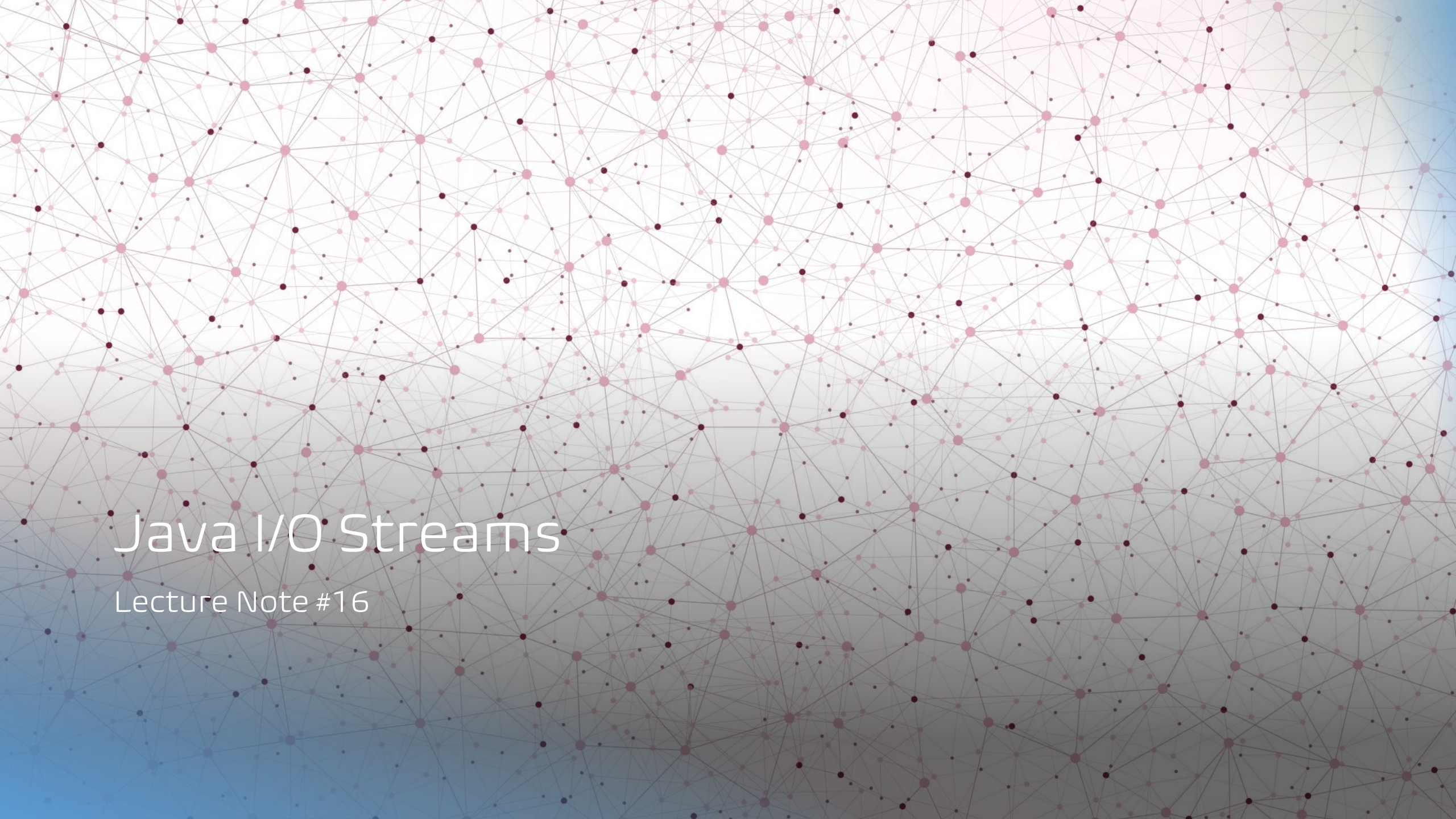
CS 213 SOFTWARE METHODOLOGY

Lily Chang

CS Department @ Rutgers New Brunswick

FALL 2023



Java I/O Streams

Lecture Note #16

Data in main memory vs. Persistent data

- Data you process with your programs so far are temporary data in main memory
  - You will not have the reference to those data after your program terminates
- To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device.
  - The data in files can be referenced at a later time when you run your programs again, or they can be referenced by other programs

File Names

Absolute file name

- Every file is placed in a directory in the file system.
- An absolute file name (or full name) contains a file name with its complete path and drive letter.
- For example, c:\book\welcome.java is the absolute file name for the file **welcome.java** on the Windows operating system; for UNIX based operating system, it could be /home/username/Documents/welcome.java, where **/home/username/Documents** referred to as the directory name

Relative file name

- Is in relation to the current working directory; that is, the complete directory path for a relative file name is omitted.

The File class in java

- The **File class** is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
- The filename is a string
- The File class is a wrapper class for the file name and its directory path.
- The File class contains the methods for obtaining file and directory properties, and for renaming and deleting files and directories, however, the File class does not contain the methods for reading and writing file contents.

Field Summary

Fields

Modifier and Type	Field	Description
static String	pathSeparator	The system-dependent path-separator character, represented as a string for convenience.
static char	pathSeparatorChar	The system-dependent path-separator character.
static String	separator	The system-dependent default name-separator character, represented as a string for conveni
static char	separatorChar	The system-dependent default name-separator character.

Constructor Summary

Constructors

Constructor	Description
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 pathna
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.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
boolean	canExecute()	Tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	Tests whether the application can read the file denoted by this abstract pathname.
boolean	canWrite()	Tests whether the application can modify the file denoted by this abstract pathname.
int	compareTo(File pathname)	Compares two abstract pathnames lexicographically.
boolean	createNewFile()	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static File	createTempFile(String prefix, String suffix)	Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static File	createTempFile(String prefix, String suffix, File directory)	Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean	delete()	Deletes the file or directory denoted by this abstract pathname.
void	deleteOnExit()	Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean	equals(Object obj)	Tests this abstract pathname for equality with the given object.
boolean	exists()	Tests whether the file or directory denoted by this abstract pathname exists.
File	getAbsoluteFile()	Returns the absolute form of this abstract pathname.
String	getAbsolutePath()	Returns the absolute pathname string of this abstract pathname.
File	getCanonicalFile()	Returns the canonical form of this abstract pathname.
String	getCanonicalPath()	Returns the canonical pathname string of this abstract pathname.
long	getFreeSpace()	Returns the number of unallocated bytes in the partition named by this abstract path name.
String	getName()	Returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File	getParentFile()	Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
String	getPath()	Converts this abstract pathname into a pathname string.

