The Java Series

IO, Serialization and Persistence

Input/Output

Often programs need to

- retrieve information from an external source.
- send information to an external source.

An external source is any resource accessible by the application we are writing:

- The file system: files and directories
- The network
- Another program
- The memory
- Any device

Input/Output

Java provides a basic framework in which to deal with I/O.

There is a set of conceptualizations and base classes on which I/O operations are built.

Once these conceptualizations are mastered we combine the base classes and build new ones to provide the functionality that we want.

This tutorial

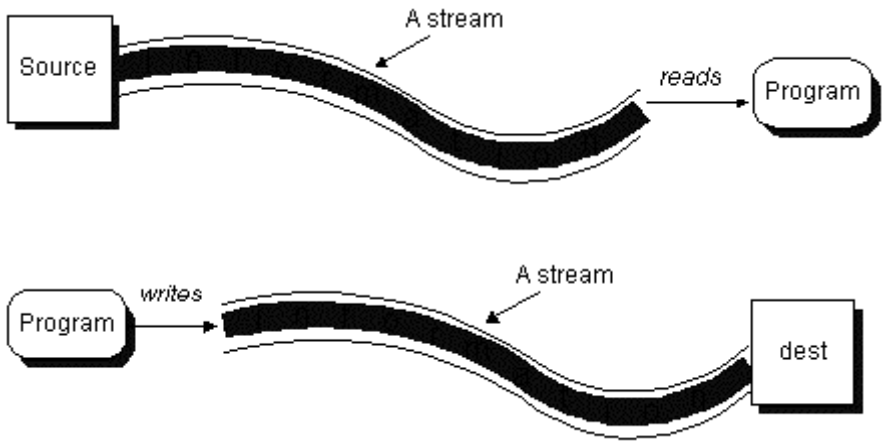
Will focus on the I/O conceptualizations and basic classes.

Will use examples reading and writing to files.

Is the basis for next tutorial focused on network programming.

The basic idea

A program reads/writes information from/to a channel. In Java, a channel from where a program may read or write information is referred to as a **STREAM**:



Streams

When reading:

No matter the source or the kind of information read any program reading from a stream:

```
build and open a stream
while there is information
      read information
close stream
```

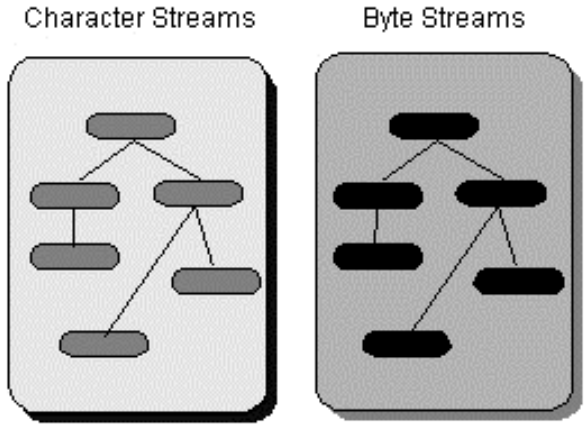
No matter the destination or the kind of information any program writing from a stream:

```
build and open a stream
while there is information to write
      write information
close stream
```

Streams

There are two kinds of Streams:

- Bytes Streams (classes named *Stream)
- Character Streams (classes named *Reader or *Writer)



A Character is two bytes. Provided from jdk 1.1 for internationalization and modular parsing.

Streams

Every Stream able to write to a data destination has a set of **write** methods.

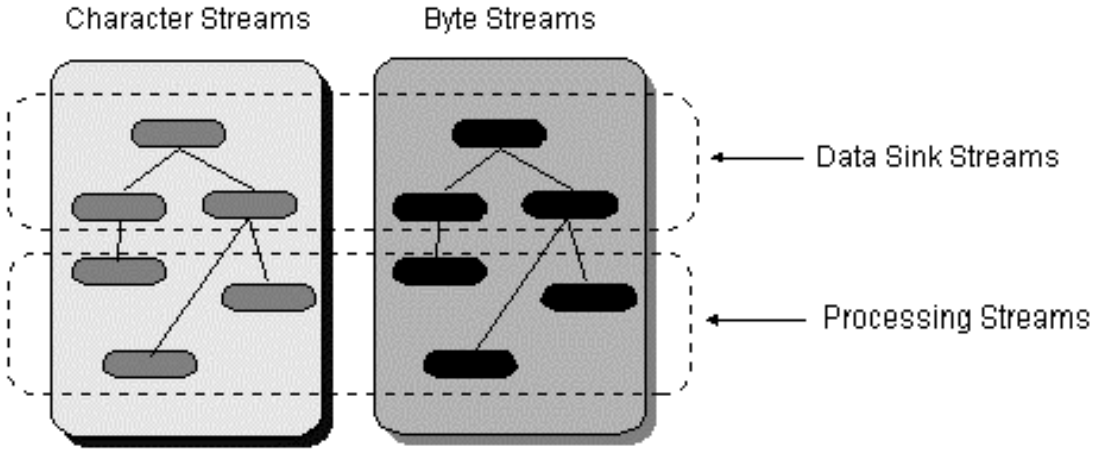
Every Stream able to read from a data source has a set of **read** methods.

