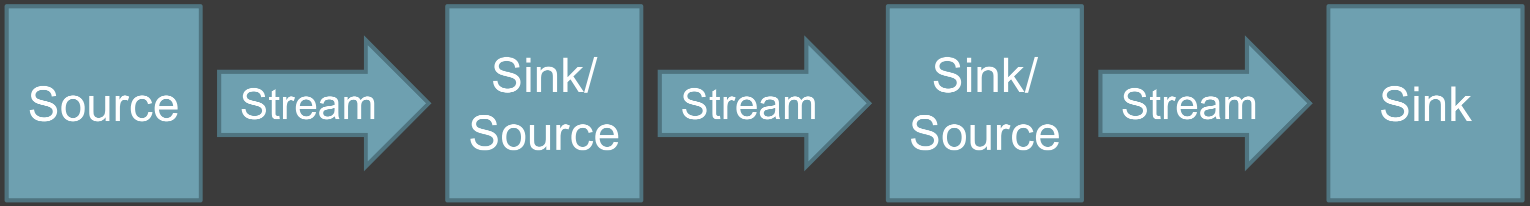
- Java 1.0 legacy
- Very similar to PrintWriter
- Uses platform default encoding
  - This can lead to platform dependencies
- Has methods for writing raw bytes
  - PrintWriter preferred for writing characters
  - DataOutputStream preferred for writing raw data
- Handles flushing differently
  - PrintWriter flushing is more robust

