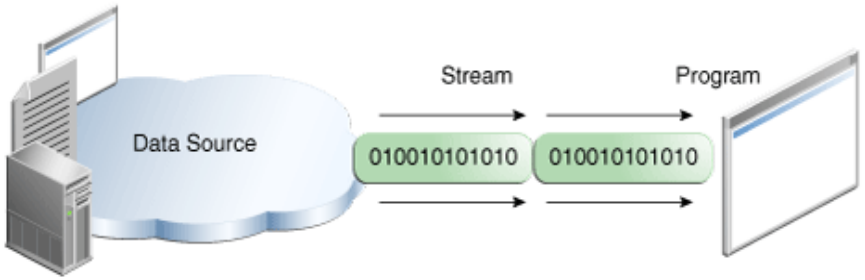
Basic I/O

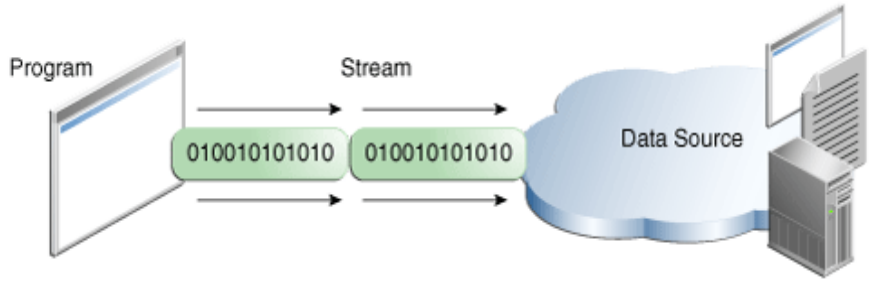
I/O Streams

- 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, and memory arrays.
- 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.

I/O Streams



Reading information into a program



Writing information from a program

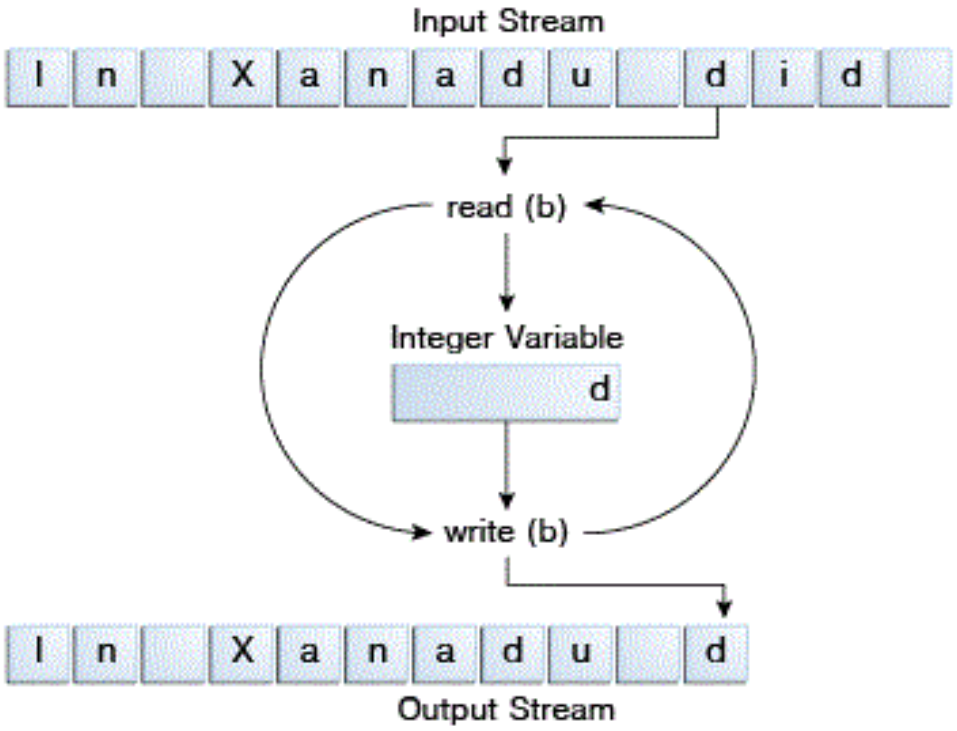
Byte Streams

- 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. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream.
- Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Using Byte Streams

```
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();    }
        }
    }
}
```

Simple Byte Stream Input and Output



Always Close Streams

- 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.
- One possible error is that CopyBytes was unable to open one or both files.
  - When that happens, the stream variable corresponding to the file never changes from its initial null value.
  - That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

When Not to Use Byte Streams

- CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid.
- Since xanadu.txt contains character data, the best approach is to use character streams.
- There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.
- So why talk about byte streams? Because all other stream types are built on byte streams.

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.
- In Western locales, the local character set is usually an 8-bit superset of ASCII.
- For most applications, I/O with character streams is no more complicated than I/O with byte streams.
- Input and output done with stream classes automatically translates to and from the local character set.

Using Character Streams

- All character stream classes are descended from Reader and Writer.
- As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter.

Using Character Streams

```
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();  }
      }
   }
}
```

Character Streams that Use Byte 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.
  - FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.
  - There are two general-purpose byte-to-character "bridge" streams: InputStreamReader and OutputStreamWriter. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

Buffered Streams

- Most of the examples we've seen 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 input streams read data from a memory area known as a *buffer*. Similarly, buffered output streams write data to a buffer.