The File class in java

- The directory separator for Windows is a backslash (\); the backslash is a special - character in Java and should be written as \\ in a string literal
    - Do not use absolute file names in your program. If you use a file name such as c:\\book\\Welcome.java, it will work on Windows but not on other platforms. You should use a file name relative to the current directory.
- The forward slash (/) is the Java directory separator, which is the same as on UNIX
    - For example, the statement **new File("image/us.gif")** works on Windows, UNIX, and any other platform.
- Constructing a File instance does not create a file on the machine.
    - You can create a File instance for any file name regardless of whether it exists or not.
    - You can invoke the **exists()** method on a File instance to check whether the file exists; so the exception at runtime can be properly handled.

The File class – path name

By default the classes in the java.io package always resolve relative pathnames against the current user directory. This directory is named by the system property user.dir and is typically the directory in which the Java virtual machine was invoked.

The parent of an abstract pathname may be obtained by invoking the getParent() method of this instance

Some Examples

```
public class TestFileClass {
    public static void main(String[] args) {
        java.io.File file = new java.io.File("image/us.gif");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path is " + file.getAbsolutePath());
        System.out.println("Last modified on " +
                                    new java.util.Date(file.lastModified()));
    }
}
```

Java Input and Output (I/O) classes

A File object encapsulates the properties of a file or a path, but it does not contain the methods for writing/reading data to/from a file (referred to as data input and output, or I/O for short).

In order to perform I/O, you need to create objects using appropriate Java I/O classes

There are two types of files: **text** and **binary**

Text I/O – Scanner class (read)

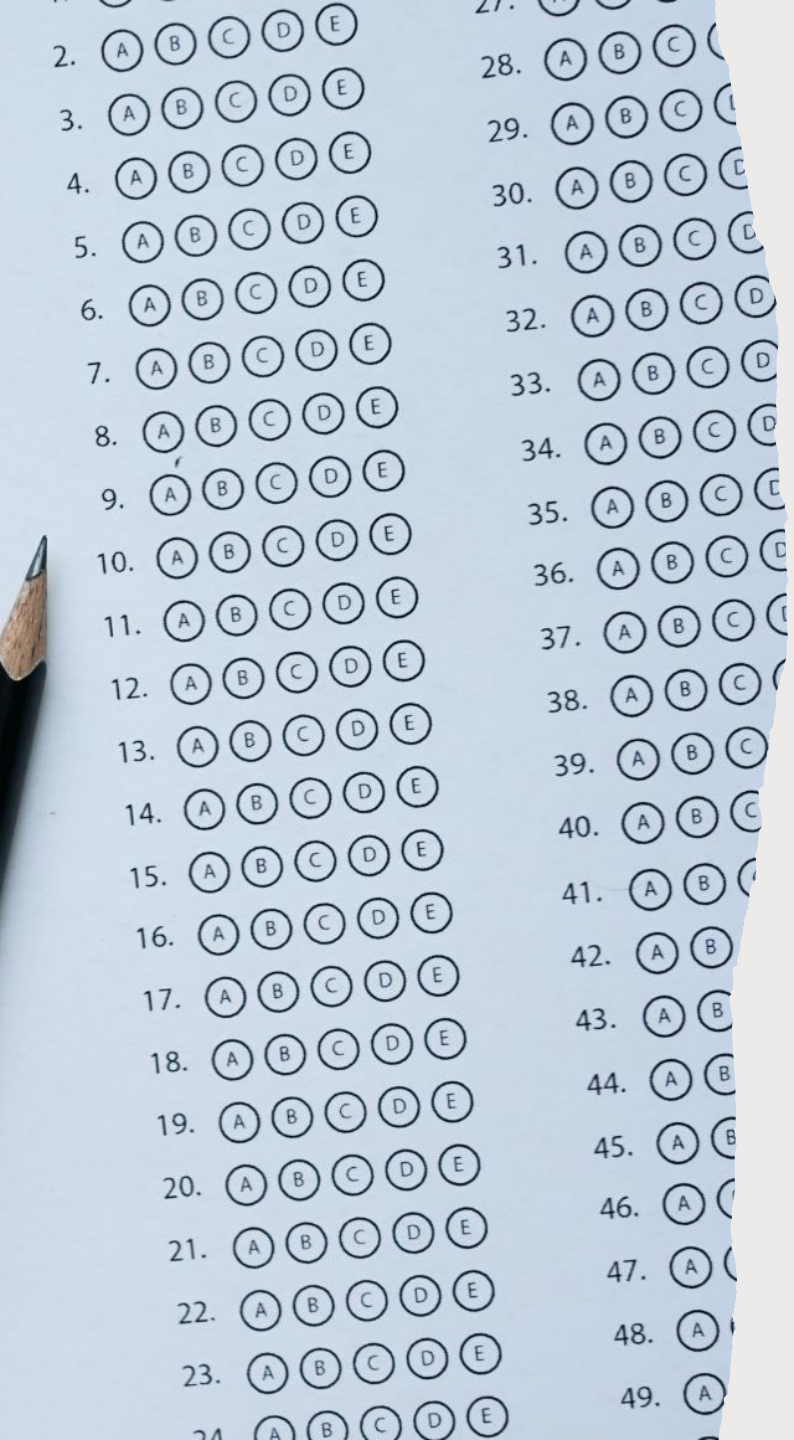
A simple text scanner which can **parse primitive types** and **strings** using regular expressions.

A **Scanner** breaks its input into tokens using a delimiter pattern, which by default matches whitespace, i.e., space, tab (\t) or new line(\n)

The resulting tokens may then be converted into values of different types using the various next methods.

For example, the following code allows a user to read a number from **System.in**

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```



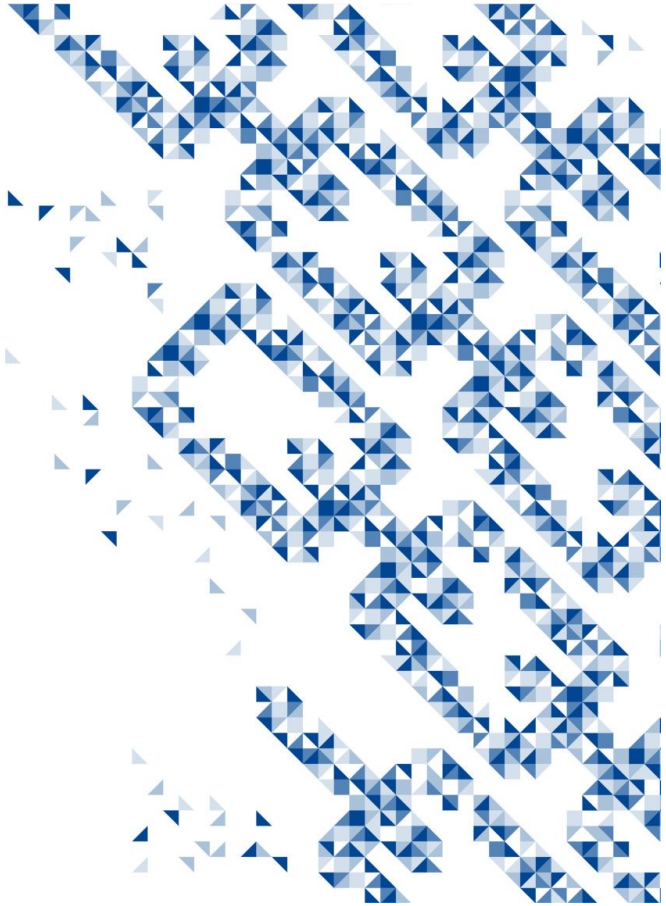
Text I/O – Scanner class

- A scanning operation may block waiting for input.
- The next() and hasNext() methods and their companion methods (such as nextInt() and hasNextInt()) first skip any input that matches the delimiter pattern, and then attempt to return the next token.
- When a scanner throws an InputMismatchException, the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method
- A Scanner is not safe for multithreaded use without external synchronization.

