Pipes

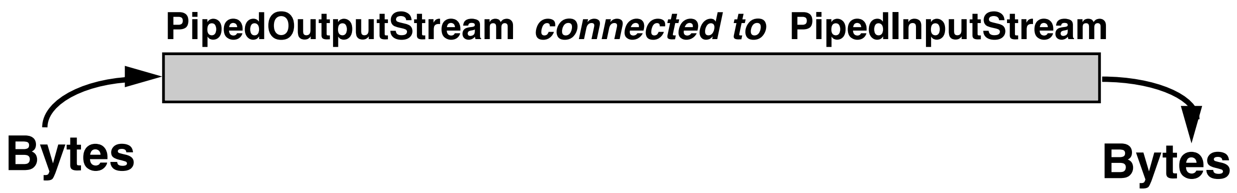
A pipe is a stream mechanism that connects two threads so that output from one becomes input to the other.

Properties of a Pipe

- Uni-directional stream
- Buffered communication conduit
- Thread-safe
- Two ends to a pipe, one each for sending and receiving

Pipes come in two varieties

1. byte streams using the classes
   PipedInputStream
   PipedOutputStream
2. character streams using the classes
   PipedReader
   PipedWriter


PipedOutputStream ***connected to*** **PipedInputStream**
Bytes
Bytes

Constructing a Pipe

Create two pipe stream objects, one for output and one for input, and connect the objects to form a single pipe.

```
   PipedOutputStream pipeOut =new PipedOutputStream();

   PipedInputStream pipeIn = new PipedInputStream();

   pipeOut.connect(pipeIn);
            or
   pipeIn.connect(pipeOut);
```

Alternatively, the second piped stream can be connected to the first using another constructor:

```
   PipedInputStream pipeIn =
                        new PipedInputStream(pipeOut);
```

Note that the pipe just set up provides a byte stream whose ends are of type OutputStream and InputStream, respectively.

The ends of the pipe can then be wrapped as a DataOutputStream and DataInputStream for sending binary data from one thread to another.

Example: Hamming Numbers

Generate the sequence of integers consisting of 1 and the numbers whose only factors are 2, 3, and 5, in ascending order with no duplicates.

Want to produce the increasing sequence of all numbers of the form $2^i \cdot 3^j \cdot 5^k$ where $i \geq 0$, $j \geq 0$, and $k \geq 0$.

Two Observations

- If h is a Hamming number, then so are 2•h, 3•h, and 5•h.
- If n≠1 is a Hamming number, then n = 2•h, n = 3•h, or n = 5•h for some Hamming number h.

Note that there may be more than one way of calculating n.

First 15 Hamming numbers:

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24

Note that 16 = 2•8 whereas 20 = 2•10 and 5•4.

Solution using Four Threads

- One thread receives a number from the main method and sends back 2 times that number.
- Another thread receives a number from the main method and sends back 3 times that number.
- Another thread receives a number from the main method and sends back 5 times that number.
- The main method, also a thread, prints 1 and sends 1 to each of the other threads. It then receives the numbers sent by the other threads, printing the next larger number and sending it back to the other threads.

This solution requires six pipes connected to the main method for sending and receiving numbers to and from each of the other three threads.

The three threads that multiply by 2, 3, and 5 are instances of the class Multiple, which is given the ends of two pipes in its constructor.