Once the stream is set up we just have to invoke those methods.

Streams

But functionally we consider:

- Data sink streams: connected directly with the source or destination.
- Processing streams: connected to other streams to provide further processing transparently (filtering, compression, etc..)



Data Sink Streams

Are connected directly to the source or destination of the information. For instance, the following classes:

- FileReader: reads characters from a file.
- FileWriter: writes characters to a file.
- FileInputStream: reads bytes from a file.
- FileOutputStream: reads bytes from a file.

Data sink streams only have **simple** read/write methods

When creating Data Sink streams we have to specify the source

- For files: the filename
- For network: the socket
- From memory: the location

S1: Reading and Writing files

Character Streams version

```
import java.io.*;

public class Copy {
  public static void main(String[] args) throws IOException {
     File inputFile = new File("farrago.txt");
     File outputFile = new File("outagain.txt");

     FileReader in = new FileReader(inputFile);
     FileWriter out = new FileWriter(outputFile);


     while (in.ready() )
         out.write(in.read());

     in.close();
     out.close();
  }
}
```

Create the file objects

Create the character streams

Read from one stream and write into the other

S1: Reading and Writing files

Byte Streams version

```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);

        while (in.available() != 0)
           out.write(in.read());

        in.close();
        out.close();
    }
}
```

The principles are the same.
1. Create the streams
2. Read/Write to them

In this case the OS just reads bytes so both examples are equivalent

Processing Streams

Processing Streams provide further functionality when reading or writing other streams.

When creating a processing stream we have to specify what stream to connect it to.

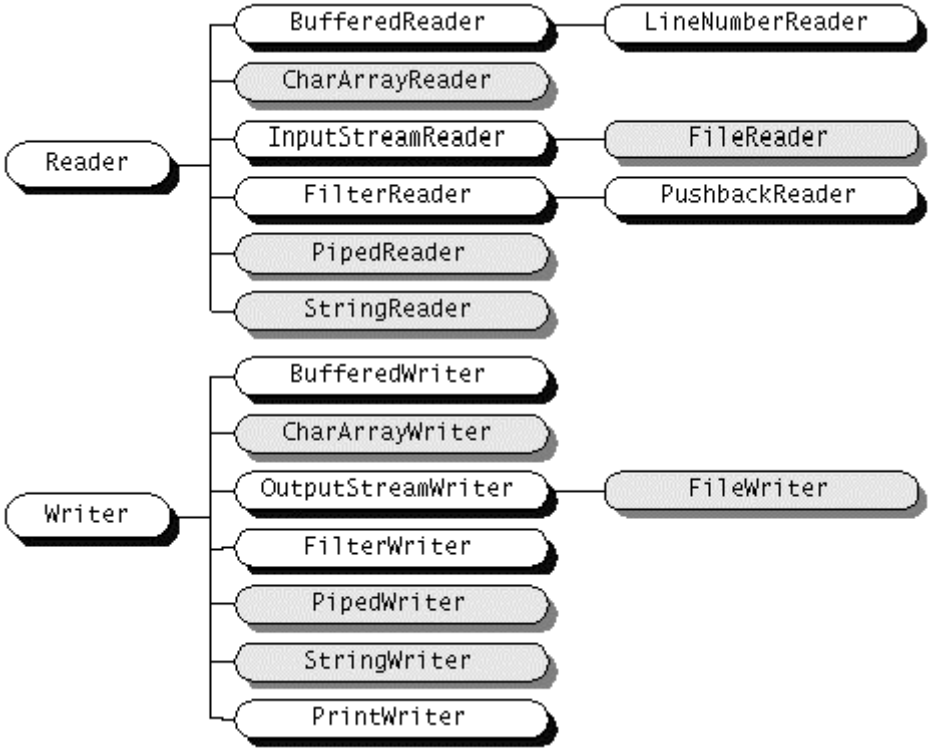
Some processing streams:

- DataInputStream, DataOutputStream, BufferedWriter, BufferedReader, etc...