Constructors

Constructor	Description
`Scanner(File source)`	Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)`	Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, Charset charset)`	Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(InputStream source)`	Constructs a new Scanner that produces values scanned from the specified input
`Scanner(InputStream source, String charsetName)`	Constructs a new Scanner that produces values scanned from the specified input
`Scanner(InputStream source, Charset charset)`	Constructs a new Scanner that produces values scanned from the specified input
`Scanner(Readable source)`	Constructs a new Scanner that produces values scanned from the specified source
`Scanner(String source)`	Constructs a new Scanner that produces values scanned from the specified string
`Scanner(ReadableByteChannel source)`	Constructs a new Scanner that produces values scanned from the specified chann
`Scanner(ReadableByteChannel source, String charsetName)`	Constructs a new Scanner that produces values scanned from the specified chann
`Scanner(ReadableByteChannel source, Charset charset)`	Constructs a new Scanner that produces values scanned from the specified chann
`Scanner(Path source)`	Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)`	Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, Charset charset)`	Constructs a new Scanner that produces values scanned from the specified file.

Text I/O – PrintWriter class (write)



- Prints formatted representations of objects to a text-output stream.
- This class implements all the print methods found in PrintStream class.
- It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.
- Unlike the PrintStream class, if automatic flushing is enabled, it will be done only when one of the println, printf, or format methods is invoked, rather than whenever a newline character happens to be output.
- Methods in this class never throw I/O exceptions, although some of its constructors may; the client may inquire as to whether any errors have occurred by invoking checkError().

Constructor Summary

Constructors

Constructor	Description
`PrintWriter(File file)`	Creates a new PrintWriter, without automatic line flushing, with the specified file.
`PrintWriter(File file, String csn)`	Creates a new PrintWriter, without automatic line flushing, with the specified file and charset.
`PrintWriter(File file, Charset charset)`	Creates a new PrintWriter, without automatic line flushing, with the specified file and charset.
`PrintWriter(OutputStream out)`	Creates a new PrintWriter, without automatic line flushing, from an existing OutputStream.
`PrintWriter(OutputStream out, boolean autoFlush)`	Creates a new PrintWriter from an existing OutputStream.
`PrintWriter(OutputStream out, boolean autoFlush, Charset charset)`	Creates a new PrintWriter from an existing OutputStream.
`PrintWriter(Writer out)`	Creates a new PrintWriter, without automatic line flushing.
`PrintWriter(Writer out, boolean autoFlush)`	Creates a new PrintWriter.
`PrintWriter(String fileName)`	Creates a new PrintWriter, without automatic line flushing, with the specified file name.
`PrintWriter(String fileName, String csn)`	Creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.
`PrintWriter(String fileName, Charset charset)`	Creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
PrintWriter	append(char c)	Appends the specified character to this writer.
PrintWriter	append(CharSequence csq)	Appends the specified character sequence to this writer.
PrintWriter	append(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer.
boolean	checkError()	Flushes the stream if it's not closed and checks its error state.
protected void	clearError()	Clears the error state of this stream.
void	close()	Closes the stream and releases any system resources associated with it.
void	flush()	Flushes the stream.
PrintWriter	format(String format, Object... args)	Writes a formatted string to this writer using the specified format string and arguments.
PrintWriter	format(Locale l, String format, Object... args)	Writes a formatted string to this writer using the specified format string and arguments.
void	print(boolean b)	Prints a boolean value.
void	print(char c)	Prints a character.
void	print(char[] s)	Prints an array of characters.
void	print(double d)	Prints a double-precision floating-point number.
void	print(float f)	Prints a floating-point number.
void	print(int i)	Prints an integer.
void	print(long l)	Prints a long integer.

Example – PrintWriter

```
File file = new File("scores.txt");
if (file.exist()) {
    System.out.println("File already exists.");
    System.exit(1);
}
PrintWriter pw = new PrintWriter(file);
pw.print("John Smith "); //write to file scores.txt
pw.println("90");
pw.print("Eric Johnson ");
pw.println("85");
pw.close();
```

create a new file if the file does not exist.

If the file already exists, the current content in the file will be overwritten without verifying with the user.

Binary I/O

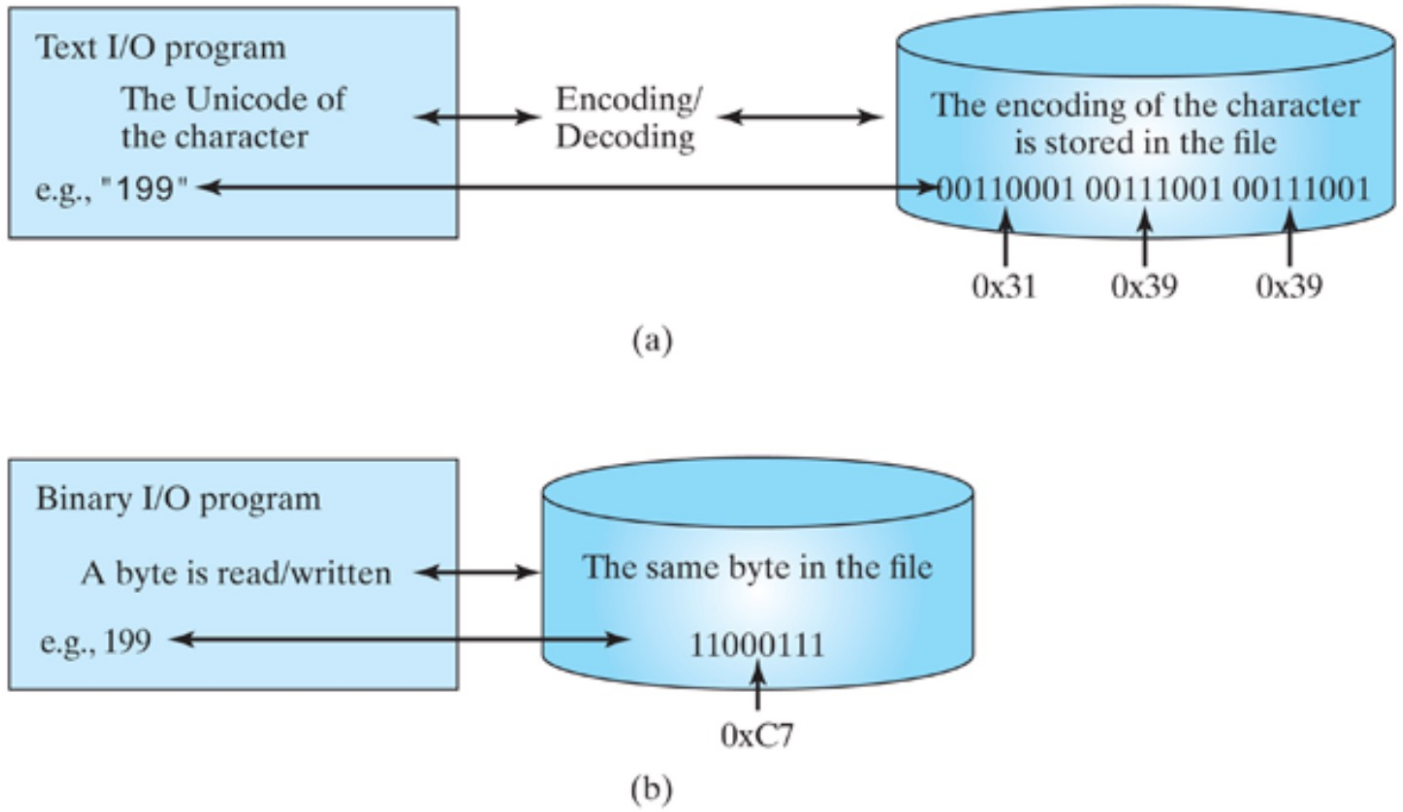
- Files can be classified as either text or binary.
- A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a text file.
- All other files are called binary files.
  - You cannot read binary files using a text editor
  - Binary files are application specific; they are designed to be read by application programs
  - For example, a Microsoft Word document can be processed by MS Word application; Java .class files are processed by the Java JVM