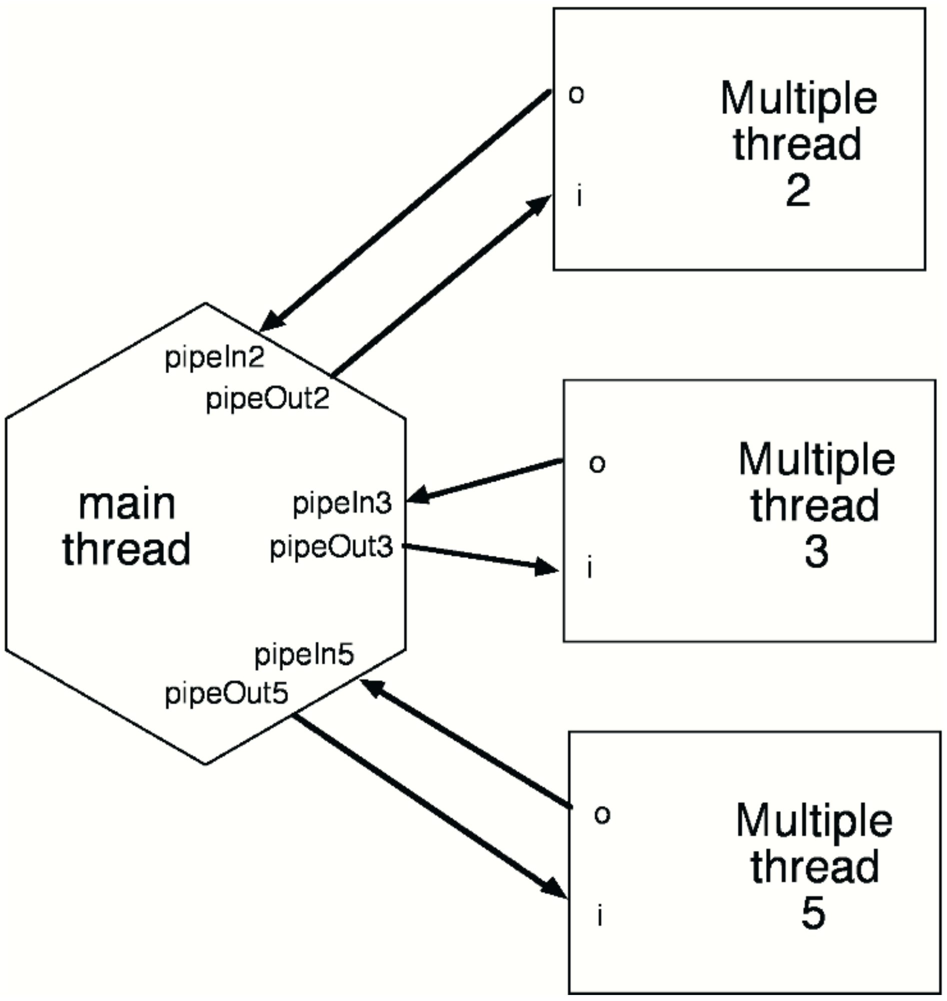
```
}
```

Each thread wraps the pipes as DataInput and DataOutput streams to facilitate communicating binary integers.

```
import java.io.*;
class Multiple extends Thread
{
   private int mult;
   private DataInputStream in;
   private DataOutputStream out;

   public Multiple(int m, PipedInputStream i,
                                    PipedOutputStream o)
   {  mult = m;
      try
      {  in = new DataInputStream(new PipedInputStream(o));
         out = new DataOutputStream(new PipedOutputStream(i));
      }
      catch (IOException e)  { }
   }

   public void run()
   {
      try
      {     while (true)
            {     int k = in.readInt();
                  out.writeInt(mult*k);
            }
      }
      catch (IOException e)  {  }
   }
}
```

The main class sets up the six pipes, starts the three threads, and begins the Hamming sequence by sending the number 1 to each of the threads.

This diagram shows how the six pipes are connected among the four threads.



```
public class Hamming
{
   static int NUMBER = 100;
   public static void main(String [] args)
   {
      try
```

```
{
   PipedInputStream pipeIn2 = new PipedInputStream();
   PipedOutputStream pipeOut2 =
                              new PipedOutputStream();
   new Multiple(2, pipeIn2, pipeOut2).start();

   PipedInputStream pipeIn3 = new PipedInputStream();
   PipedOutputStream pipeOut3 =
                              new PipedOutputStream();
   new Multiple(3, pipeIn3, pipeOut3).start();

   PipedInputStream pipeIn5 = new PipedInputStream();
   PipedOutputStream pipeOut5 =
                              new PipedOutputStream();
   new Multiple(5, pipeIn5, pipeOut5).start();


   DataInputStream myIn2 =
                         new DataInputStream(pipeIn2);
   DataInputStream myIn3 =
                         new DataInputStream(pipeIn3);
   DataInputStream myIn5 =
                         new DataInputStream(pipeIn5);

   DataOutputStream myOut2 =
                         new DataOutputStream(pipeOut2);
   DataOutputStream myOut3 =
                         new DataOutputStream(pipeOut3);
   DataOutputStream myOut5 =
                         new DataOutputStream(pipeOut5);
   myOut2.writeInt(1);          // Send 1 to
   myOut3.writeInt(1);          // each thread
   myOut5.writeInt(1);

// Start generating the sequence
```

```
    int k = 1;      // Counter
    int last = 1;  // Last term in sequence so far
    format(last, 6);    // Formatted output

    int n2 = myIn2.readInt();     // Get first values
    int n3 = myIn3.readInt();     // from each thread
    int n5 = myIn5.readInt();

    while (k < NUMBER)
    {
      k++;
// Skip terms that are not bigger than last
      while (n2 <= last) n2 = myIn2.readInt();
      while (n3 <= last) n3 = myIn3.readInt();
      while (n5 <= last) n5 = myIn5.readInt();

// Select the smallest of available numbers
      if (n2 <= n3 && n2 <= n5)
      {     last = n2;
            format(last, 6);
            myOut2.writeInt(n2);
            myOut3.writeInt(n2);
            myOut5.writeInt(n2);
            n2 = myIn2.readInt();
      }
      else if (n3 <= n2 && n3 <= n5)
      {     last = n3;
            format(last, 6);
            myOut2.writeInt(n3);
            myOut3.writeInt(n3);
```

```
                myOut5.writeInt(n3);
                n3 = myIn3.readInt();
            }
            else if (n5 <= n3 && n5 <= n2)
            {   last = n5;
                format(last, 6);
                myOut2.writeInt(n5);
                myOut3.writeInt(n5);
                myOut5.writeInt(n5);
                n5 = myIn5.readInt();
            }
            if (k%10==0) System.out.println();
          }
        }
        catch (IOException e)  {  }
        System.out.println("That's all");
    }

    static void format(int n, int size)
    {
        String s = String.valueOf(n);
        for (int k=1; k<=size-s.length(); k++)
                                        System.out.print(" ");
        System.out.print(s);
    }
}
```