Writer Hierarchy



class FilterOutputStream

- Superclass for chaining streams
- Sink of one stream is source for another



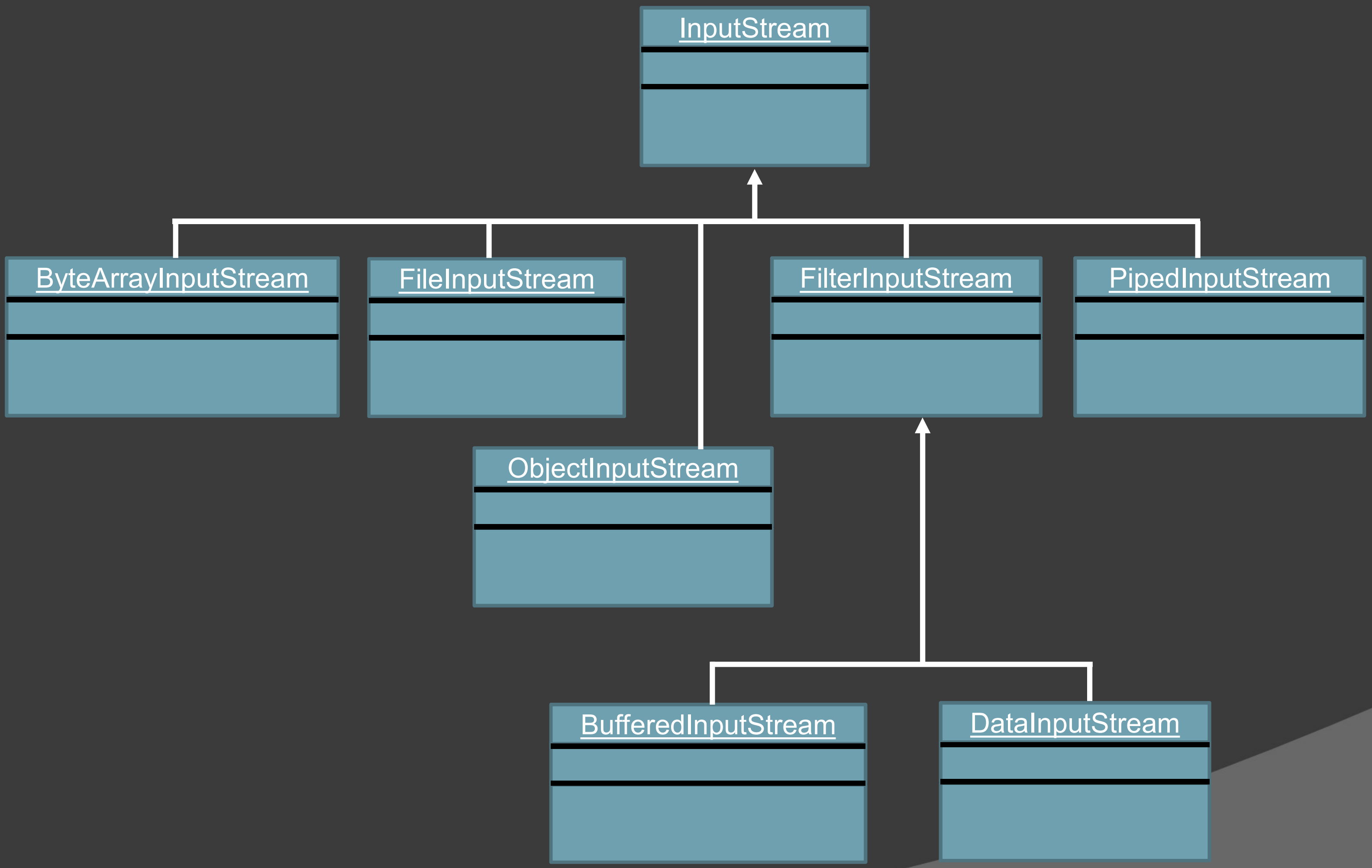
abstract class InputStream

- Key methods:
  - abstract intread() throws IOException
  - void read(byte[] b) throws IOException
  - void close() throws IOException

InputStream: Some Concrete Subclasses

- class ByteArrayInputStream
  - Source is byte[]
- class FileInputStream
  - Source is file
- class PipedInputStream
  - Source is a pipe from another thread
- class FilterInputStream
- class ObjectInputStream

Input Stream Hierarchy



Readers

Reader – Abstract superclass for all classes that read streams of type char. Always wrap an input stream.

- BufferedReader
- CharArrayReader
- FileReader
- FilterReader
- InputStreamReader
- PipedReader
- StringReader

abstract class Reader

Principal Methods:

- read() – read a single character
- read(char[]) – Reads characters into an array
- reset() – Resets to start of stream, <u>if supported</u>.
- close() – Closes the stream

class CharArrayReader

Reads from a char array.

Principal methods:

- CharArrayReader(char[] buf) – constructor

```
char[] array = …
CharArrayReader reader  =
    new CharArrayReader( array );
```

class BufferedReader

Adds buffering to an input stream.

Principal methods:

- BufferedReader(Reader) – constructor
- readLine() – reads a line of text
- skip(long) – skips characters

```
char[]          array     = …
CharArrayReader charRdr   =
    new CharArrayReader( array );
BufferedReader  bufReader =
    new BufferedReader( charRdr );
```

class FileReader

Reads from a file.

Principal methods:

- FileReader(File) – constructor
- FileReader(String) – constructor

```
FileReader reader = new FileReader( “tmp.txt );
```

class InputStreamReader

Bridge between byte streams and char streams.

Principal methods:

- InputStreamReader(InputStream) – constructor

```
InputStreamReader   reader  =
    new InputStreamReader( System.in );
```

class StringReader

Reads from a string.

Principal methods:

- StringReader(String) – constructor

```
String       buf     = …
StringReader reader  = new StringReader( buf );
```

Detecting End-of-Stream (1)

- Reading a string: check for null return

```
String  line    = bufReader.readLine();
while ( line != null )
{
    System.out.println( line );
    line = bufReader.readLine();
}
```

Detecting End-of-Stream (2)

- Reading a byte: check for -1 return

```
int next    = dStream.read();
while ( next != -1 )
{
    next &= 0xff;
    System.out.println( next );
    next = dStream.read();
}
```

Digression: Fun With Expressions

- This code is not recommended because you are duplicating *next = dStream.read()*.

```
int next    = dStream.read();
while ( next != -1 )
{
    next &= 0xff;
    System.out.println( next );
    next = dStream.read();
}
```

Continued on next slide

Continued from previous slide

Digression: Fun With Expressions

- A for loop would be better, but you still have duplicate code

```
for ( String line = bufReader.readLine() ;
      line != null ;
      line = bufReader.readLine()
     )
    System.out.println( line );
```

Continued on next slide

Continued from previous slide

Digression: Fun With Expressions

- Consider using this technique

The parentheses around *line = buf…* are required to resolve precedence issues

```
String  line    = null;
while ( (line = bufReader.readLine()) != null )
    System.out.println( line );
```