Java I/O classes

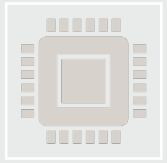
- There are many I/O classes for various purposes. In general, these can be classified as input classes and output classes.
- An input class contains the methods to read data, and an output class contains the methods to write data.
- PrintWriter is an example of an output class, and Scanner is an example of an input class.
- Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially binary files.
- Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding,

Java I/O classes

- Encoding and decoding are automatically performed for text I/O. The JVM converts Unicode to a file-specific encoding when writing a character and converts a file-specific encoding to Unicode when reading a character.
- Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file.



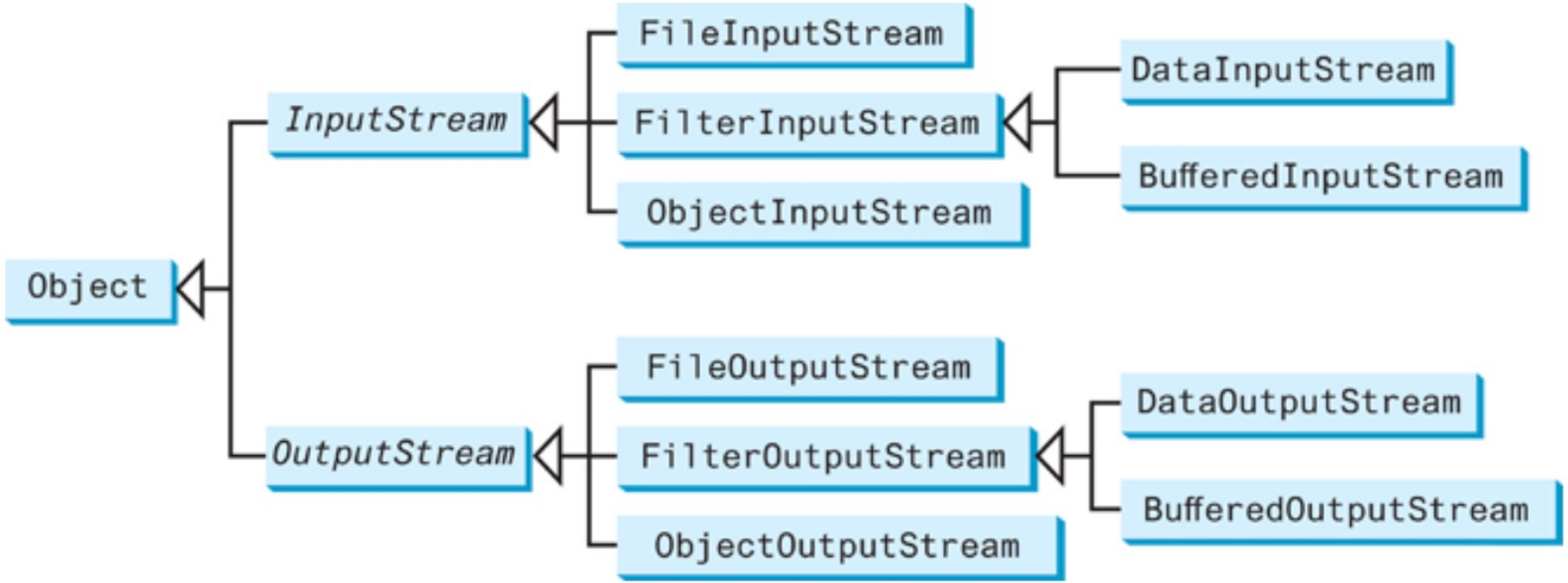
Java I/O classes



In general, you should use text input to read a file created by a text editor or a text output program and use binary input to read a file created by a Java binary output program.

Binary I/O is more efficient than text I/O because binary I/O does not require encoding and decoding.

JAVA BINARY I/O CLASSES



- java.lang.**Object**
  - java.io.**Console** (implements java.io.Flushable)
  - java.io.**File** (implements java.lang.Comparable<T>, java.io.Serializable)
  - java.io.**FileDescriptor**
  - java.io.**InputStream** (implements java.io.Closeable)
    - java.io.**ByteArrayInputStream**
    - java.io.**FileInputStream**
    - java.io.**FilterInputStream**
      - java.io.**BufferedInputStream**
      - java.io.**DataInputStream** (implements java.io.DataInput)
      - java.io.**LineNumberInputStream**
      - java.io.**PushbackInputStream**
    - java.io.**ObjectInputStream** (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
    - java.io.**PipedInputStream**
    - java.io.**SequenceInputStream**
    - java.io.**StringBufferInputStream**
  - java.io.**ObjectInputFilter.Config**
  - java.io.**ObjectInputStream.GetField**
  - java.io.**ObjectOutputStream.PutField**
  - java.io.**ObjectStreamClass** (implements java.io.Serializable)
  - java.io.**ObjectStreamField** (implements java.lang.Comparable<T>)
  - java.io.**OutputStream** (implements java.io.Closeable, java.io.Flushable)
    - java.io.**ByteArrayOutputStream**
    - java.io.**FileOutputStream**
    - java.io.**FilterOutputStream**
      - java.io.**BufferedOutputStream**
      - java.io.**DataOutputStream** (implements java.io.DataOutput)
      - java.io.**PrintStream** (implements java.lang.Appendable, java.io.Closeable)
    - java.io.**ObjectOutputStream** (implements java.io.ObjectOutput, java.io.ObjectStreamConstants)
    - java.io.**PipedOutputStream**
  - java.security.**Permission** (implements java.security.Guard, java.io.Serializable)
    - java.security.**BasicPermission** (implements java.io.Serializable)
      - java.io.**SerializablePermission**
    - java.io.**FilePermission** (implements java.io.Serializable)
  - java.io.**RandomAccessFile** (implements java.io.Closeable, java.io.DataInput, java.io.DataOutput)
  - java.io.**Reader** (implements java.io.Closeable, java.lang.Readable)
    - java.io.**BufferedReader**
      - java.io.**LineNumberReader**
    - java.io.**CharArrayReader**
    - java.io.**FilterReader**
      - java.io.**PushbackReader**
    - java.io.**InputStreamReader**
      - java.io.**FileReader**
    - java.io.**PipedReader**
    - java.io.**StringReader**
  - java.io.**StreamTokenizer**

