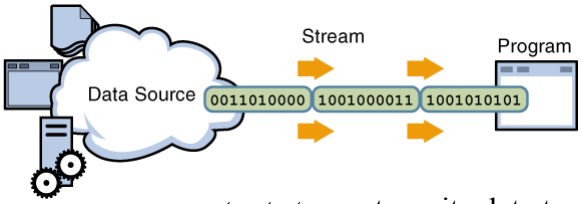
I/O Streams Basics

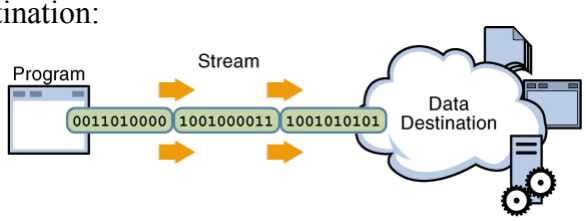
- An *I/O Stream* represents an input source or an output destination
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, etc.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways

Reading/Writing to Stream

A stream is a sequence of data.
A program uses an *input stream* to read data from a source :



A program uses an *output stream* to write data to a destination:



Types of Streams

- Byte Streams
- Character Streams
- Buffered Streams

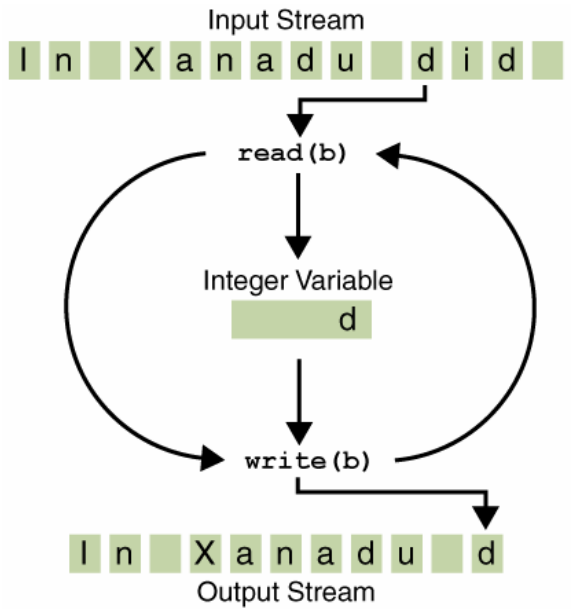
Byte Streams

- Programs use *byte streams* to perform input and output of 8-bit bytes
- All byte stream classes are descended from InputStream and OutputStream
- There are many byte stream classes
  - You will find out!!
- Examples
  - FileInputStream
  - FileOutputStream

Example to copy a file

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes { public static void main(String[] args) throws IOException
{
        FileInputStream in = null;
        FileOutputStream out = null;
                try {
                in = new FileInputStream("xanadu.txt");
                out = new FileOutputStream("outagain.txt");
                int c;
                while ((c = in.read()) != -1) {
                        out.write(c); }
                } finally {
                 if (in != null) {
                in.close();
                }
                if (out != null)
                { out.close(); }
}} }
```



Important Methods

- int **read**() : Reads a byte of data from this input stream
- int **read**(byte[] b) : Reads up to b.length bytes of data from this input stream into an array of bytes
- void **write** (int b) : Writes the specified byte to this file output stream.
- void **write**(byte[] b) : Writes b.length bytes from the specified byte array to this file output stream.
- void **close**() : Closes the file input and output streams and releases any system resources associated with the stream

Analysis

- Notice that read() returns an int value. If the input is a stream of bytes, why doesn't read() return a byte value?
  - Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.
- Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs This practice helps avoid serious *resource leaks*

Character Streams

- The Java platform stores character values using Unicode conventions
- Character stream I/O automatically translates this internal format to and from the local character set
- For most applications, I/O with character streams is no more complicated than I/O with byte streams
- A program that uses character streams in place of byte streams automatically adapts to the local character set

Character Streams

- All character stream classes are descended from *Reader* and *Writer*
- 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
- FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

CopyCharacters Example

