Files and Streams

Input and Output

Java provides streams as a general mechanism for dealing with data I/O.

Streams implement sequential access of data.

There are two kinds of streams: byte streams and character streams (aka binary streams and text streams, respectively).

Input and Output

- An input stream is an object that an application can use to read a sequence of data, and an output stream is an object that an application can use to write a sequence of data.
- An input stream acts as a source of data, and an output stream acts as a destination of data.

Input and Output

The following entities can act as both input and output streams:

- an array of bytes or characters
- a file
- a pipe (a mechanism by which a program can communicate data to another program during execution)
- a network connection

Input and Output

- Streams can be chained with filters to provide new functionality. In addition to dealing with bytes and characters, streams are provided for input and output of Java primitive values and objects.
- The **java.io** package also provides a general interface to interact with the file system of the host platform.

The File Class

- The File class provides a general machine-independent interface for the file system of the underlying platform.
- A File object represents the pathname of a file or directory in the host file system. An application can use the functionality provided by the File class for handling files and directories in the file system.
- The File class is not meant for handling the contents of files.

The File Class

- The pathname for a file or directory is specified using the naming conventions of the host system.
- However, the File class defines platform-dependent constants that can be used to handle file and directory names in a platform-independent way:

The File Class

public static final char separatorChar

public static final String separator

- Defines the character or string that separates the directory and the file components in a pathname. This separator is '/', '\' or ':' for Unix, Windows, and Macintosh, respectively.

public static final char pathSeparatorChar

public static final String pathSeparator

- Defines the character or string that separates the file or directory names in a “path list.” This character is ':' or ';' for Unix and Windows, respectively

The File Class

- Some examples of pathnames are:

`/book/chapter1` on Unix

`C:\book\chapter1` on Windows

`HD:book:chapter1` on Macintosh

- Some examples of path lists are:

`/book:/manual:/draft` on Unix

`C:\book;D:\manual;A:\draft` on Windows

The File Class

- The File class has various constructors for associating a file or a directory pathname to an object of the File class.
- Creating a File object does not mean creation of any file or directory based on the pathname specified.
- A File instance, called the abstract pathname, is a representation of the pathname of a file and directory.
- The pathname cannot be changed once the File object is created.

The File Class Constructors

- `File(String pathname)`

The pathname (of a file or a directory) can be an absolute pathname or a pathname relative to the current directory. An empty string as argument results in an abstract pathname for the current directory.

- `File(String directoryPathname, String fileName)`

This creates a File object whose pathname is as follows: directoryPathname + separator + fileName.

- `File(File directory, String fileName)`

If the directory argument is null, the resulting File object represents a file in the current directory. If the directory argument is not null, it creates a File object that represents a file in the given directory. The pathname of the file is then the pathname of the directory File object + separator + fileName.

The File Class

A File object can also be used to query the file system for information about a file or directory:

- whether the entry exists
- whether the File object represents a file or directory
- get and set read, write, or execute permissions for the entry
- get pathname information about the file or directory
- list all entries under a directory in the file system

Querying the File System

- `String getName()`

Returns the name of the file entry, excluding the specification of the directory in which it resides.

- `String getPath()`

The method returns the (absolute or relative) pathname of the file represented by the File object.

- `String getAbsolutePath()`

If the File object represents an absolute pathname, this pathname is returned, otherwise the returned pathname is constructed by concatenating the current directory pathname, the separator character and the pathname of the File object.

Querying the File System

- `String getCanonicalPath() throws IOException`

Also platform-dependent, the canonical path usually specifies an absolute pathname in which all relative references have been completely resolved.

- `String getParent()`

The parent part of the pathname of this File object is returned if one exists, otherwise the null value is returned. The parent part is generally the prefix obtained from the pathname after deleting the file or directory name component found after the last occurrence of the separator character. However, this is not true for all platforms.

- `boolean isAbsolute()`

Whether a File object represents an absolute pathname can be determined using this method.

Querying the File System

- `long lastModified()`

The modification time returned is encoded as a long value, and should only be compared with other values returned by this method.

- `long length()`

Returns the size (in bytes) of the file represented by the File object.

- `boolean equals(Object obj)`

This method just compares the pathnames of the File objects, and returns true if they are identical. On Unix systems, alphabetic case is significant in comparing pathnames; on Windows systems it is not

File or Directory Existence

A File object is created using a pathname. Whether this pathname denotes an entry that actually exists in the file system can be checked using the exists() method:

- `boolean exists()`

Since a File object can represent a file or a directory, the following methods can be used to distinguish whether a given File object represents a file or a directory, respectively:

- `boolean isFile()`
- `boolean isDirectory()`

File and Directory Permissions

To check whether the specified file has write, read, or execute permissions, the following methods can be used.

They throw a SecurityException if general access is not allowed, i.e., the application is not even allowed to check whether it can read, write or execute a file.

- `boolean canWrite()`
- `boolean canRead()`
- `boolean canExecute()`

File and Directory Permissions

Write, read and execute permissions can be set by calling the following methods.

If the first argument is true, the operation permission is set; otherwise it is cleared.

If the second argument is true, the permission only affects the owner; otherwise it affects all users.

- `boolean setReadable(boolean readable)`
- `boolean setReadable(boolean readable, boolean owner)`
- `boolean setWritable(boolean writable)`
- `boolean setWritable(boolean writable, boolean owner)`
- `boolean setExecutable(boolean executable)`
- `boolean setExecutable(boolean executable, boolean owner)`

These methods throw a SecurityException if permission cannot be changed. It should be noted that the exact interpretation of these permissions is platform dependent.

Listing Directory Entries

- `String[] list()`
- `String[] list(FilenameFilter filter)`
- `File[]   listFiles()`
- `File[]   listFiles(FilenameFilter filter)`
- `File[]   listFiles(FileFilter filter)`

Listing Directory Entries

A filter is an object of a class that implements either of these two interfaces:

```
interface FilenameFilter {
  boolean accept(File currentDirectory, String entryName);
}
interface FileFilter {
    boolean accept(File pathname);
}
```

Creating New Files and Directories

- `boolean createNewFile() throws IOException`

It creates a new, empty file named by the abstract pathname if, and only if, a file with this name does not already exist. The returned value is true if the file was successfully created, false if the file already exists. Any I/O error results in an IOException.

- `boolean mkdir()`
- `boolean mkdirs()`

The mkdirs() method creates any intervening parent directories in the pathname of the directory to be created

Renaming and Deleting Files and Directories

A file or a directory can be renamed, using the following method which takes the new pathname from its argument. It throws a SecurityException if access is denied.