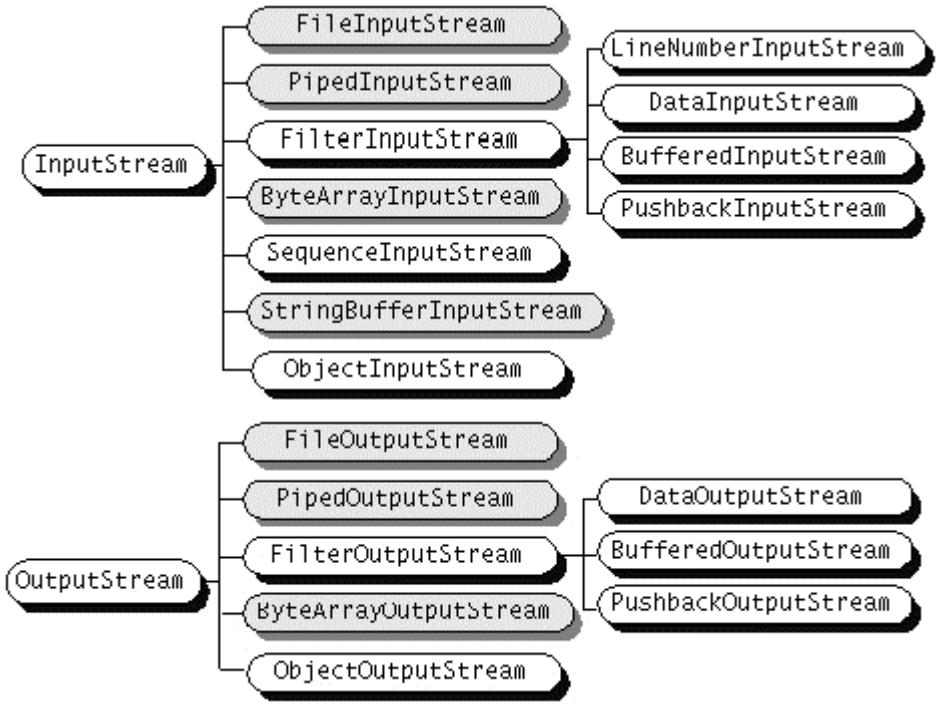
For instance:

- connect a DataStream to a FileStream to write read data types other than byte.
- connect a BufferStream to a NetworkStream to buffer network access.

Character Stream Classes



Byte Stream Classes



Streams Chains

Don t let the amount of classes scare you.

The key in working with streams is building the appropriate **STREAM CHAIN**.

A **stream chain** is:

- a chain of processing streams.
- one sink stream.

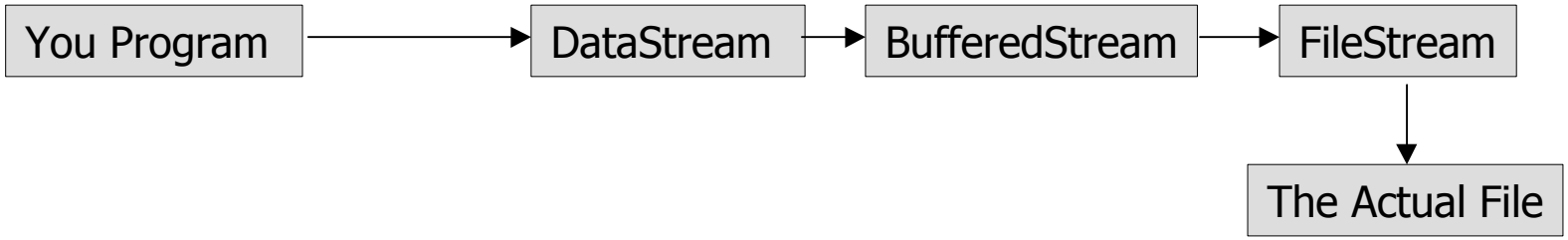
One end of the stream is the data sink stream.

You write/read information to the end of the chain.

The Java stream mechanism makes the information pass through all the streams in the chain. Each stream in the chain gets its chance to process info in an orderly manner.

Stream Chains

For instance:

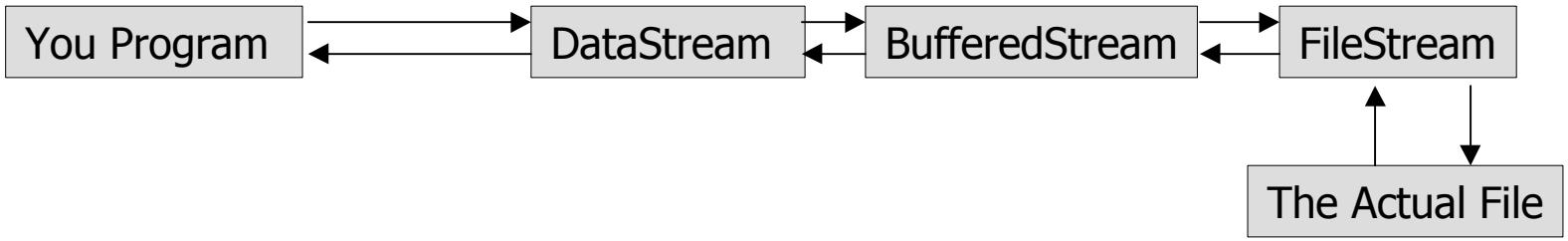


When everything is set up and you write a real to the data stream (invoking any write method):