Detecting End-of-Stream (3)

- Reading an array: check length == -1

```
byte[]  bytes   = new byte[BUF_SIZE / 3];
int     len     = 0;
while ( (len = dStream.read( bytes )) != -1 )
{
    for ( int inx = 0 ; inx < len ; ++inx )
        System.out.print( bytes[inx] );
    System.out.println();
}
```

Detecting End-of-Stream (4)

- Reading a char: check return -1
  - Note: ‘\uFFFF’ is *not* a valid Unicode character

```
int     ccc = 0;
while ( (ccc = reader.read()) != -1 )
    System.out.print( (char)ccc );
System.out.println();
```

Detecting End-of-Stream (5)

- Reading a byte or char in some classes:
  - DataInputStream.readChar()
  - DataInputStream.readByte()
- Can only catch EOFException

🚫

```
    while ( true )
    {
        int ccc = dStream.readChar();
        System.out.print( (char)ccc );
    }
}
catch ( EOFException exc )
{
    System.out.println();
}
```

Continued on next slide

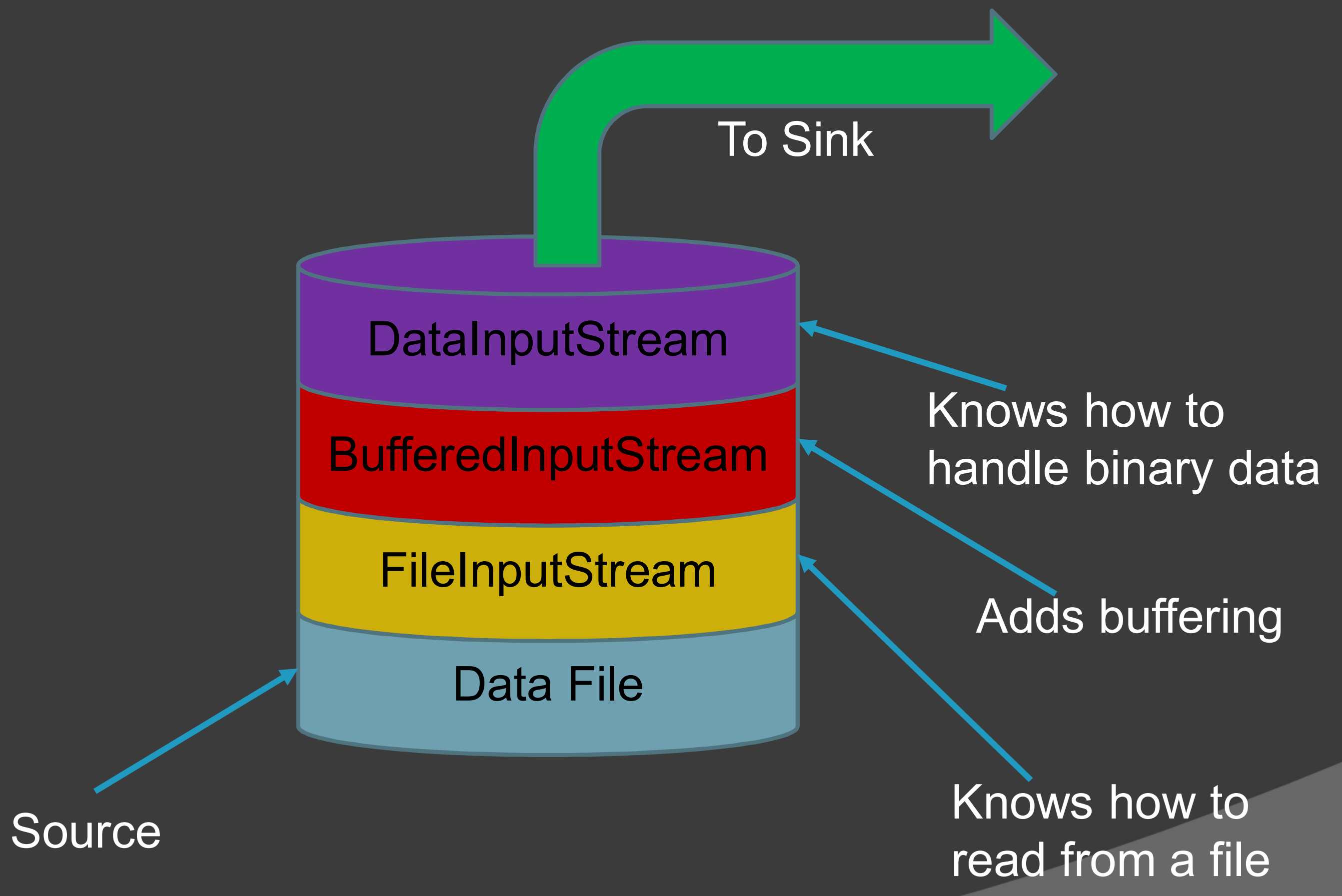
Continued from previous slide

Detecting End-of-Stream (5)

- Catching EOFException is bad practice
- If forced, consider using a one-element array

```
byte[]  next    = new byte[1];
int     len     = dStream.read( next );
while ( len == 1 )
{
    next[0] &= 0xff;
    System.out.println( next[0] );
    len = dStream.read( next );
}
```

Layered Streams



Layered Streams Demo

```
try (
    FileInputStream inStr =
        new FileInputStream( "temp.txt” );
    InputStreamReader strReader =
        new InputStreamReader( inStr );
    BufferedReader bufReader =
        new BufferedReader( strReader );
)
{
    String  line    = null;
    while ( (line = bufReader.readLine()) != null )
        System.out.println( line );
}
catch ( IOException exc )
{
    exc.printStackTrace();
    System.exit( 1 );
}
```

class File

- Represents a virtual file or directory
  - The file may or may not exist

- Constructors:
  - File(String full-path-name)
  - File(String path-to-directory, String file-name)
  - File(File path-to-directory, String file-name)

Continued on next slide

Continued from previous slide

class File

Common Methods:

- Common methods:
- boolean canExecute()
- boolean canRead()
- boolean canWrite()