Output
```
   1    2    3    4    5    6     8    9   10   12
  15   16   18   20   24   25    27   30   32   36
  40   45   48   50   54   60    64   72   75   80
  81   90   96  100  108  120   125  128  135  144
 150  160  162  180  192  200   216  225  240  243
 250  256  270  288  300  320   324  360  375  384
 400  405  432  450  480  486   500  512  540  576
 600  625  640  648  675  720   729  750  768  800
 810  864  900  960  972 1000  1024 1080 1125 1152
1200 1215 1250 1280 1296 1350  1440 1458 1500 1536
That's all
```

Example: Character Stream Pipes

Problem: Read text from a file and write it to another file, removing multiple spaces and formatting the text so that no line has more than 30 characters.

Solution: The text is processed by a series of threads that are connected by pipes.
Each thread performs one step of the processing.

Note that the individual threads act on Readers and Writers and are unaware of the pipes that are connecting them.
Each can be used in other applications that need the particular behavior that it performs.

Thread One: ReadFile

- Constructor takes a String (file name) and a Writer.
- Characters read from the file are sent to the Writer.
- A final space is appended to the output stream so that the last word of the text is recognized.

```
import java.io.*;
class ReadFile extends Thread
{
    private String fileName;
    private FileReader fr;
    private Writer out;

    public ReadFile(String fn, Writer w)
    {   fileName  = fn;  out = w;      }

    public void run()
    {
        try
        {   fr = new FileReader(fileName);
            int ch = fr.read();
            while (ch != -1)
            {
                if (out!=null) out.write(ch);
                ch = fr.read();
            }
        }
        catch (IOException ioe)
        {  System.out.println("Error1:" + ioe); }
        finally
        {   try
            {   if (fr!=null) fr.close();
                if (out!=null)
                {  out.write(' ');  out.close();  }
            }
            catch (IOException ioe)
            {  System.out.println("Error2:" + ioe);  }
        }
    }
}       // End of ReadFile Thread
```

Thread Two: Tokenize

- Constructor takes a Reader and a Writer.
- Characters are read and tokenized as words using whitespace as the only delimiter.
- The individual words (Strings) are written to the output stream.

```
class Tokenize extends Thread
{
    private Reader in;
    private Writer out;
    private PrintWriter pw;

    public Tokenize(Reader r, Writer w)
    {
        in  = r;        out = w;
    }

    public void run()
    {
        boolean newWord = false;
        try
        {   pw = new PrintWriter(out);
            String word = "";
            int ch = in.read();
            while (ch != -1)
            {
                if  (Character.isWhitespace((char)ch))
                {   if (!newWord && pw!=null)
                            pw.println(word);
                    word = "";
                    newWord = true;
                }
                else if (newWord)
                {    word = word + (char)ch;
                     newWord = false;
                }
                else
                     word = word + (char)ch;
                ch = in.read();
            }
        }
        catch (IOException ioe)
        {  System.out.println("Error3:" + ioe);  }
        finally
        {    try
             {    if (in!=null) in.close();
                  if (out!=null) out.close();
             }
             catch (IOException ioe)
             {  System.out.println("Error4:" + ioe);  }
        }
    }
}        // End of Tokenize Thread
```