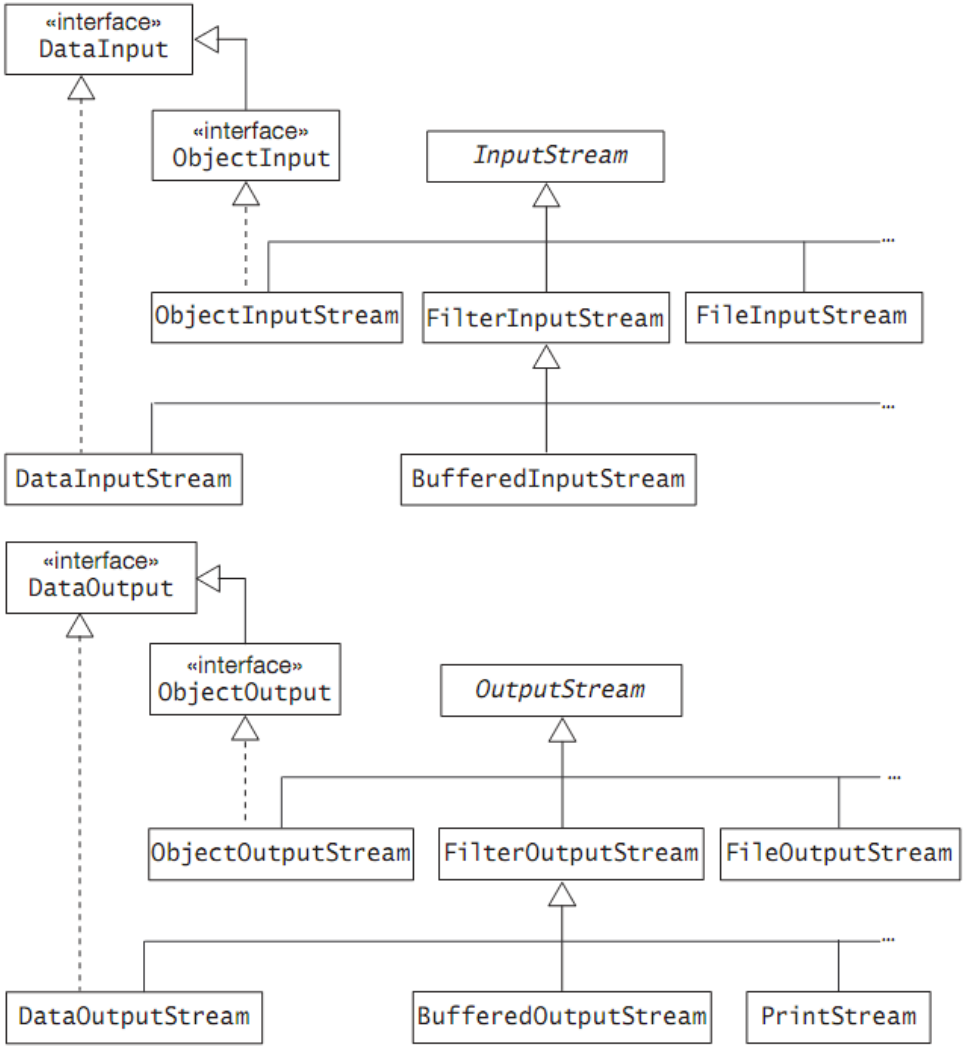
- `boolean renameTo(File dest)`

A file or a directory can be deleted using the following method. In the case of a directory, it must be empty before it can be deleted. It throws a SecurityException if access is denied.

- `boolean delete()`

Byte Streams: Input Streams and Output Streams

- The abstract classes InputStream and OutputStream are the root of the inheritance hierarchies for handling the reading and writing of bytes (Figure in the next slide).
- Their subclasses, implementing different kinds of input and output streams, override methods from the InputStream and OutputStream classes to customize the reading and writing of bytes, respectively


«interface»
DataInput
«interface»
ObjectInput
InputStream
ObjectInputStream
FilterInputStream
FileInputStream
DataInputStream
BufferedInputStream
«interface»
DataOutput
«interface»
ObjectOutput
OutputStream
ObjectOutputStream
FilterOutputStream
FileOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

Byte Streams:
Input Streams and Output Streams

The InputStream class:

- `int read() throws IOException`
- `int read(byte[] b) throws IOException`
- `int read(byte[] b, int off, int len) throws IOException`

Note that the first read() method reads a byte, but returns an int value.

The byte read resides in the eight least significant bits of the int value, while the remaining bits in the int value are zeroed out.

The read() methods return the value –1 when the end of the stream is reached.

Byte Streams: Input Streams and Output Streams

The OutputStream class:

- `void write(int b) throws IOException`
- `void write(byte[] b) throws IOException`
- `void write(byte[] b, int off, int len) throws IOException`

The first write() method takes an int as argument, but truncates it down to the eight least significant bits before writing it out as a byte

Byte Streams:
Input Streams and Output Streams

A stream should be closed when no longer needed, to free system resources.

```
void close() throws IOException
void flush() throws IOException //Only for OutputStream
```

Closing an output stream automatically flushes the stream, meaning that any data in its internal buffer is written out.
An output stream can also be manually flushed by calling the second method.

Selected Input Streams

`FileInputStream`	Data is read as bytes from a file. The file acting as the input stream can be specified by a `File` object, a `FileDescriptor` or a `String` file name.
`FilterInputStream`	Superclass of all input stream filters. An input filter must be chained to an underlying input stream.
`DataInputStream`	A filter that allows the binary representation of Java primitive values to be read from an underlying input stream. The underlying input stream must be specified.
`ObjectInputStream`	Allows binary representations of Java objects and Java primitive values to be read from a specified input stream.

Selected Output Streams

`FileOutputStream`	Data is written as bytes to a file. The file acting as the output stream can be specified by a `File` object, a `FileDescriptor` or a `String` file name.
`FilterOutputStream`	Superclass of all output stream filters. An output filter must be chained to an underlying output stream.
`DataOutputStream`	A filter that allows the binary representation of Java primitive values to be written to an underlying output stream. The underlying output stream must be specified.
`ObjectOutputStream`	Allows the binary representation of Java objects and Java primitive values to be written to a specified underlying output stream.

File Input Streams

The file can be specified by its name, through a File object, or using a FileDescriptor object.

- `FileInputStream(String name) throws FileNotFoundException`
- `FileInputStream(File file) throws FileNotFoundException`
- `FileInputStream(FileDescriptor fdObj)`

If the file does not exist, a FileNotFoundException is thrown.

If it exists, it is set to be read from the beginning.

A SecurityException is thrown if the file does not have read access.

File Output Streams

The file can be specified by its name, through a File object, or using a File Descriptor object.

- `FileOutputStream(String name) throws FileNotFoundException`
- `FileOutputStream(String name, boolean append) throws FileNotFoundException`
- `FileOutputStream(File file) throws IOException`
- `FileOutputStream(FileDescriptor fdObj)`

If the file does not exist, it is created.