- Your program asks the **data stream** to write something
- The **data stream** obtains a set of bytes and writes them to the **buffer stream**.
- The **buffer stream** retains bytes until it decides to write them all together to the **file stream**.
- The **file stream** effectively writes the bytes to the file whenever received from the buffer stream

Stream Chains

For instance:



When reading (using the write method):

- Your program asks the data stream to read a real number.
- The data streams asks the buffer stream to read a number of bytes corresponding to the length of a real number.
- The buffer stream asks the file stream to read some more bytes so that they are buffered for the next read.
- The file stream actually reads the bytes
- The data is passed back and interpreted by each stream

SC2: The First Chain: Writing

```
import java.io.*;

public class DataOutTest {
    public static void main(String[] args) throws IOException {

        // write the data out
        DataOutputStream out = new DataOutputStream(new
                                   FileOutputStream("invoice1.txt"));

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

        for (int i = 0; i < prices.length; i ++) {
            out.writeDouble(prices[i]);
            out.writeChar('\t');
            out.writeInt(units[i]);
            out.writeChar('\t');
            out.writeChars(descs[i]);
            out.writeChar('\n');
        }
        out.close();
    }
}
```

We build the Stream Chain. In this case just a DataStream and a FileStream

Create some data (arrays,) and loop through it

We always write to the first stream in the chain. Here it is the DataStream.

writeDouble, etc. are methods of the DataOutputStream class.

SC2: The First Chain: Reading

```
import java.io.*;
public class DataInTest {
    public static void main(String[] args) throws IOException {

        // read it in again
        DataInputStream in = new DataInputStream(new
                                FileInputStream("invoice1.txt"));
        double price;
        int unit;
        String desc;
        double total = 0.0;
        try {
            while (true) {
                price = in.readDouble();
                in.readChar();       // throws out the tab
                unit = in.readInt();
                in.readChar();       // throws out the tab
                desc = in.readLine();
                System.out.println("You've ordered " +
                                   unit + " units of " +
                                   desc + " at $" + price);
                total = total + unit * price;
            }
        } catch (EOFException e) {}
        System.out.println("For a TOTAL of: $" + total);
        in.close();
    }
}
```

We build the Stream Chain. In this case we are reading

Notice the EOF exception

We read from the first stream of the chain.
readChar, etc.. are methods of the DataInputStream class

SC3: A byte chain

```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        BufferedInputStream in =
                new BufferedInputStream(
                     new FileInputStream(inputFile));
        BufferedOutputStream out =
                new BufferedOutputStream (
                     FileOutputStream(outputFile));


        while (in.available() != 0)
           out.write(c);

        in.close();
        out.close();
    }
}
```

We build the read and write chains.

Note that we decide to buffer input AND output

As always, we interact with the first stream of either chain.

SC3: A character Chain

```
import java.io.*;

public class Copy {
  public static void main(String[] args) throws IOException {
     File inputFile = new File("farrago.txt");
     File outputFile = new File("outagain.txt");

     BufferedReader in =
               new BufferedReader (
                    new FileReader(inputFile));
     BufferedWriter out =
               new BufferedWriter (
                    new FileWriter(outputFile));
     int c;

     while (in.ready())
         out.write(c);

     in.close();
     out.close();
  }
}
```

We build the read and write chains. In this case we use the character oriented classes .

As always, we interact with the first stream of either chain.

More Chains

Remember, the key is building the correct Stream Chain so that you get what you want.

Another example: There are two classes in java.io which is **GZIPInputStream** and **GZIPOutputStream**.

They are just **processing streams**. To use them just include them appropriately in your stream chains.

- Buffer to GZIP to File: First buffer then compress.
- GZIP to Buffer to File: First compress then buffer.
- Data to Buffer to GZIP to Network: Compressed Buffered data to network (the other end must understand it as well).

SC4: Compressing a file

```
import java.io.*;
import java.util.zip.*;

public class GZIPcompress {

  public static void main(String args[]) {
    try {
        BufferedInputStream in =
          new BufferedInputStream (
            new FileInputStream (args[0]));

        BufferedOutputStream out =
          new BufferedOutputStream (
             new GZIPOutputStream (
                new FileOutputStream(args[0]+".gz")));

        while ( in.available()!=0)
           out.write(in.read());

        in.close();
        out.close();
     } catch (IOException e) {
        e.printStackTrace();
     }
  }
}
```

We read a regular file, so we build a buffered stream chain to read it.

We write a compressed file, so we add the GZIP stream to the chain

As always, we interact with the first stream of either chain.