Java I/O streams

- I/O Streams, a powerful concept that greatly simplifies I/O operations
- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text
- I/O from the Command Line
- Data Streams handle binary I/O of primitive data type and String values
- Object Streams handle binary I/O of objects

Java I/O Streams

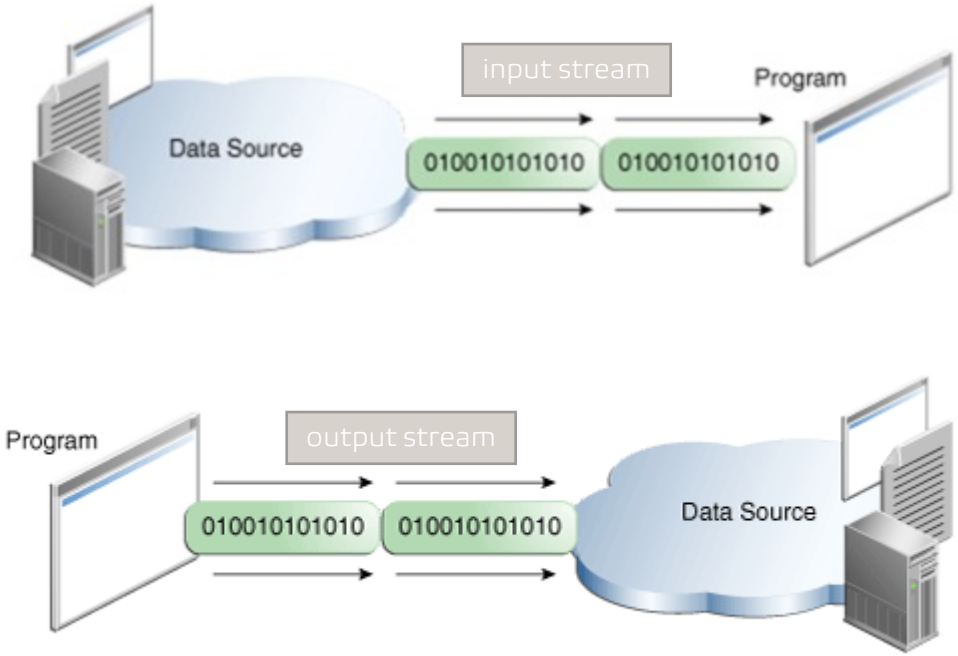
A stream is a sequence of data

I/O Stream represents an input source or an output destination

A stream can represent many kinds of sources and destinations, including disk files, devices, other programs, and memory arrays

Streams support many kinds of data, including simple bytes, primitive data types, localized characters, and objects

- A program uses an input stream to read data from a source, and uses an output stream to write data to a destination, one item at time



I/O Streams

Byte Stream

- Perform input and output of 8-bit bytes (low-level I/O)
- All byte stream classes are descended from abstract classes InputStream and OutputStream, for example
  - File I/O byte streams FileInputStream and FileOutputStream
  - FileInputStream/FileOutputStream read/write bytes from/to a file in a file system; they are meant for reading/writing streams of raw bytes such as image data.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
...
FileInputStream inFile;
FileOutputStream outFile;
...
try {
    inFile = new FileInputStream("myin.dat");
    outFile = new FileOutputStream("myout.dat");
    int c;
    while ((c = in.read()) != -1)
        out.write(c);
}
catch (FileNotFoundException ex) { }
finally { }
```

Closing I/O Streams

- Closing a stream when it's no longer needed is very important
- Avoid serious resource leaks!

```
try { ... }
Catch ( ... ) { }
finally {
    if (inFile != null)
        in.close();
    if (outFile != null)
        out.close();
}
```

Try-with-Resources

```
try ( BufferedReader br = new BufferedReader(new
                                        FileReader(path)); )
{
    return br.readLine();
}
```

- A resource is declared and created followed by the keyword try in the parentheses; the resources must be a subtype of AutoCloseable
- Multiple resources can be declared and created inside the parentheses.
- After the block is finished, the resource's close() method is automatically invoked to close the resource.
- Interface java.lang.AutoCloseable and java.io.closeable
- BufferedReader br will be closed regardless of whether the try statement completes normally or abruptly

Character Streams

Java platform stores character values using Unicode conventions

Character stream I/O automatically translates this internal format to and from the local character set

Ready for internationalization

All character stream classes are descended from Reader and Writer

For example, FileReader and FileWriter

Character Streams

Character streams are often "wrappers" for byte streams

The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes

Example—FileReader and FileWriter