If it exists, its contents are reset, unless the appropriate constructor is used to indicate that output should be appended to the file.

A SecurityException is thrown if the file does not have write access or it cannot be created.

Filter Streams

- A filter is a high-level stream that provides additional functionality to an underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter.
- The **FilterInputStream** and **FilterOutputStream** classes, together with their subclasses, define input and output filter streams.
- The subclasses **BufferedInputStream** and **BufferedOutputStream** implement filters that buffer input from and output to the underlying stream, respectively.
- The subclasses **DataInputStream** and **DataOutputStream** implement filters that allow binary representation of Java primitive values to be read and written, respectively, to and from an underlying stream.

Reading and Writing Binary Values

- The java.io package contains the two interfaces DataInput and DataOutput, that streams can implement to allow reading and writing of binary representations of Java primitive values (boolean, char, byte, short, int, long, float, double).
- The methods for writing binary representations of Java primitive values are named **writeX**, where X is any Java primitive data type.
- The methods for reading binary representations of Java primitive values are similarly named **readX**.

The DataInput and DataOutput Interfaces

Type	Methods in the *DataInput* Interface	Methods in the *DataOutput* interface
boolean	readBoolean()	writeBoolean(boolean b)
char	readChar()	writeChar(int c)
byte	readByte()	writeByte(int b)
short	readShort()	writeShort(int s)
int	readInt()	writeInt(int i)
long	readLong()	writeLong(long l)
float	readFloat()	writeFloat(float f)
double	readDouble()	writeDouble(double d)
String	readLine()	writeChars(String str)
String	readUTF()	writeUTF(String str)

Writing Binary Values to a File

1. Create a FileOutputStream:

```
FileOutputStream outputFile =
    new FileOutputStream("primitives.data");
```

2. Create a DataOutputStream which is chained to the FileOutputStream:

```
DataOutputStream outputStream =
            new DataOutputStream(outputFile);
```

Writing Binary Values to a File

3. Write Java primitive values using relevant writeX() methods:

```
outputStream.writeBoolean(true);
outputStream.writeChar('A');   // int written as Unicode char
outputStream.writeByte(Byte.MAX_VALUE);   // int written as 8-bits byte
outputStream.writeShort(Short.MIN_VALUE);// int written as 16-bits short
outputStream.writeInt(Integer.MAX_VALUE);
outputStream.writeLong(Long.MIN_VALUE);
outputStream.writeFloat(Float.MAX_VALUE);
outputStream.writeDouble(Math.PI);
```

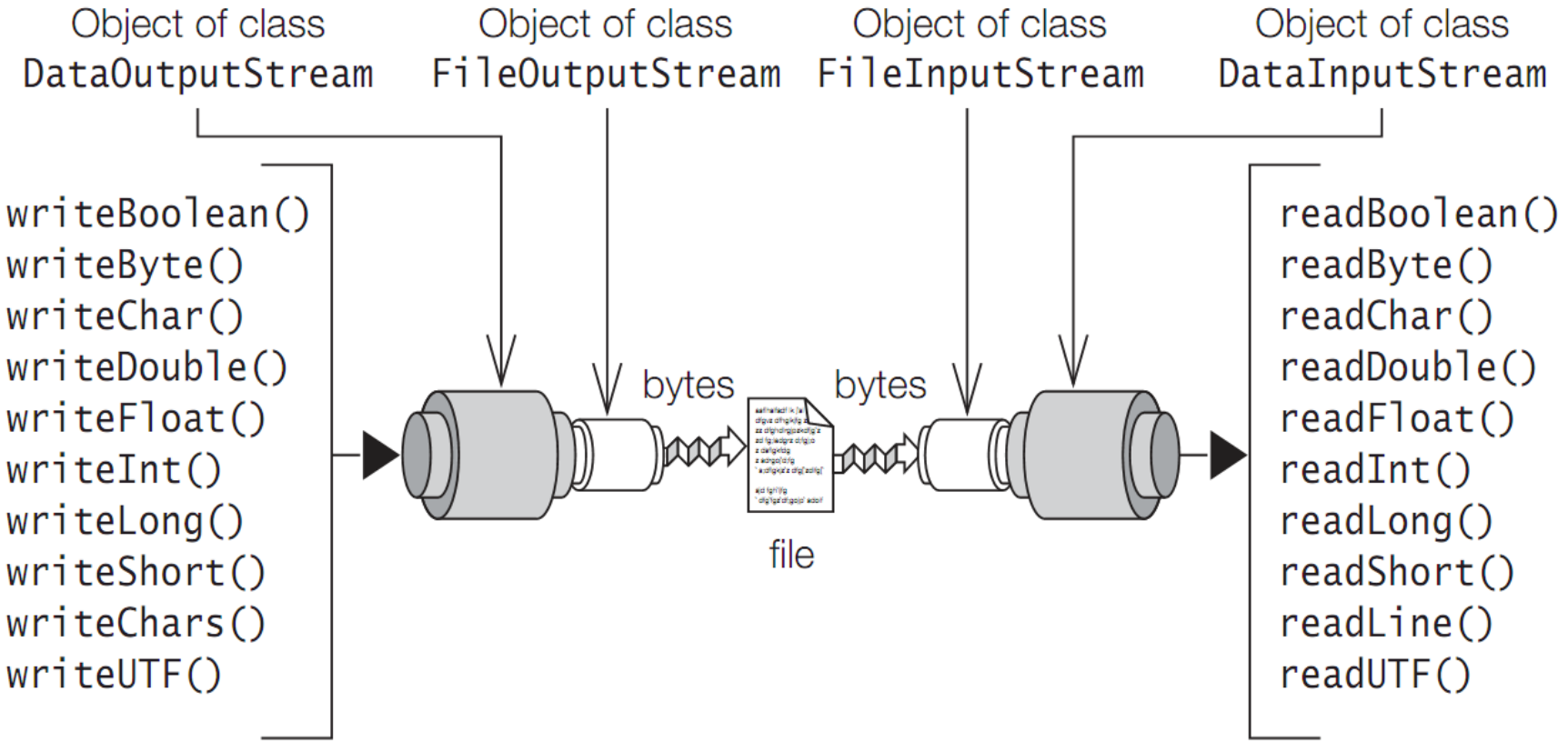
Note that in the case of char, byte, and short data types, the int argument to the writeX() method is converted to the corresponding type, before it is written

Writing Binary Values to a File

4. Close the filter stream, which also closes the underlying stream:

```
outputStream.close();
```

Stream Chaining for Reading and Writing Binary Values to a File



Reading Binary Values From a File

1. Create a FileInputStream:

```
FileInputStream inputFile =
     new FileInputStream("primitives.data");
```

2. Create a DataInputStream which is chained to the FileInputStream:

```
DataInputStream inputStream =
               new DataInputStream(inputFile);
```