Thread Three: Format

- Constructor takes a Reader and a Writer.
- Read Strings (words) and write them, keeping track of the length of the line written.
- Insert newlines so that the lines have no more than 30 characters.

```
class Format extends Thread
{
    static final int WIDTH = 30;
    private Reader  in;
    private Writer out;
```

```
PrintWriter pw;

public Format(Reader r, Writer w)
{
     in  = r;   out = w;
}
public void run()
{
     try
     {    BufferedReader br = new BufferedReader(in);
          pw = new PrintWriter(out);
          int len = 0;
          String s = br.readLine();
          while (s!=null)
          {
               if (len + s.length() <= WIDTH)
               {
                    if (pw!=null) pw.print(s + " ");
                    len = len + s.length() + 1;
               }
               else
               {    if (pw!=null) pw.println();
                    if (pw!=null) pw.print(s + " ");
                    len = s.length() + 1;
               }
               s = br.readLine();
          }
     }
     catch  (IOException  ioe)
     {  System.out.println("Error5:" + ioe);  }
     finally
     {    try
          {    if (in!=null) in.close();
               if (out!=null) out.close();
          }
```

```
            catch (IOException ioe)
            { System.out.println("Error6:" + ioe); }
        }
    }
//      End of Format Thread
```

Thread Four: WriteFile

- Constructor takes a Reader and a String, a file name.
- Characters read from stream are written to the new file.

```
class WriteFile extends Thread
{
    private String fileName;
    private FileWriter fw;
    private Reader in;

    public WriteFile(Reader r, String fn)
    {
        fileName = fn; in = r;
    }

    public void run()
    {
        try
        {   fw = new FileWriter(fileName);
            int ch = in.read();
            while (ch != -1)
            {
                if (fw!=null) fw.write(ch);
                ch = in.read();
            }
        }
        catch (IOException ioe)
        { System.out.println("Error7:" + ioe); }
```

```
      finally
      {    try
           {    fw.write('\n');
                if (in!=null) in.close();
                if (fw!=null) fw.close();
           }
           catch (IOException ioe)
           {  System.out.println("Error8:" + ioe);  }
      }
   }
}          // End of WriteFile Thread
```