```
import java.io.FileReader;
import java.io.FileWriter;
...
FileReader reader;
FileWriter writer;
try {
    reader = new FileReader("mycharIn.dat");
    writer = new FileWriter("mycharOutt.txt");
    int c;
    while ((c = reader.read()) != -1)
        writer.write(c);
}
catch (FileNotFoundException ex) { }
finally { }
```

- FileReader
  - Reads text from character files using a default buffer size.
  - Decoding from bytes to characters uses the platform's default charset.
  - For reading streams of characters
- FileWriter
  - Writes text to character files using a default buffer size.
  - Encoding from characters to bytes uses the platform's default charset
  - For writing streams of characters

Filter Streams

Filter streams are streams that filter bytes for some purpose.

The basic byte input stream provides a read method that can be used only for reading bytes.

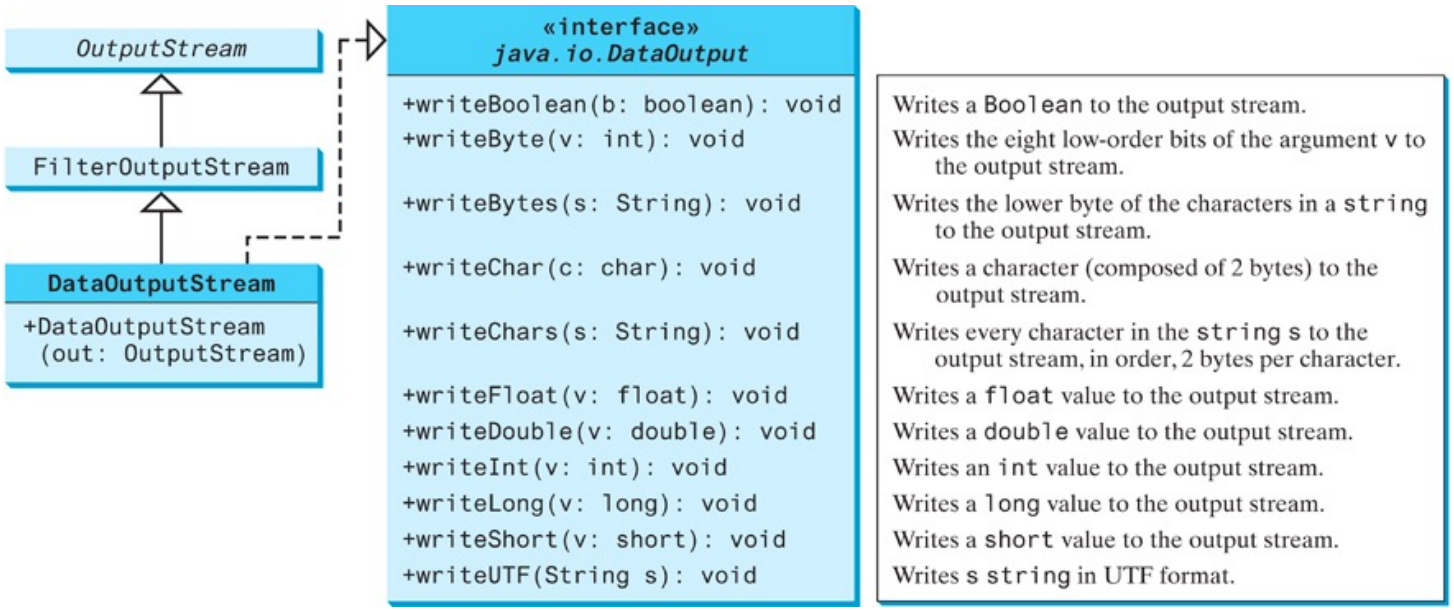
If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream.

Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters.

FilterInputStream and FilterOutputStream are the base classes for filtering data.

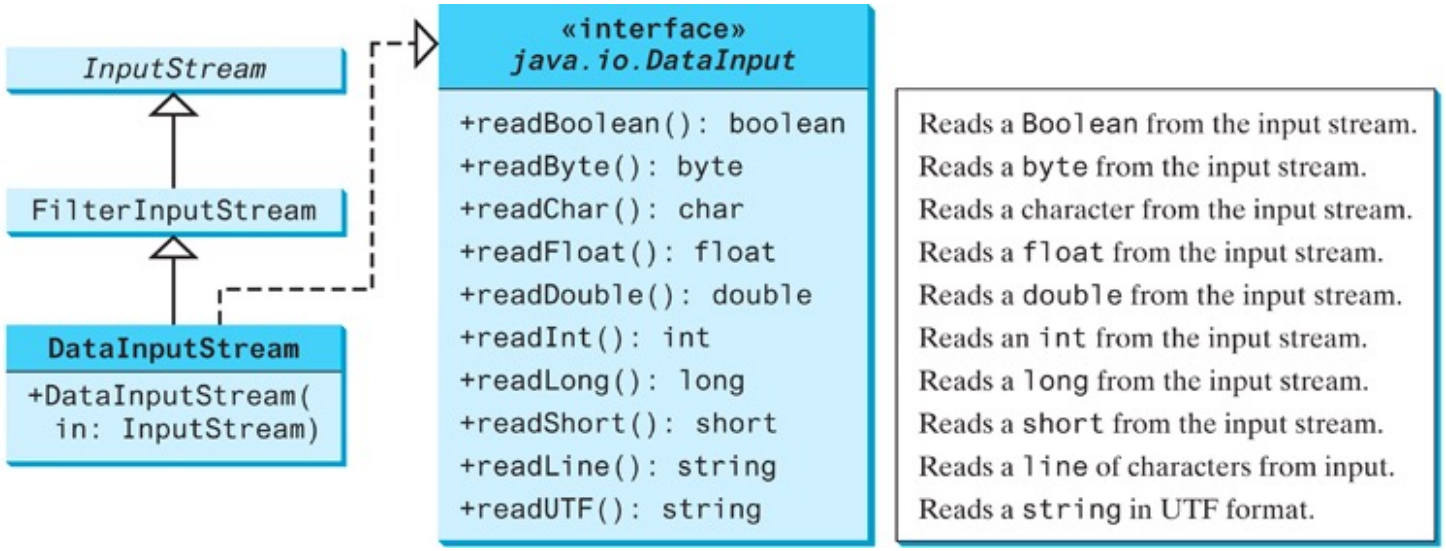
When you need to process primitive numeric types, use DataInputStream and DataOutputStream to filter bytes.

Dataoutputstream Class



- DataOutputStream converts primitive-type values or strings into bytes and outputs the bytes to the stream.
- Enable you to write primitive data-type values and strings into an output stream.

Datainputstream Class



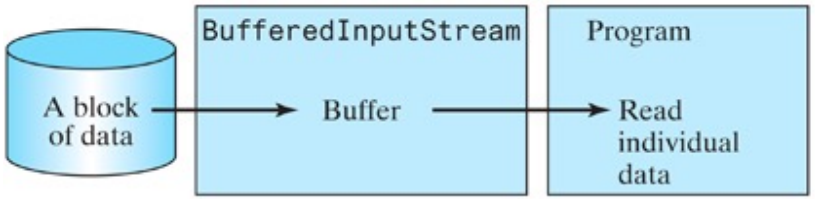
- DataInputStream reads bytes from the stream and converts them into appropriate primitive-type values or strings.
- Filter an input stream of bytes into primitive data-type values and strings.

Buffered Streams

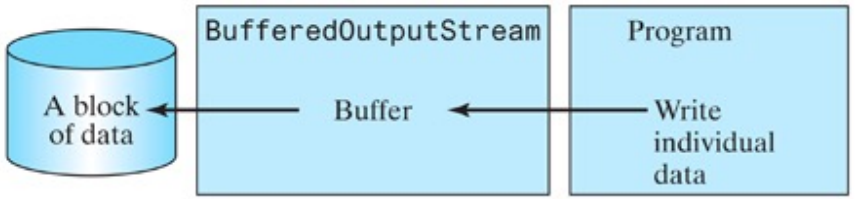
- Unbuffered I/O—this means each read or write request is handled directly by the OS (Operating System)
  - Less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive
- Buffered I/O streams—read/write data from a memory area known as a buffer
  - The native input/output API is called only when the buffer is empty/full
  - Four buffered stream classes
    - Byte Streams—BufferedInputStream, BufferedOutputStream
    - Character Streams—BufferedReader, BufferedWriter

Buffered Streams

- BufferedInputStream/BufferedOutputStream can be used to speed up input and output by reducing the number of disk reads and writes.
- Using BufferedInputStream, the whole block of data on the disk is read into the buffer in the memory at a time
- The individual data are then loaded to your program from the buffer
- Using BufferedOutputStream, the individual data are first written to the buffer in the memory. When the buffer is full, all data in the buffer are written to the disk at a time



(a)



(b)

Buffered Streams

- BufferedInputStream/BufferedOutputStream does not contain new methods; they inherit the methods from the InputStream/OutputStream classes.
- It manages a buffer behind the scene and automatically reads/writes data from/to disk on demand.
- You can wrap a BufferedInputStream/BufferedOutputStream on any InputStream/ OutputStream using the constructors