```
import java.io.FileReader;
 import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {

        public static void main(String[] args) throws IOException {
                FileReader inputStream = null;
                FileWriter outputStream = null;
                 try {
                        inputStream = new FileReader("xanadu.txt");
                        outputStream = new FileWriter("characteroutput.txt");
                        int c;
                        while ((c = inputStream.read()) != -1) { outputStream.write(c); }
                } finally {
                 if (inputStream != null)
                { inputStream.close();
                } if (outputStream != null)
                { outputStream.close();
        } } } }
```

Buffered Streams

- The examples shown so far use *unbuffered* I/O
- This means each read or write request is handled directly by the underlying OS
- This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive
- To reduce this kind of overhead, the Java platform implements *buffered* I/O streams

Buffered Streams

- Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty
- Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full

Buffered Streams

- A program can convert a unbuffered stream into a buffered stream using the wrapping

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new
    FileWriter("characteroutput.txt"));
```

- There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

Flushing Buffered Streams

- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer
- To flush a stream manually, invoke its `flush` method.
- The flush method is valid on any output stream, but has no effect unless the stream is buffered
- Some buffered output classes support *autoflush*, specified by an optional constructor argument.
  - When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format
- The flush method is valid on any output stream, but has no effect unless the stream is buffered.

Line Oriented I/O

- Character I/O usually occurs in bigger units than single characters.
- One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n").
- Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

CopyLine Example

```
import java.io.FileReader;
import java.io.FileWriter;
 import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines { public static void main(String[] args) throws IOException {
BufferedReader inputStream = null;
PrintWriter outputStream = null;
        try { inputStream = new BufferedReader(new FileReader("xanadu.txt"));
        outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));
        String l;
         while ((l = inputStream.readLine()) != null) {
        outputStream.println(l); }
        } finally { if (inputStream != null)
        { inputStream.close(); } if (outputStream != null)
        { outputStream.close(); } } } }
```

Scanning and Formatting

- Programming I/O often involves translating to and from the neatly formatted data humans like to work with.
- To assist you with these chores, the Java platform provides two APIs.
  - The scanner API breaks input into individual tokens associated with bits of data.
  - The formatting API assembles data into nicely formatted, human-readable form.

Scanning

- Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.
- By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators)

Scanning Example

- ScanXan, a program that reads the individual words in xanadu.txt

```
import java.io.*;
import java.util.Scanner;
public class ScanXan {

public static void main(String[] args) throws IOException {
Scanner s = null;
try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
            while (s.hasNext()) {
                        System.out.println(s.next());
            }
} finally {
if (s != null)
{ s.close(); } } } }
```

Notice that ScanXan invokes Scanner's close method when it is done

Output of ScanXan

In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome

Homework Question1!!

- Translating Individual Tokens
  - The ScanXan example treats all input tokens as simple String values. Scanner also supports tokens for all of the Java language's primitive types
  - http://java.sun.com/javase/6/docs/api/java/util/Scanner.html

Formatting

- Stream objects that implement formatting are instances of either PrintWriter, a character stream class, and PrintStream, a byte stream class.
- Like all byte and character stream objects, instances of PrintStream and PrintWriter implement a standard set of write methods for simple byte and character output.
- In addition, both PrintStream and PrintWriter implement the same set of methods for converting internal data into formatted output.
- Two levels of formatting are provided:
  - `print` and `println` format individual values in a standard way.
  - `format` formats almost any number of values based on a format string, with many options for precise formatting.

- Write java code to print to find square root of 2 and print the result as

  The square root of 2 is 1.4142135623730951.

The format Method

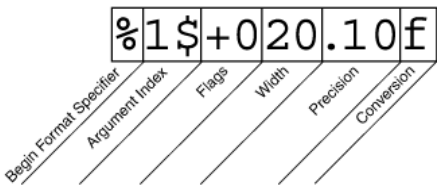
- The format method formats multiple arguments based on a *format string*.
- The format string consists of static text embedded with *format specifiers*; except for the format specifiers, the format string is output unchanged.

```
public class Root2
{
public static void main(String[] args)
{
int i = 2;
double r = Math.sqrt(i);
System.out.format("The square root of %d is %f.%n", i, r);
 }
}
```

d formats an integer value as a decimal value.
f formats a floating point value as a decimal value.
n outputs a platform-specific line terminator.