Reading Binary Values From a File

3. Read the (exact number of) Java primitive values in the same order they were written out, using relevant readX() methods:

```
boolean v = inputStream.readBoolean();
char    c = inputStream.readChar();
byte    b = inputStream.readByte();
short   s = inputStream.readShort();
int     i = inputStream.readInt();
long    l = inputStream.readLong();
float   f = inputStream.readFloat();
double  d = inputStream.readDouble();
```

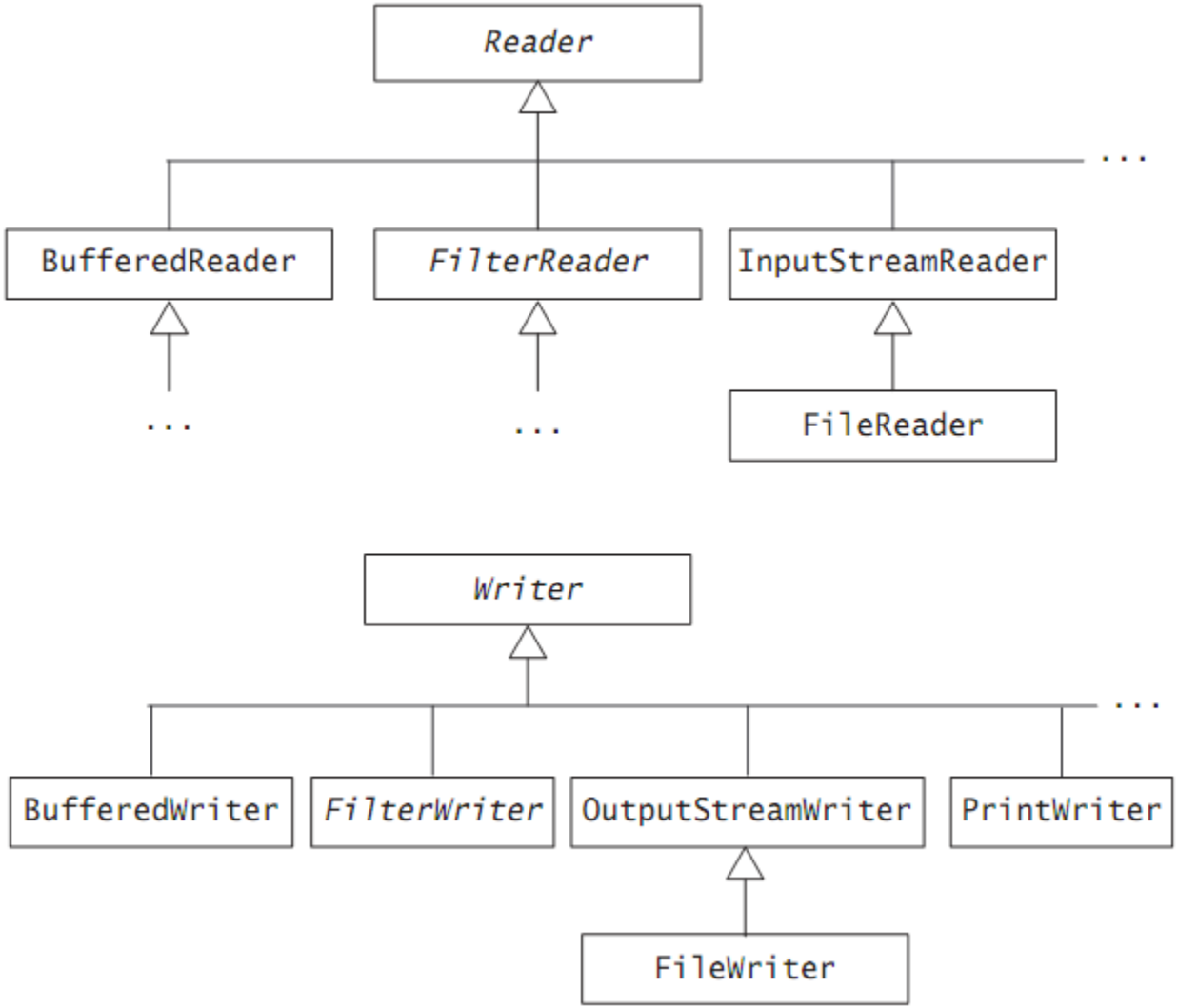
Reading Binary Values From a File

4. Close the filter stream, which also closes the underlying stream:

```
inputStream.close();
```

Character Streams: Readers and Writers

- The abstract classes Reader and Writer are the roots of the inheritance hierarchies for streams that read and write Unicode characters using a specific character encoding (as shown in next slide).
- A reader is an input character stream that reads a sequence of Unicode characters, and a writer is an output character stream that writes a sequence of Unicode characters.
- Character encodings are used by readers and writers to convert between external encoding and internal Unicode characters.


Reader
...
BufferedReader
FilterReader
InputStreamReader
...
...
FileReader
Writer
...
BufferedWriter
FilterWriter
OutputStreamWriter
PrintWriter
FileWriter

Selected Readers

`BufferedReader`	A reader that buffers the characters read from an underlying reader. The underlying reader must be specified and an optional buffer size can be given.
`InputStreamReader`	Characters are read from a byte input stream which must be specified. The default character encoding is used if no character encoding is explicitly specified.
`FileReader`	Reads characters from a file, using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileInputStream` that is associated with the file.

Readers

- Readers use the following methods for reading Unicode characters:

```
int read() throws IOException
int read(char cbuf[]) throws IOException
int read(char cbuf[], int off, int len) throws
IOException
```

- Note that the read() methods read the character as an int in the range 0 to 65535 (0x0000–0xFFFF).
- The value –1 is returned if the end of the stream has been reached.

```
long skip(long n) throws IOException
```

- A reader can skip over characters using the skip() method.

Selected Writers

`BufferedWriter`	A writer that buffers the characters before writing them to an underlying writer. The underlying writer must be specified, and an optional buffer size can be specified.
`OutputStreamWriter`	Characters are written to a byte output stream which must be specified. The default character encoding is used if no explicit character encoding is specified.
`FileWriter`	Writes characters to a file, using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileOutputStream` that is associated with the file.
`PrintWriter`	A filter that allows *text* representation of Java objects and Java primitive values to be written to an underlying output stream or writer. The underlying output stream or writer must be specified.