```
inStream = new BufferedReader(new FileReader("myinput.dat"));

outStream = new BufferedWriter(new FileWriter("myoutput.dat"));
```

Constructors

Constructor
`BufferedOutputStream(OutputStream out)`
`BufferedOutputStream(OutputStream out, int size)`

Constructors

Constructor
`BufferedInputStream(InputStream in)`
`BufferedInputStream(InputStream in, int size)`

Flushing Buffered Streams

Write out a buffer at critical points, without waiting for it to fill (full)

Some buffered output classes support autoflush, specified by an optional constructor argument

For example, an autoflush PrintWriter object flushes the buffer on every invocation of println() or format()

flush() method is valid on any output stream, but has no effect unless the stream is buffered

You should always use buffered I/O to speed up input and output.

For small files, you may not notice performance improvements. However, for large files—over 100 MB—you will see substantial improvements using buffered I/O.

Example – copying files

```
File sourceFile = newFile(args[0]); //getting the source file name from command line
if (!sourceFile.exists()) {
    System.out.println("Source file " + args[0] + " does not exist");
    System.exit(2);
}
File targetFile = newFile(args[1]); //getting the destination file name from command line
if (targetFile.exists()) {
    System.out.println("Target file " + args[1] + " already exists");
    System.exit(3);
}
try (BufferedInputStream input = new BufferedInputStream(new FileInputStream(sourceFile));
    BufferedOutputStream output = new BufferedOutputStream(new FileOutputStream(targetFile));)
{
    int r, numberOfBytesCopied = 0;
    while ((r = input.read()) != -1) { //while not EOF, keep copying the bytes
        output.write((byte)r);
        numberOfBytesCopied++;
    }
    System.out.println(numberOfBytesCopied + " bytes copied");
}
```

Object I/O

ObjectInputStream/ObjectOutputStream classes can be used to read/write serializable objects.

ObjectInputStream/ObjectOutputStream enables you to perform I/O for objects in addition to primitive-type values and strings.

Since ObjectInputStream/ ObjectOutputStream contains all the functions of DataInputStream/DataOutputStream, you can replace DataInputStream/DataOutputStream completely with ObjectInput Stream/ObjectOutputStream.

Object I/O – example