Main Program

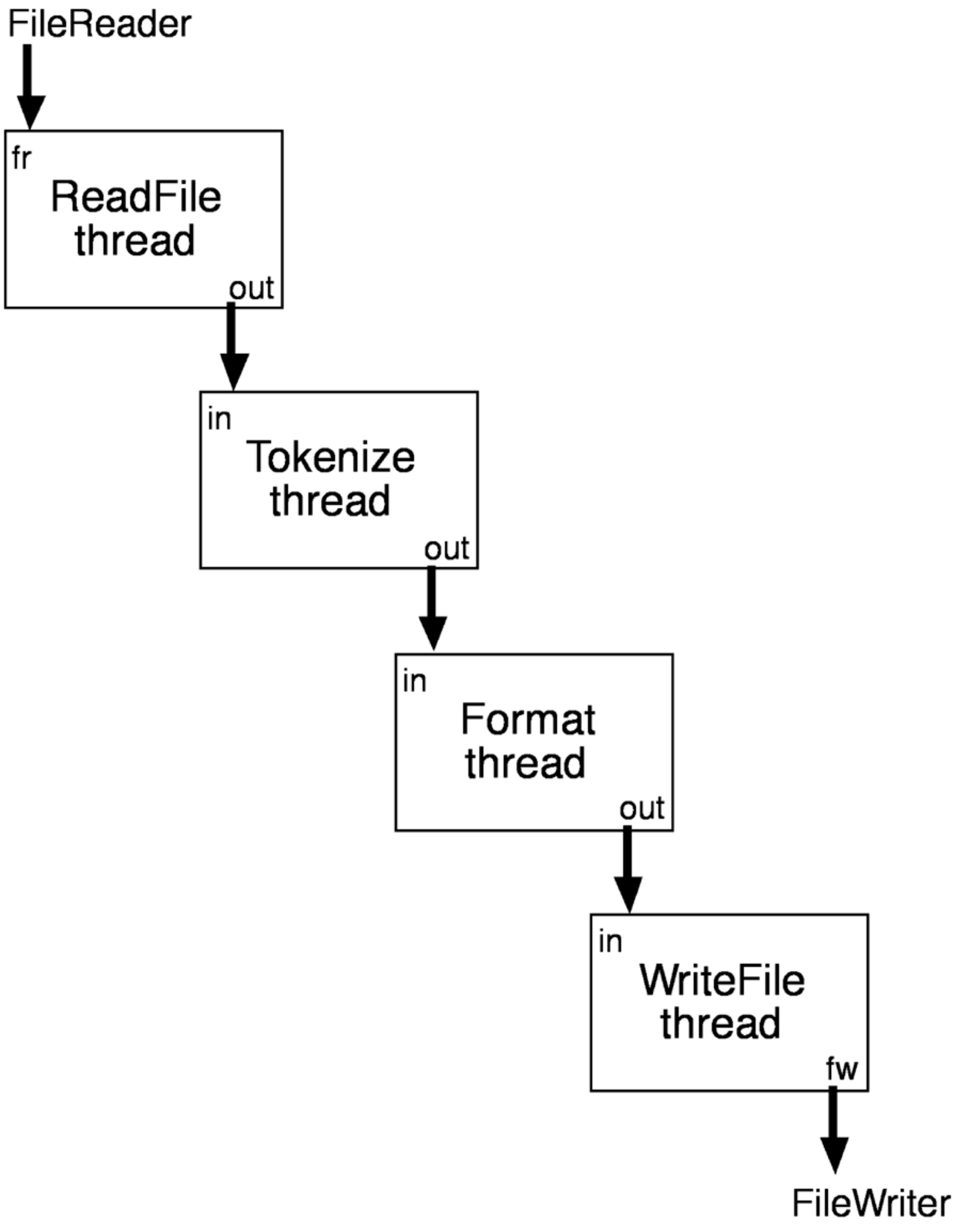
- Get names of source and target text files.
- Build three pipes.
- Create four threads and start them.
- Wait for threads to terminate.

```
public class CharStream
{
   public static void main(String [] args)
   {
      String inFile = "", outFile = "";
      try
      {  BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));

         System.out.print("Enter a source file name: ");
         inFile = br.readLine().trim();

         System.out.print("Enter a target file name: ");
         outFile = br.readLine().trim();

         PipedReader pin1 = new PipedReader();
```

```
        PipedWriter pout1 = new PipedWriter();
        pin1.connect(pout1);

        PipedReader pin2 = new PipedReader();
        PipedWriter pout2 = new PipedWriter();
        pin2.connect(pout2);

        PipedReader pin3 = new PipedReader();
        PipedWriter pout3 = new PipedWriter();
        pin3.connect(pout3);

        System.out.println("Formatting file " + inFile);

        ReadFile rf = new ReadFile(inFile, pout1);
        rf.start();

        Tokenize tk = new Tokenize(pin1, pout2);
        tk.start();

        Format fm = new Format(pin2, pout3);
        fm.start();

        WriteFile wf = new WriteFile(pin3, outFile);
        wf.start();

        try       // Wait for each thread to terminate
        {
           rf.join();   tk.join();   fm.join();   wf.join();
        }
        catch (InterruptedException ie)
        {  System.out.println("main interrupted!");  }

        System.out.println("Done.");
     }
     catch (FileNotFoundException fnfe)
     {  System.out.println("File not found: " + inFile);  }
```

```
         catch (IOException ioe)
         { System.out.println("Error9:" + ioe); }
      }
}         // End of CharStream
```

Overall View



Sample Execution

```
Enter a source file name: jeeves
Enter a target file name: jeeves.out
Formatting the file jeeves
Done.
```

File jeeves

```
Now, touching this  business of old Jeeves--my man,
you    know--how do we stand?
Lots of people  think I'm much too dependent on him.
My Aunt Agatha, in fact, has even gone so
far as to   call him my     keeper.
Well, what I say is: Why not?    The man's a genius.
From the collar upward he stands alone.
I gave       up trying to run my own affairs
within a week of his coming       to me.
```

File jeeves.out

```
Now, touching this business of
old Jeeves--my man, you
know--how do we stand? Lots of
people think I'm much too
dependent on him. My Aunt
Agatha, in fact, has even gone
so far as to call him my
keeper. Well, what I say is:
Why not? The man's a genius.
From the collar upward he
stands alone. I gave up trying
to run my own affairs within a
week of his coming to me.
```