The chain setup guarantees that it will work as expected.

SC4: Uncompressing a file

```
import java.io.*;
import java.util.zip.*;

public class GZIPuncompress {

  public static void main(String args[]) {
    try {
        BufferedOutputStream out =
          new BufferedOutputStream (
            new FileOutputStream ("test"));

        BufferedInputStream in =
          new BufferedInputStream (
             new GZIPInputStream (
                new FileInputStream(args[0])));

        while ( in.available()!=0)
           out.write(in.read());

        in.close();
        out.close();
     } catch (IOException e) {
        e.printStackTrace();
     }
  }
}
```

We write a regular file, so we build a buffered stream chain to write it.

We read a compressed file, so we add the GZIP stream to the chain

We read from the end of the reading chain and write to the end of the writing chain

Stream Classes

As you probably have noticed, for every stream type there is an InputStream (byte) or Reader (char) and an OutputStream (byte) or Reader (char).

Stream Classes can be subclassed. This is, you can create your own streams to be included in a chain.

We just have to:

- create the constructor accepting a Stream.
- comply with the read/write methods and implement our functionality there.

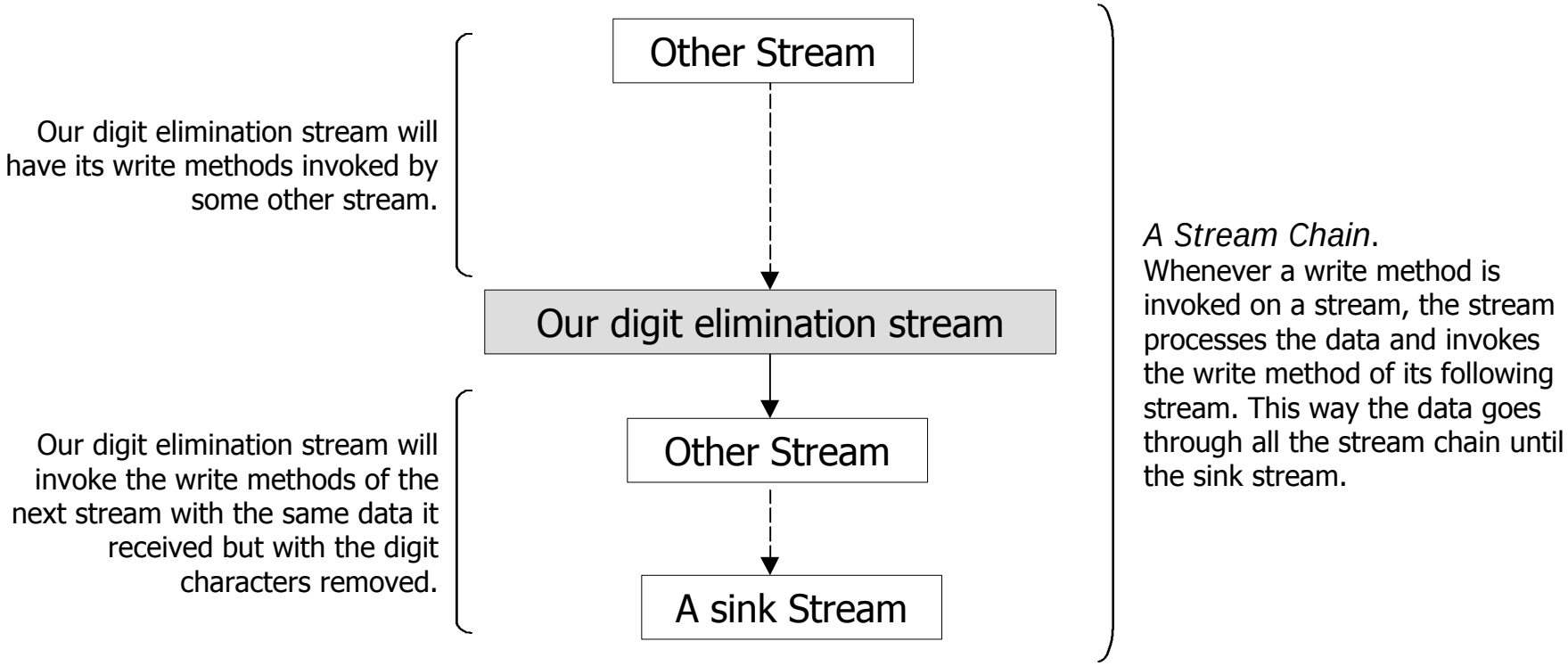
Creating your processing stream

The base classes from which to derive your own processing streams are **FilterOutputStream**, **FilterInputStream**, **FilterReader** and **FilterWriter**.