- boolean createNewFile()
- static File createTempFile()
- boolean delete()
- boolean exists()
- boolean isDirectory()
- boolean isFile()
- long lastModified()
- mkdir()
- renameTo(File)
- File[] listFiles

class File, Example

```
private
static void enumerate( File dir, int indentLen )
{
    String  indent  =
        new String( spaces, 0, indentLen );
    File[]  files   = dir.listFiles();

    for ( File file : files )
    {
        String  name    = file.getName();
        System.out.println( indent + name );
        if ( file.isDirectory() )
            enumerate( file, indentLen + 4 );
    }
}
```

Redirecting stdin and stdout

- To redirect stdin
  - Create a PrintStream
  - Set using System.setIn()
- To redirect stdout
  - Create an input stream
  - Set using System.setOut()
- Restore original stdin and stdout when done.

Redirecting stdin and stdout: JUnit Test for DumbJavaCalc

- Given:

```
private static final double EPSILON     = .000001;
private File    tempIn;  // replacement for stdin
private File    tempOut; // replacement for stdout
private static final String[]   allExpressions =
{
   "3 + 5",         // 8
   "3+5",           // 8
   "  3  + 5  ",    // 8
   "-4 + -6",       // -10
   "+4++6",         //10
   // etc.
   "quit"
};
private static final double[]   expResults   =
    { 8, 8, 8, -10, 10 };
```

Continued on next slide

Redirecting stdin and stdout: JUnit Test for DumbJavaCalc

Continued from previous slide

```
@Before
public void setUp() throws IOException
{
    tempIn = File.createTempFile( "TempIn", ".tmp" );
    tempOut = File.createTempFile( "TempOut", ".tmp" );
}

@After
public void tearDown()
{
    tempIn.delete();
    tempOut.delete();
}
```

Continued on next slide

Redirecting stdin and stdout: JUnit Test for DumbJavaCalc

Continued from previous slide

```
@Test
public void goRightTest()
{
    PrintStream stdout  = System.out;
    InputStream stdin   = System.in;

    createInput();
    DumbJavaCalc.main( null );

    System.setOut( stdout );
    System.setIn( stdin );

    validateOutput();
}
```

Continued on next slide

Redirecting stdin and stdout: executeClientApp()