Homework Q2: What is the output??

```
public class Format {
   public static void main(String[] args) {
       System.out.format("%f, %1$+020.10f
       %n", Math.PI);
   }
}
```



- The elements must appear in the order shown. Working from the right, the optional elements are:
  - **Precision**. For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.
  - **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.
  - **Flags** specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
  - The **Argument Index** allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier.
- Thus the example could have said: System.out.format("%f, %<+020.10f %n", Math.PI);

I/O from the Command Line

- The Java platform supports three Standard Streams:
  - *Standard Input*, accessed through System.in;
  - *Standard Output*, accessed through System.out; and
  - *Standard Error*, accessed through System.err
- These standard streams are byte streams
- To use Standard Input as a character stream, wrap System.in in InputStreamReader.

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

ReadingLine Example

```
import java.io.*;
public class ReadString {
  public static void main (String[] args) {
    //  prompt the user to enter their name
    System.out.print("Enter your name: ");
    //  open up standard input
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String userName = null;
    //  read the username from the command-line; need to use try/catch with the
    //  readLine() method
    try {
      userName = br.readLine();
    } catch (IOException ioe) {
      System.out.println("IO error trying to read your name!");
      System.exit(1);
    }
    System.out.println("Thanks for the name, " + userName);
  }
}  // end of ReadString class
```

Data Streams

- Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values
- data streams implement either the DataInput interface or the DataOutput interface
- The DataStreams example demonstrates data streams by writing out a set of data records
- Homework Q3: Write a program to read the set of data records

Example code

```
static final String dataFile = "invoicedata";
 static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
"Java T-shirt", "Java Mug", "Duke Juggling Dolls",
"Java Pin", "Java Key Chain" };

 out = new DataOutputStream(new BufferedOutputStream(new
 FileOutputStream(dataFile)));


 for (int i = 0; i < prices.length; i ++) {
 out.writeDouble(prices[i]);
 out.writeInt(units[i]); o
 ut.writeUTF(descs[i]);
 }
```

Non-Stream File I/O

- The File class makes it easier to write platform-independent code that examines and manipulates files.
- The name of this class is misleading: File instances represent *file names*, not files. The file corresponding to the file name might not even exist.
- Why create a File object for a file that doesn't exist?
  - A program can use the object to parse a file name. Also, the file can be created by passing the File object to the constructor of some classes, such as FileWriter.
  - If the file *does* exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, or changing its permissions.

- A File object contains the file name string used to construct it.
  - That string never changes throughout the lifetime of the object.
  - The following code creates a File object with the constructor invocation
    - File a = new File("xanadu.txt");

Methods

Method Invoked	Returns on Microsoft Windows	Returns on Solaris
a.toString()	xanadu.txt	xanadu.txt
a.getName()	xanadu.txt	xanadu.txt
b.getParent()	NULL	NULL
a.getAbsolutePath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt
a.getCanonicalPath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt

File b = new File(".." + File.separator + "examples" + File.separator + "xanadu.txt");

Method Invoked	Returns on Microsoft Windows	Returns on Solaris
b.toString()	..\examples\xanadu.txt	../examples/xanadu.txt
b.getName()	xanadu.txt	xanadu.txt
b.getParent()	..\examples	../examples
b.getAbsolutePath()	c:\java\examples\..\examples\xanadu.txt	/home/cafe/java/examples/../examples/xanadu.txt
b.getCanonicalPath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt

Manipulating Files

- If a File object names an actual file, a program can use it to perform a number of useful operations on the file.
  - These include passing the object to the constructor for a stream to open the file for reading or writing.
- The delete method deletes the file immediately, while the deleteOnExit method deletes the file when the virtual machine terminates.
- The setLastModified sets the modification date/time for the file. For example, to set the modification time of xanadu.txt to the current time, a program could do
  - new File("xanadu.txt").setLastModified(new Date().getTime());
- The renameTo() method renames the file. Note that the file name string behind the File object remains unchanged, so the File object will not refer to the renamed file.

Working with Directories

- File has some useful methods for working with directories.
- The mkdir method creates a directory. The mkdirs method does the same thing, after first creating any parent directories that don't yet exist.
- The list and listFiles methods list the contents of a directory.
  - The list method returns an array of String file names, while listFiles returns an array of File objects.

Assignment

- Q1, Q2 and Q3
- Write a program to ask for a person's name and age and print a message in response to the user input. For example, a sample run could look like this

```
Enter name: Jam Jenkins
Enter age: 18
Hello Jam Jenkins, you are 18 years old.
```

- Write a program that will allow five names to be entered from keyboard and appended to an existing file, Again read the file and print the output.
- Write a program that reads a specified file and searches for a specified word, printing each line number and line in which the word is found.