Our scenario: We want to create a stream which will eliminate any digit character written into it. The steps to take are:

- Create a class derived from FilterOutputStream.
- Create a constructor which taking the stream to which our stream is going to be connected to along a chain.
- Create the write methods writing to the stream passed along the constructor.

A Digit Elimination Stream



There are typically three write methods to implement.
For a byte, an array of bytes and a part of an array of bytes.

SC5: Digit Elimination Stream

```
import java.io.*;

public class DigitFilterOutputStream extends FilterOutputStream {

    public DigitFilterOutputStream (OutputStream out) {
        super(out);
    }

    public void write(int b) throws IOException {
        if (b<48 || b>57) out.write(b);
    }

    public void write(byte[] b) throws IOException {
        for (int i=0; i<b.length; i++)
             write(b[i]);
    }

    public void write(byte[] b, int off, int len) throws IOException {
        for (int i=off; i<len; i++)
             write(b[i]);
    }
}
```

This is an output processing stream, so we derive it from FilterOuputStream.

In the constructor we accept the stream to connect to

Implement write methods so that we write on the stream passed along in the constructor

SC5: Using our Stream (1)

```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        FileInputStream in = new FileInputStream(inputFile);
        DigitFilterOutputStream out =
              new DigitFilterOutputStream(
                 new FileOutputStream(outputFile));

        while ( in.available()!=0 )
           out.write(c);

        in.close();
        out.close();
    }
}
```

We just include the stream we defined in any stream chain

And the rest works just as with any other stream

SC5: Using our stream (2)

```
import java.io.*;
import java.util.zip.*;
public class GZIPcompress {
  public static void main(String args[]) {
    try {
        BufferedInputStream in =
          new BufferedInputStream (
            new FileInputStream (args[0]));

        BufferedOutputStream out =
          new BufferedOutputStream (
            new DigitFilterOutputStream (
              new GZIPOutputStream (
                new FileOutputStream(args[0]+".gz"))));

        while ( in.available()!=0)
           out.write(in.read());

        in.close();
        out.close();
     } catch (IOException e) {
        e.printStackTrace();
     }
  }
}
```

Again, we just include our stream in the stream chain, wherever we want.

This Stream Chain.
In this example, the data is:
(1) Buffered.
(2) Filtered.
(3) Compressed.
(4) Written into the file.

Think about this Chain

In the previous example the only write method that we invoke is the one on BufferedOutputStream, so that data is buffered according to its implementation.

Then, the BufferedOutputStream invokes the write method of our DigitFilterOutputStream, so that digit characters are left out.

Then, the DigitFilterOutputStream invokes the write method of the GZIPOutputStream, so data is compressed.

Then, GZIPOutputStream invokes the write method of FileOutputStream which effectively writes the file.

Think about chains

What happens if the order is:

Buffer → GZIP → DigitFilter → File ?

The numbers will be filtered AFTER the compression takes place.

Our gzipped file will have an incorrect format **BUT**

The chain construction is valid.

The same principle goes with reading chains when writing input streams or readers. There is a chain of **read** methods which are invoked.

SC6: Checksum Output

```
import java.io.*;

public class CheckedOutputStream extends FilterOutputStream {
    private Checksum cksum;

    public CheckedOutputStream(OutputStream out, Checksum cksum) {
        super(out);
        this.cksum = cksum;
    }

    public void write(int b) throws IOException {
        out.write(b);
        cksum.update(b);
    }

    public void write(byte[] b) throws IOException {
        out.write(b, 0, b.length);
        cksum.update(b, 0, b.length);
    }

    public void write(byte[] b, int off, int len) throws IOException {
        out.write(b, off, len);
        cksum.update(b, off, len);
    }

    public Checksum getChecksum() {
        return cksum;
    }
}
```

Create the stream with the stream it is connected to

Implement the write methods with the desired functionality

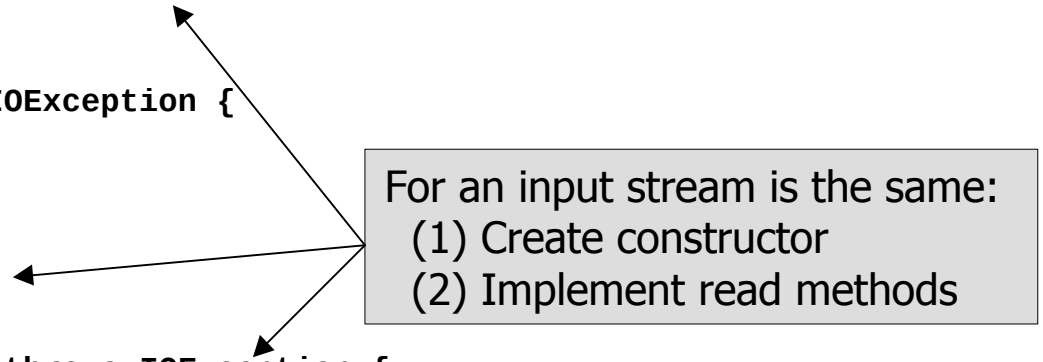
SC6: Checksum Input

```
import java.io.InputStream;
import java.io.IOException;

public class CheckedInputStream extends FilterInputStream {
    private Checksum cksum;

    public CheckedInputStream(InputStream in, Checksum cksum) {
        super(in);
        this.cksum = cksum;
    }
    public int read() throws IOException {
        int b = in.read();
        if (b != -1) {
            cksum.update(b);
        }
        return b;
    }
    public int read(byte[] b) throws IOException {
        int len;
        len = in.read(b, 0, b.length);
        if (len != -1) {
            cksum.update(b, 0, len);
        }
        return len;
    }

    . . . .
```