Writers

- Writers use the following methods for writing Unicode characters:

```
void write(int c) throws IOException
```

The write() method takes an int as argument, but writes only the least significant 16 bits.

```
void write(char[] cbuf) throws IOException
void write(String str) throws IOException
void write(char[] cbuf, int off, int length) throws IOException
void write(String str, int off, int length) throws IOException
```

- These methods write the characters from an array of characters or a string.

```
void close() throws IOException
void flush() throws IOException
```

Like byte streams, a character stream should be closed when no longer needed to free system resources. Closing a character output stream automatically flushes the stream. A character output stream can also be manually flushed.

Print Writers

- The capabilities of the **OutputStreamWriter** and the **InputStreamReader** classes are limited, as they primarily write and read characters.
- In order to write a text representation of Java primitive values and objects, a **PrintWriter** should be chained to either a **writer**, a **byte output stream**, **File**, or a **String file name**, using one of the following constructors:

Print Writers

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(File file)
PrintWriter(File file, String charsetName)
PrintWriter(String fileName)
PrintWriter(String fileName, String charsetName)
```

The autoFlush argument specifies whether the PrintWriter should be flushed when any println() method of the PrintWriter class is called.

Print Writers

- When the underlying writer is specified, the character encoding supplied by the underlying writer is used.
- However, an OutputStream has no notion of any character encoding, so the necessary intermediate OutputStreamWriter is automatically created, which will convert characters into bytes, using the default character encoding.
- When supplying the File object or the file name, the character encoding can be specified explicitly.

Print Methods of the PrintWriter Class

print()-methods	*println*-methods
	println()
print(boolean b)	println(boolean b)
print(char c)	println(char c)
print(int i)	println(int i)
print(long l)	println(long l)
print(float f)	println(float f)
print(double d)	println(double d)
print(char[] s)	println(char[] ca)
print(String s)	println(String str)
print(Object obj)	println(Object obj)

Print Methods of the PrintWriter Class

- The println() methods write the text representation of their argument to the underlying stream, and then append a line-separator. The println() methods use the correct platform-dependent line-separator.
- *For example, on Unix platforms the line-separator is '\n' (newline), while on Windows platforms it is "\r\n" (carriage return + newline) and on the Macintosh it is '\r' (carriage return).*
- The print() methods create a text representation of an object by calling the toString() method on the object.
- The print() methods do not throw any IOException.
- Instead, the checkError() method of the PrintWriter class must be called to check for errors.
- In addition, the PrintWriter class provides the format() method and the convenient printf() method to write formatted values

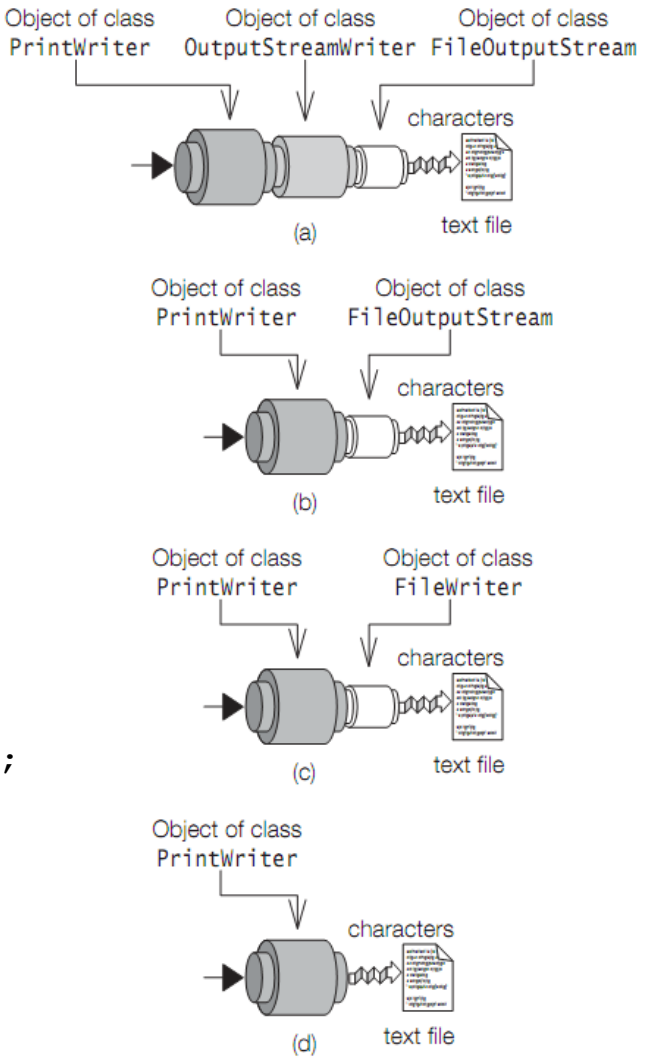
Writing Text Files

a)
```
FileOutputStream outputFile =
        new FileOutputStream("info.txt");
OutputStreamWriter outputStream =
        new OutputStreamWriter(outputFile);
PrintWriter printWriter1 =
        new PrintWriter(outputStream, true);
```

b)
```
FileOutputStream outputFile =
        new FileOutputStream("info.txt");
PrintWriter printWriter2 =
        new PrintWriter(outputFile, true);
```

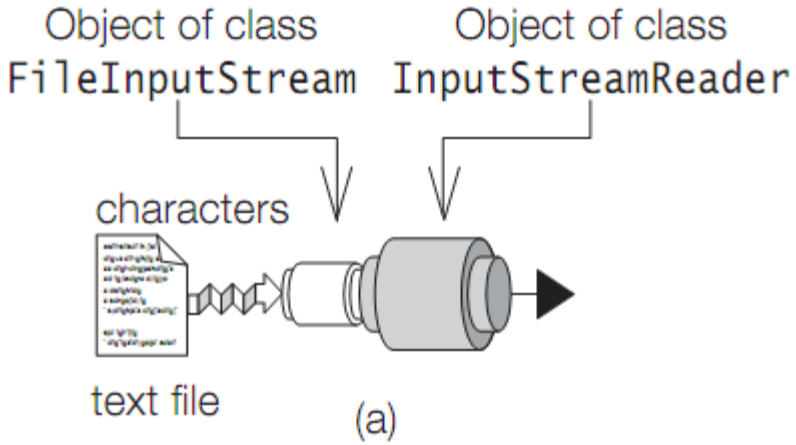
c)
```
FileWriter fileWriter = new FileWriter("info.txt");
PrintWriter printWriter3 =
        new PrintWriter(fileWriter, true);
```

d)
```
PrintWriter printWriter3 =
        new PrintWriter("info.txt");
```

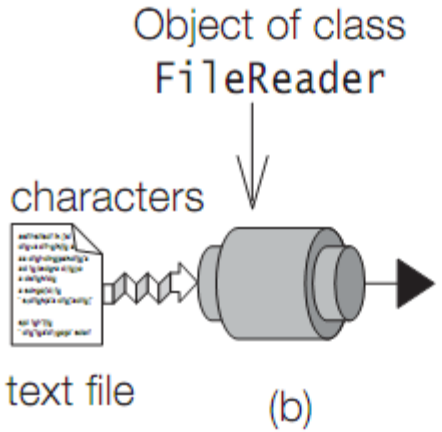


Reading Text Files