```
try (ObjectOutputStream output =
    new ObjectOutputStream(new FileOutputStream("object.dat")); )
{
    output.writeUTF("John");
    output.writeDouble(85.5);
    output.writeObject(new java.util.Date());
}
...
String name = input.readUTF();
double score = input.readDouble();
Date date = (Date)(input.readObject());
```

The Serializable Interface

- Not every object can be written to an output stream. Objects that can be written are said to be serializable.
- A serializable object is an instance of the java.io.Serializable interface, so the object's class must implement Serializable.
- The Serializable interface is a marker interface. Since it has no methods, you don't need to add additional code in your class that implements Serializable.
- Implementing this interface enables the Java serialization mechanism to automate the process of storing objects and arrays.

Serialization

- To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object.
- To appreciate this automation feature, consider what you otherwise need to do in order to store an object.
- Suppose that you wish to store an ArrayList object. To do this, you need to store all the elements in the list. Each element is an object that may contain other objects. As you can see, this would be a very tedious process.
- Java provides a built-in mechanism to automate the process of writing objects. This process is referred as object serialization, which is implemented in ObjectOutputStream.
- In contrast, the process of reading objects is referred as object deserialization, which is implemented in ObjectInputStream.

Serialization

- Many classes in the Java API implement Serializable.
- All the wrapper classes for primitive-type values implement java.io.Serializable.
- Attempting to store an object that does not support the Serializable interface would cause a NotSerializableException.
- When a serializable object is stored, the class of the object is encoded; this includes the class name and the signature of the class, the values of the object's instance variables, and the closure of any other objects referenced by the object. The values of the object's static variables are not stored.

The transient Keyword

- If an object is an instance of Serializable but contains nonserializable instance data fields, can it be serialized? The answer is NO.
- To enable the object to be serialized, mark these data fields with the transient keyword to tell the JVM to ignore them when writing the object to an object stream

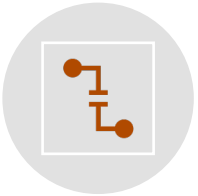
```
public class C implements java.io.Serializable {
    private int v1;
    private static double v2; //not serialized
    private transient A v3 = new A(); //not serialized
}
```

Serialization

If an object is written to an object stream more than once, will it be stored in multiple copies? No, it will not.

When an object is written for the first time, a serial number is created for it.

The JVM writes the complete contents of the object along with the serial number into the object stream.



After the first time, only the serial number is stored if the same object is written again.

When the objects are read back, their references are the same since only one object is actually created in the memory.

Serializing arrays

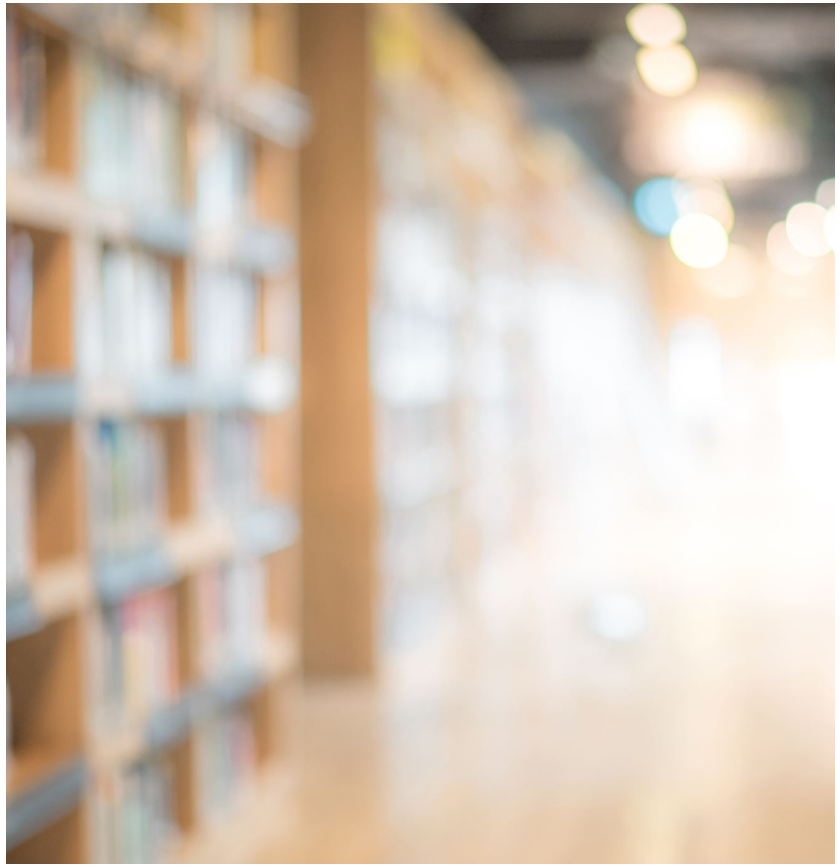
An array is serializable if all its elements are serializable.

An entire array can be saved into a file using writeObject and later can be restored using readObject.

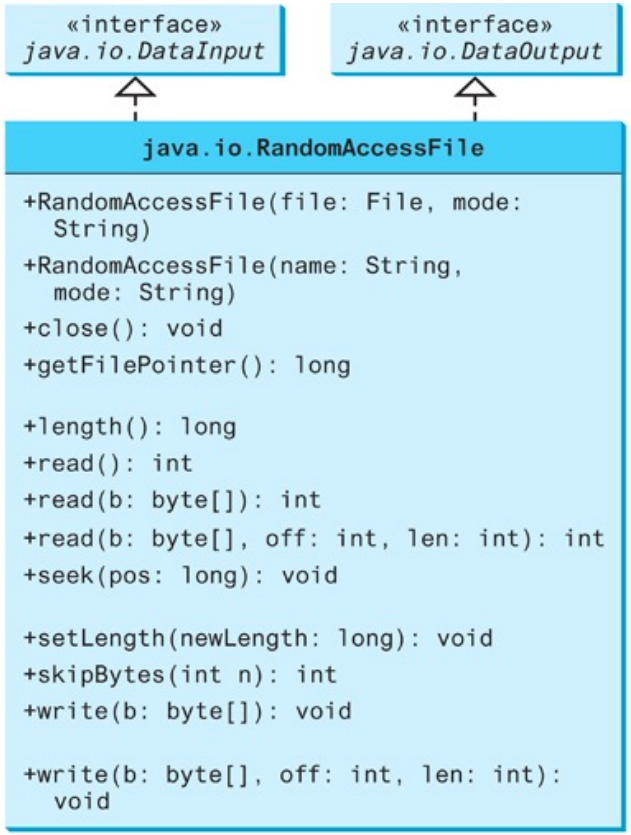
```
int[] numbers = {1, 2, 3, 4, 5};
String[] strings = {"John", "Susan", "Kim"};
try ( ObjectOutputStream output = new ObjectOutputStream( new FileOutputStream("array.dat",true)); )
{
    output.writeObject(numbers);
    output.writeObject(strings);
}
...
int[] newNumbers = (int[])(input.readObject());
String[] newStrings = (String[])(input.readObject());
```

Random Access Files

- Java provides the RandomAccessFile class to allow data to be read from and written to at any locations in the file.
- All of the streams you have used so far are known as read-only OR write-only streams. These streams are called sequential streams.
- A file that is opened using a sequential stream is called a sequential-access file. The contents of a sequential-access file cannot be updated.
- RandomAccessFile class to allow data to be read from and written to at any locations in the file. A file that is opened using the -RandomAccessFile class is known as a random-access file.



RANDOM ACCESS FILES



`RandomAccessFile(File file, String mode)`	Creates a `RandomAccessFile` stream with the specified `File` object and mode.
`RandomAccessFile(String name, String mode)`	Creates a `RandomAccessFile` stream with the specified file name string and mode.
`close()`	Closes the stream and releases the resource associated with it.
`getFilePointer()`	Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.
`length()`	Returns the length for this file.
`read()`	Reads a byte of data from this file and returns –1 at the end of stream.
`read(b)`	Reads up to `b.length` bytes of data from this file into an array of bytes.
`read(b, off, len)`	Reads up to `len` bytes of data from this file into an array of bytes.
`seek(pos)`	Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.
`setLength(newLength)`	Sets a new length for this file.
`skipBytes(int n)`	Skips over *n* bytes of input.
`write(b)`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
`write(b, off, len)`	Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

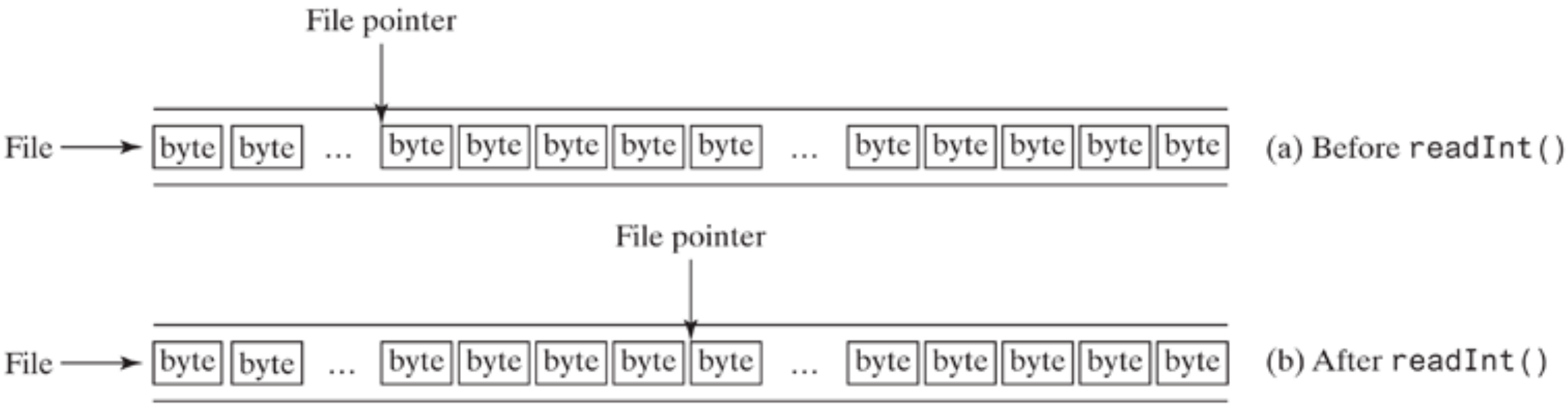
Random Access files – Access Modes

"r" – Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.

"rw" – Open for reading and writing. If the file does not already exist then an attempt will be made to create it.

"rws" – Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.

"rwd" – Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device.



Random access files – file pointer

- A random-access file consists of a sequence of bytes. A special marker called a file pointer is positioned at one of these bytes.
- A read or write operation takes place at the location of the file pointer.
- When a file is opened, the file pointer is set at the beginning of the file.
- When you read from or write data to the file, the file pointer moves forward to the next data item