Continued from previous slide

```
try (
    PrintStream inData  = new PrintStream( tempIn );
    PrintStream outStream = new PrintStream( tempOut );
    FileInputStream inStream =
        new FileInputStream( tempIn );
)
{
    for ( String str : allExpressions )
        inData.println( str );
    System.setOut( outStream );
    System.setIn( inStream );
    DumbJavaCalc.main( null );
}
catch ( IOException exc )
{
    fail( exc.getMessage() );
}
```

Continued on next slide

Redirecting stdin and stdout: validateOutput()

Continued from previous slide

```
try (
       FileInputStream valStream =
           new FileInputStream( tempOut );
       InputStreamReader inReader =
           new InputStreamReader( valStream );
       BufferedReader reader =
           new BufferedReader( inReader );
   )
   {
       // throw away prompt
       assertNotNull( reader.readLine() );

       String  line    = null;
       int     limit   = expResults.length;
       ...
```

Continued on next slide

Redirecting stdin and stdout: validateOutput()

Continued from previous slide

```
    for ( int inx = 0 ; inx < limit ; ++inx )
    {
        assertNotNull(line = reader.readLine() );
        double actualResult = Double.parseDouble( line );
        assertEquals( expResults[inx],
                                         actualResult, EPSILON );
        // throw away prompt
        assertNotNull( reader.readLine() );
    }
}
catch ( IOException exc )
{
    exc.printStackTrace();
    fail( exc.getMessage() );
}
...
```

Properties

What is a Property?

- Key/value pair, key and value both strings
- Maintained at the system level
- Persistent

```
String userDir = System.getProperty( "user.dir" );
System.out.println( "*** " + userDir + " ***" );
```

java.util.Properties

Common methods:

- String getProperty(String key) returns null on failure
- String getProperty(String key, String default)
  returns default on failure
- Object setProperty(String key, String value)
  returns previous value, null if none
- void load(InputStream inStream) loads from a stream

Common System Properties

- Use System.getProperty(String key)

Key	Meaning
file.separator	Character used in pathnames, e.g. ‘/’ or ‘\’
java.class.path	Classpath used by class loader
java.version	JRE Version number
line.separator	String used to terminate lines, e.g. “\n” or “\n\r”
user.name	User name

Make Your Own Properties

```
business.name=The Small Consulting Group
business.street=1616 Index Ct.
business.city=Renton
business.state=WA
business.zip=98058
```

Properties Demo

```
private Color   fontColor       = null;
private Float   fontSize        = null;
private Integer fontStyle       = null;
private Color   backgroundColor = null;
public LoadPropertiesDemo()
{
    try(
        FileInputStream inStream =
            new FileInputStream( PROPERTIES );
    )
    {
        getProperties( inStream );
    }
    catch ( IOException exc )
    {
        exc.printStackTrace();
        System.exit( 1 );
    }
}
```

Continued on next slide

Properties Demo: getProperties()

Continued from previous slide

```
Properties  props    = new Properties();
props.load( stream );

String  sColor  = props.getProperty( "font.color" );
String  sSize   = props.getProperty( "font.size" );
String  sStyle  = props.getProperty( "font.style" );
String  sBColor = props.getProperty( "background.color" );

if ( sColor != null )
    fontColor = assembleColor( sColor );
if ( sSize != null )
    fontSize = Float.parseFloat( sSize );
if ( sStyle != null )
    fontStyle = deriveStyle( sStyle );
if ( sBColor != null )
    backgroundColor = assembleColor( sBColor );
```

What Are Resources?

- Miscellaneous files used by your application
- Stored in a directory in your application JAR file
- In Maven projects:
  - src/main/resources
- Loaded by ClassLoader
  - URL getSystemResource(String name)
  - InputStream getSystemResourceAsStream(String name)

Loading a Resource, Example

```
try ( InputStream inStream =
    ClassLoader.getSystemResourceAsStream( FILE_NAME );
)
{
    if ( inStream == null )
        throw new IOException( FILE_NAME + " not found" );
    InputStreamReader strReader   =
        new InputStreamReader( inStream );
    BufferedReader    reader      =
        new BufferedReader( strReader );

    String            line        = null;
    while ( (line = reader.readLine()) != null )
        System.out.println( line );
}
```