```
FileInputStream inputFile =
   new FileInputStream("info.txt");
InputStreamReader reader =
   new InputStreamReader(inputFile);
```




```
FileReader fileReader =
   new FileReader("info.txt");
```



Using Buffered Writers

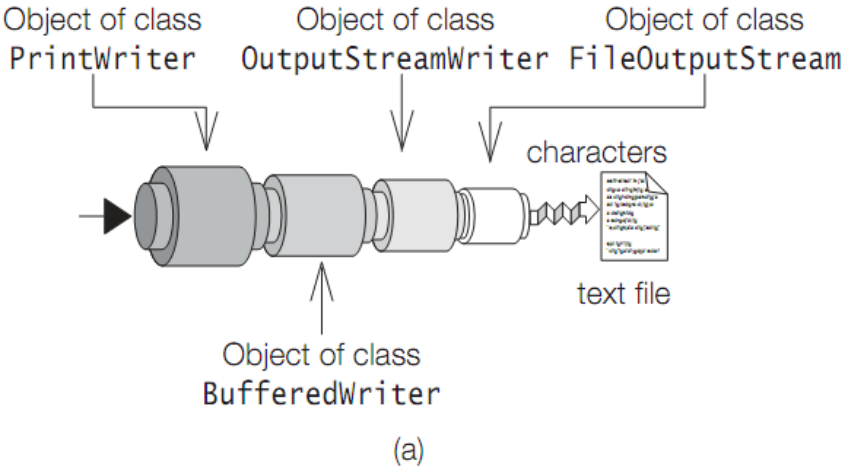
The following code creates a PrintWriter whose output is buffered and the characters are written using the 8859_1 character encoding (a):

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
OutputStreamWriter outputStream =
        new OutputStreamWriter(outputFile, "8859_1");
BufferedWriter bufferedWriter1 = new BufferedWriter(outputStream);
PrintWriter printWriter1    = new PrintWriter(bufferedWriter1, true);
```

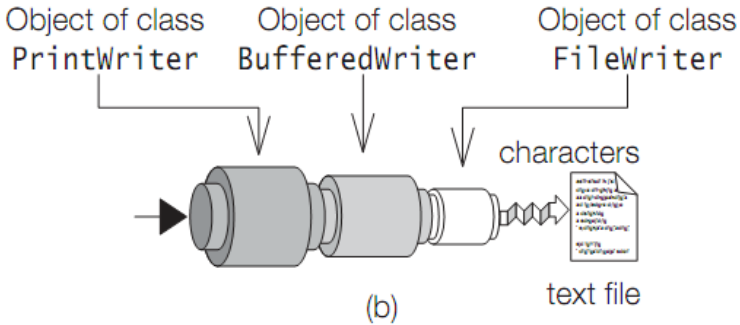
The following code creates a PrintWriter whose output is buffered, and the characters are written using the default character encoding (b):

```
FileWriter   fileWriter    = new FileWriter("info.txt");
BufferedWriter bufferedWriter2 = new BufferedWriter(fileWriter);
PrintWriter  printWriter2    = new PrintWriter(bufferedWriter2, true);
```

Using Buffered Writers



(a)



(b)

Using Buffered Readers

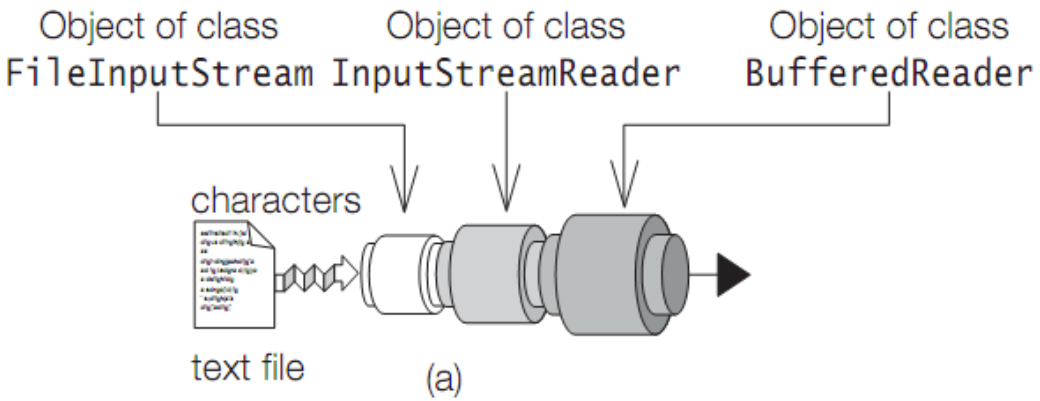
The following code creates a BufferedReader that can be used to read text lines from a file, using the 8859_1 character encoding (a):

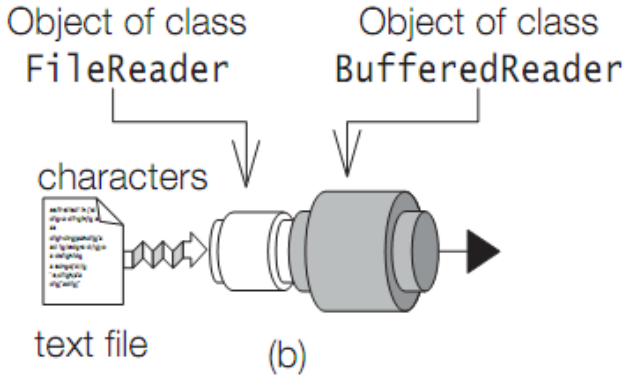
```
FileInputStream inputFile =
        new FileInputStream("info.txt");
InputStreamReader reader =
        new InputStreamReader(inputFile, "8859_1");
BufferedReader bufferedReader1 =
        new BufferedReader(reader);
```

The following code creates a BufferedReader that can be used to read text lines from a file, using the default character encoding (b):

```
FileReader fileReader = new FileReader("lines.txt");
BufferedReader bufferedReader2 =
        new BufferedReader(fileReader);
```

Using Buffered Readers





The Standard Input, Output, and Error Streams

- The standard output stream (usually the display) is represented by the PrintStream object System.out.
- The standard input stream (usually the keyboard) is represented by the InputStream object System.in. In other words, it is a byte input stream.
- The standard error stream (also usually the display) is represented by System.err which is another object of the PrintStream class. The PrintStream class offers print() methods which act as corresponding print() methods from the PrintWriter class.
- These methods can be used to write output to System.out and System.err. In other words, both System.out and System.err act like PrintWriter, but in addition they have write() methods for writing bytes.