Buffered Streams

- A program can convert an unbuffered stream into a buffered stream using the wrapping idiom, where the unbuffered stream object is passed to the constructor for a buffered stream class.
- Example:

```
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.
- 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.
- 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.

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.)

Breaking Input into Tokens

```
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();
            }
        }
    }
}
```

Breaking Input into Tokens

The output of ScanXan looks like this:

- In
- Xanadu
- did
- Kubla
- Khan
- A
- stately
- pleasure-dome
- …

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 (except for char), as well as BigInteger and BigDecimal.
- Also, numeric values can use thousands separators. Thus, in a US locale, Scanner correctly reads the string "32,767" as representing an integer value.

Formatting

- Stream objects that implement formatting are instances of either PrintWriter, a character stream class, or 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.

The print and println Methods

```
public class Root {
   public static void main(String[] args) {
      int i = 2;
      double r = Math.sqrt(i);
            System.out.print("The square root of ");
      System.out.print(i);
      System.out.print(" is ");
      System.out.print(r);
      System.out.println(".");
      i = 5;
      r = Math.sqrt(i);
      System.out.println("The square root of " + i + " is " + r + ".");
   }
}
```

The format Method

```
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);
    }
}
```

Here is the output:

```
The square root of 2 is 1.414214.
```

The format Method

- All format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated. The three conversions used here are:
  - 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.
- Here are some other conversions:
  - x formats an integer as a hexadecimal value.
  - s formats any value as a string.
  - tB formats an integer as a locale-specific month name.

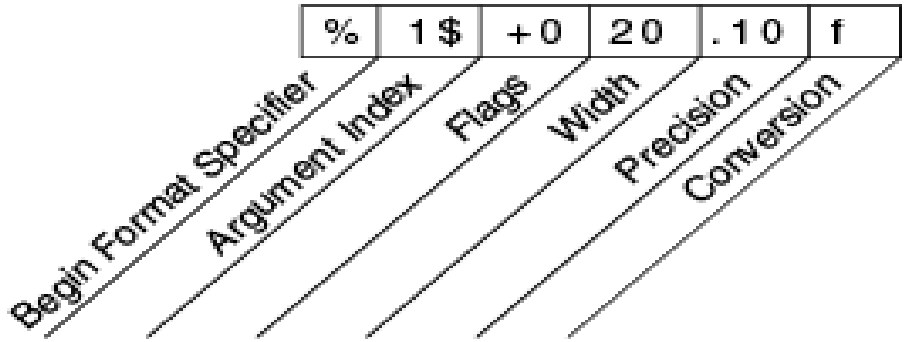
The format Method-Example

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

Here's the output:

```
3.141593, +00000003.1415926536
```

The format Method



The elements must appear in the order shown.

- 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.

The format Method

- 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.

I/O from the Command Line -Standard Streams

- The Java platform supports three Standard Streams:
  - *Standard Input*, accessed through System.in;
  - *Standard Output*, accessed through System.out;
  - *Standard Error*, accessed through System.err.
- These objects are defined automatically and do not need to be opened.
- Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages.

I/O from the Command Line -Standard Streams

- You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams.
- System.out and System.err are defined as PrintStream objects.
- Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.
- By contrast, System.in is a byte stream with no character stream features.
- To use Standard Input as a character stream, wrap System.in in InputStreamReader.

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

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.
- All data streams implement either the DataInput interface or the DataOutput interface.
  - Example: DataInputStream and DataOutputStream.

Example

Order in record	Data type	Data description	Output Method	Input Method	Sample Value
1	double	Item price	DataOutputStream .writeDouble	DataInputStream .readDouble	19.99
2	int	Unit count	DataOutputStream .writeInt	DataInputStream .readInt	12
3	String	Item description	DataOutputStream .writeUTF	DataInputStream .readUTF	"Java T-Shirt

Example

```
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"
};
```

Example

- DataStreams opens an output stream.
- Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, DataStreams provides a buffered file output byte stream.

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

DataStreams writes out the records and closes the output stream.

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

Example

- Now DataStreams reads the data back in again.
- First it must provide an input stream, and variables to hold the input data.
- Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

```
in = new DataInputStream(new
        BufferedInputStream(new FileInputStream(dataFile)));
double price;
int unit;
String desc;
double total = 0.0;
```

Example

Now DataStreams can read each record in the stream, reporting on the data it encounters.

```
try {
   while (true) {
      price = in.readDouble();
      unit = in.readInt();
      desc = in.readUTF();
      System.out.format("You ordered %d" + " units of %s at $%.2f%n",
         unit, desc, price);
      total += unit * price;
   }
} catch (EOFException e) {
}
```

Data Streams

- DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values.
- In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.
- The correct type to use for currency values is java.math.BigDecimal. Unfortunately, BigDecimal is an object type, so it won't work with data streams.
- However, BigDecimal will work with object streams.

Object Streams

- Just as data streams support I/O of primitive data types, object streams support I/O of objects.
- Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

Object Streams

- The object stream classes are ObjectInputStream and ObjectOutputStream.
- These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.
- That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values.

Object Streams

```
import java.io.*;
public class TestObjectIO {
    public static void main(String args[]) throws Exception {
      T t = new T();
      t.k = 8;
      FileOutputStream fos = new FileOutputStream("testobjectio.dat");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(t);
      os.flush();
      oos.close();
      FileInputStream fis = new FileInputStream("testobjectio.dat");
      ObjectInputStream ois = new ObjectInputStream(fis);
      T tReaded = (T)ois.readObject();
      System.out.println(tReaded.i + " " + tReaded.j + " " + tReaded.d + " " + tReaded.k);
    }
}
```

Object Streams

```
class T
    implements Serializable
{
    int i = 10;
    int j = 9;
    double d = 2.3;
    transient int k = 15;
}
```