SC6: Checksum Usage

```
try {
          in = new CheckedInputStream(
                          new FileInputStream("farrago.txt"),
                          inChecker);
          out = new CheckedOutputStream(
                           new FileOutputStream("outagain.txt"),
                           outChecker);
      } catch (FileNotFoundException e) {
          System.err.println("CheckedIOTest: " + e);
          System.exit(-1);
      } catch (IOException e) {
          System.err.println("CheckedIOTest: " + e);
          System.exit(-1);
      }

      while ( in.available() != 0)
         out.write(in.read());

      System.out.println("Input stream check sum: " +
                         inChecker.getValue());
      System.out.println("Output stream check sum: " +
                         outChecker.getValue());

      in.close();
      out.close();
} catch ( . . . . . . .
```

Create the input and output chains

Write/read at the top of the chains

Use extended functionality

Object Persistence

Object persistence refers to the possibility of objects living across application runs.

Persistence means having an object s life independent from the life time of the application in which it is running.

One way to implement persistence is storing objects and then retrieving them.

Object Persistence and Streams

Using Java s conceptualizations of streams we d like to be able to write/read objects through a stream.

Then connect the objects stream as any other stream in a chain:

- If we connect it to a file stream: store objects on the file system
- If we connect it to a network stream: send objects through the network.

This is very powerful and flexible!!!

To make it work: Serialization

There are two classes implementing processing streams:

ObjectInputStream, ObjectOutputStream

Classes must be tagged so that instances of those classes maybe written/read through an object stream.

The process of sending an object through a stream is referred to as **SERIALIZATION.**

The ObjectStream classes implement serialization and deserialization of objects.

SC7: A Sample Class

```
import java.io.*;

public class BunchOfRandoms implements Serializable {

    // Generate a random value
    private static int r() {
        return (int)(Math.random() * 10);
    }

    private int data[];

    // Constructor
    public BunchOfRandoms() {
        data = new int[r()];
        for (int i=0; i<data.length; i++)
             data[i]=r();
    }

    public void printout() {
        System.out.println("This BunchOfRandoms has "+data.length+" random ints"
);
        for (int i=0; i<data.length; i++)
              System.out.print(data[i]+":");

        System.out.println();
    }
}
```

Just tag a class as serializable. Serializable is an empty interface just to indicate Java that objects instantiated from this class can be sent through an object stream.

This class produces random numbers when created

SC7: Serializing objects out

```
import java.io.*;
import java.util.*;

public class SerializeOut {

   public static void main (String args[]) throws IOException  {

       BunchOfRandoms b1 = new BunchOfRandoms();
       BunchOfRandoms b2 = new BunchOfRandoms();

       ObjectOutputStream out =
           new ObjectOutputStream (
              new FileOutputStream("objects.dat"));

       Date now = new Date(System.currentTimeMillis());
       out.writeObject(now);
       out.writeObject(b1);
       out.writeObject(b2);
       out.close();
       System.out.println("I have written:");
       System.out.println("A Date object: "+now);
       System.out.println("Two bunchs of randoms");
       b1.printout();
       b2.printout();
   }
}
```

Just create a stream chain with an object stream at the top

And write objects through the stream. Here we are writing three objects.

SC7: Serializing Objects in

```
import java.io.*;
import java.util.*;

public class SerializeIn {

   public static void main (String args[]) throws
               IOException, ClassNotFoundException  {

       ObjectInputStream in =
           new ObjectInputStream (
              new FileInputStream("objects.dat"));

       Date d1 = (Date)in.readObject();
       BunchOfRandoms br1 = (BunchOfRandoms)in.readObject();
       BunchOfRandoms br2 = (BunchOfRandoms)in.readObject();

       System.out.println("I have read:");
       System.out.println("A Date object: "+d1);
       System.out.println("Two bunchs of randoms");
       br1.printout();
       br2.printout();
   }
}
```

Create the stream chain

Read objects in the same order

Serialization

When writing an object the ObjectOutputStream serializes as well all objects referred to by it.