Comparison of Byte Streams and Character Streams

Byte Streams	Character Streams
`OutputStream`	`Writer`
`InputStream`	`Reader`
No counterpart	`OutputStreamWriter`
No counterpart	`InputStreamReader`
`FileOutputStream`	`FileWriter`
`FileInputStream`	`FileReader`
`BufferedOutputStream`	`BufferedWriter`
`BufferedInputStream`	`BufferedReader`
`PrintStream`	`PrintWriter`
`DataOutputStream`	*No counterpart*
`DataInputStream`	*No counterpart*
`ObjectOutputStream`	*No counterpart*
`ObjectInputStream`	*No counterpart*

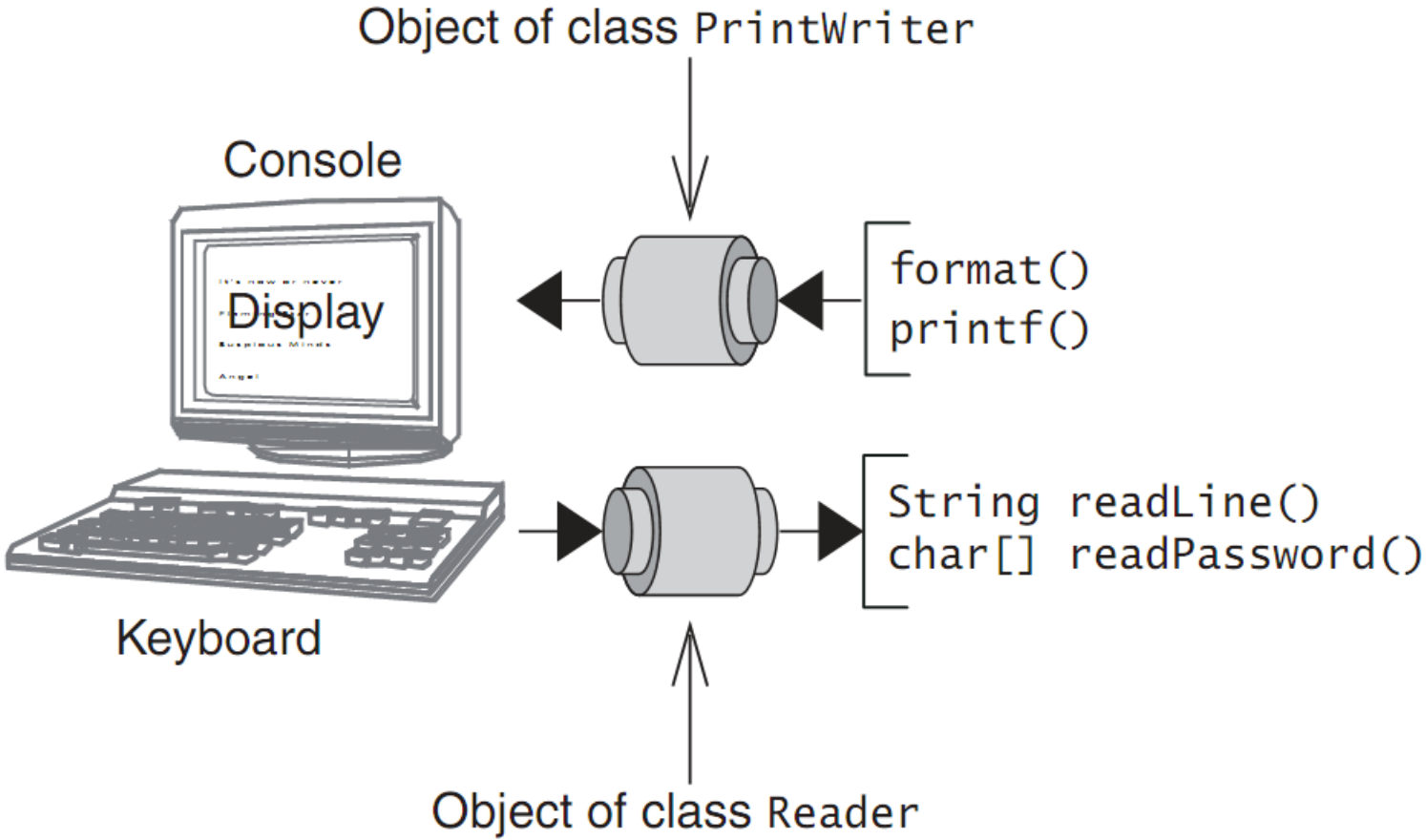
The Console Class

- A console is a unique character-based device associated with a JVM. Whether a JVM has a console depends on the platform, and also on the manner in which the JVM is invoked.
- When the JVM is started from a command line, and the standard input and output streams have not been redirected, the console will normally correspond to the keyboard and the display.
- In any case, the console will be represented by an instance of the class Console. This Console instance is obtained by calling the static method console() of the System class.
- If there is no console associated with the JVM, the null value is returned by this method.

The Console Class

```
// Obtaining the console:
Console console = System.console();
if (console == null) {
  System.err.println("No console available.");
  return;
}
// Continue ...
```

The Console Class



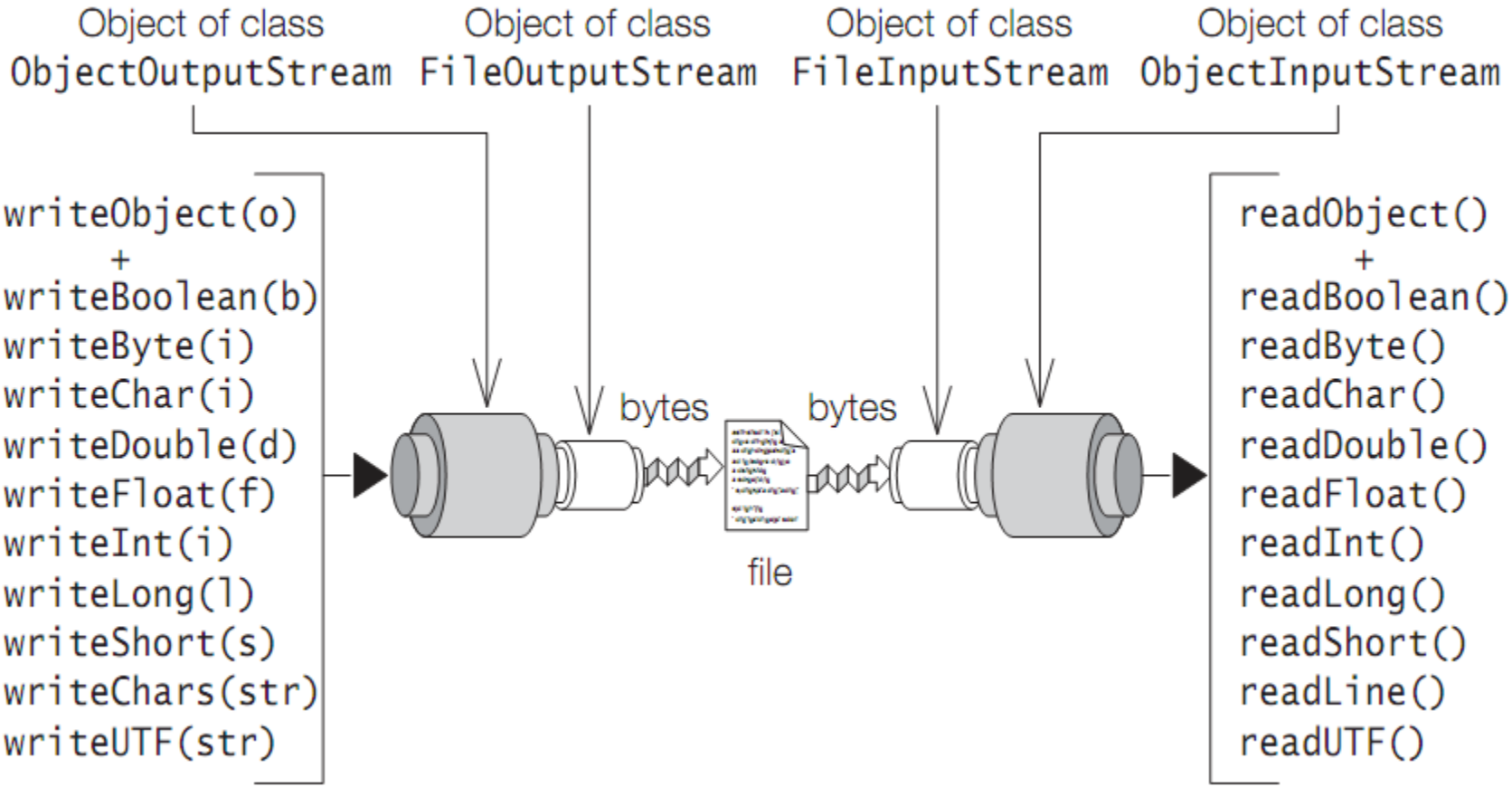
Object Serialization

- Object serialization allows an object to be transformed into a sequence of bytes that can later be re-created (deserialized) into the original object.
- After deserialization, the object has the same state as it had when it was serialized, barring any data members that were not serializable. This mechanism is generally known as persistence.
- Java provides this facility through the ObjectInput and ObjectOutput interfaces, which allow the reading and writing of objects from and to streams.
- These two interfaces extend the DataInput and DataOutput interfaces, respectively

Object Serialization

- The ObjectOutputStream class and the ObjectInputStream class implement the ObjectOutput interface and the ObjectInput interface, respectively, providing methods to write and read binary representation of objects as well as Java primitive values.
- Figure gives an overview of how these classes can be chained to underlying streams and some selected methods they provide.
- The figure does not show the methods inherited from the abstract OutputStream and InputStream superclasses.

Object Serialization



The ObjectOutputStream Class

- In order to store objects in a file and thus provide persistent storage for objects, an ObjectOutputStream can be chained to a FileOutputStream:

```
FileOutputStream outputFile =
        new FileOutputStream("obj-storage.dat");
ObjectOutputStream outputStream =
        new ObjectOutputStream(outputFile);
```

- Objects can be written to the stream using the writeObject() method of the ObjectOutputStream class:

```
final void writeObject(Object obj) throws IOException
```

The ObjectOutputStream Class

- The writeObject() method can be used to write any object to a stream, including strings and arrays, as long as the object implements the java.io.Serializable interface, which is a marker interface with no methods.
- The String class, the primitive wrapper classes and all array types implement the Serializable interface.
- A serializable object can be any compound object containing references to other objects, and all constituent objects that are serializable are serialized recursively when the compound object is written out.
- This is true even if there are cyclic references between the objects.
- Each object is written out only once during serialization.

The ObjectOutputStream Class

- The following information is included when an object is serialized:
  - the class information needed to reconstruct the object.
  - the values of all serializable non-transient and non-static members, including those that are inherited.

- An exception of the type java.io.NotSerializableException is thrown if a non-serializable object is encountered during the serialization process.
- Note also that objects of subclasses that extend a serializable class are always serializable.

The ObjectInputStream Class

- An ObjectInputStream is used to restore (deserialize) objects that have previously been serialized using an ObjectOutputStream.
- An ObjectInputStream must be chained to an InputStream, using the following constructor:

```
ObjectInputStream(InputStream in)
    throws IOException, StreamCorruptedException
```

The ObjectInputStream Class

- In order to restore objects from a file, an ObjectInputStream can be chained to a FileInputStream:

```
FileInputStream inputFile =
        new FileInputStream("obj-storage.dat");
ObjectInputStream inputStream =
        new ObjectInputStream(inputFile);
```

- The method readObject() of the ObjectInputStream class is used to read an object from the stream:

```
final Object readObject()
        throws OptionalDataException,
               ClassNotFoundException, IOException
```

Customizing Object Serialization

- The class of the object must implement the Serializable interface if we want the object to be serialized. If this object is a compound object, then all its constituent objects must also be serializable, and so on.
- It is not always possible for a client to declare that a class is Serializable. It might be declared final, and therefore not extendable. The client might not have access to the code, or extending this class with a serializable subclass might not be an option.
- Java provides a customizable solution for serializing objects in such cases.

Customizing Object Serialization

- The basic idea behind the scheme is to use default serialization as much as possible, and provide “hooks” in the code for the serialization mechanism to call specific methods to deal with objects or values that should not or cannot be serialized by the default methods of the object streams.
- Any serializable object has the option of customizing its own serialization if it implements the following pair of methods:

```
private void writeObject(ObjectOutputStream)
         throws IOException;
private void readObject(ObjectInputStream)
         throws IOException, ClassNotFoundException;
```

Serialization and Inheritance

- The inheritance hierarchy of an object also determines what its state will be after it is deserialized.
- An object will have the same state at deserialization as it had at the time it was serialized if all its superclasses are also serializable.
- This is because the normal object creation procedure using constructors is not run during deserialization

Serialization and Inheritance

- However, if any superclass of an object is not serializable, then the normal creation procedure using constructors is run, starting at the first non-serializable superclass, all the way up to the Object class.
- This means that the state at deserialization might not be the same as at the time the object was serialized, because superconstructors run during deserialization may have initialized the object's state.

That’s all