A BunchOfRandoms has an array of Integers.

To serialize a BunchOfRandoms we don t have to loop through its inner objects.

The serialization mechanism takes care of it:

- When writing writes all objects referred to.
- When reading reconstructs all and the references.

Serializing standard classes

Most classes in java.* are serializable.

This includes AWT classes.

Remember:

- A Frame is a window which contains a set of graphical components.
- If we serialize a Frame, the serialization mechanism will automatically serialize all its components and data (position, contents, etc.)

SC8: Serializing out AWT

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class WindowSerializeOut
        implements ActionListener {

  Frame f;

  public void actionPerformed (ActionEvent ev) {
    try {
       ObjectOutputStream out =
         new ObjectOutputStream (
           new FileOutputStream ("status.dat"));
       out.writeObject(f);
       out.close();
       System.out.println("Window serialized");
    } catch (IOException e) {
       e.printStackTrace();
    }
  }
```

Serialization happens when clicking

When writing the frame everything is written by the serialization mechanism

```
  public void createFrame () {
     f = new Frame();
     f.setLayout (new FlowLayout());
     f.setLocation(100,100);

     f.add (new Button("A button"));
     Button b = new Button ("Serialize Frame")
     f.add(b);
     b.addActionListener(this);
     f.add (
        new TextField("This is a textfield"));
     f.pack();
     f.show();

  }

  public static void main (String args[]) {
     WindowSerializeOut w =
           new WindowSerializeOut();
     w.createFrame();
  }
}
```

Create a frame and some components

SC8: Serializing in AWT

```
import java.awt.*;
import java.io.*;

public class WindowSerializeIn {

 public static void main (String args[])
     throws IOException, ClassNotFoundException{

     ObjectInputStream in =
        new ObjectInputStream (
          new FileInputStream ("status.dat"));

     Frame f = (Frame)in.readObject();
     in.close();
     System.out.println("Window retrieved");
     f.show();

 }
}
```

Open the stream chain

Reading the frame automatically retrieves all objects referred to by the frame

Security in Serialization

Declaring a class as serializable means sending ALL its status through an stream.

Sometimes we may want to restrict the data of an object that can be serialized.

Two ways to do it:

- By default everything is serialized and you have to declare explicitly as **transient** the data you DON T want to serialize.
- By default nothing is serialized and you have to declare explicitly the data you DO want to serialize.

SC9: Transient data

```
import java.io.*;

public class Login implements Serializable {

     private String user;
     private transient String password;

     public Login() {}

     public Login (String u, String p) {
          user = u;
          password = p;
     }

     public void printout () {
         if (user != null)
               System.out.println ("User name is: "+user);
         else
               System.out.println ("User name not defined");

         if (password != null)
               System.out.println ("Password is: "+password);
         else
               System.out.println ("Password not defined");
     }
}
```

A serializable object

One transient variable

SC9: Transient data

```
import java.io.*;

public class MyApplication {

     public static void main (String[] args)
                    throws IOException, ClassNotFoundException {

         Login l1 = new Login("teacher", "class99java");
         System.out.println ("This is the object created in memory");
         l1.printout();
         ObjectOutputStream out =
            new ObjectOutputStream (
             new FileOutputStream ("login.dat"));
         out.writeObject(l1);
         out.close();

         // We crate another login object from the stored one
         ObjectInputStream in =
            new ObjectInputStream (
             new FileInputStream ("login.dat"));

         Login l2 = (Login)in.readObject();
         System.out.println("This is the object created from serialization");
         l2.printout();
         in.close();
     }
}
```

The object is written out. The serialization mechanism takes care of not writing transient data

When retrieving the object the transient data is not restored

Method 2:

To explicitly declare the data we want to serialize:

- Tag the object as Externalizable
- Implement the writeExternal and readExternal methods to explicitly write and read your data.

Bear in mind:

- The serialization mechanism calls the default constructor (without arguments) when restoring externalizable objects.

SC10: Externalizable objects

A class with two variables, implementind the Externalizable interface

Default constructor

Customized constructor

In the readExternal and writeExternal methods we explicitly deal only with the variable we want to store

```
import java.io.*;
public class Blip implements Externalizable {
   int i;   String s;
   public Blip() {
     System.out.println("Blip's default constructor");
     i=0; s="";
   }
   public Blip (String x, int a) {
     System.out.println("Blip's constructor with data");
     s=x;i=a;
   }
   public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
     s = (String)in.readObject();
   }
   public void writeExternal (ObjectOutput out)
      throws IOException {
     out.writeObject(s);
   }
   public void printout() {
      System.out.println ("This blip has an int: "+i);
      System.out.println ("This blip has a string: "+s);
   }

}
```

S10: Externalizable

```
import java.io.*;

public class SerializeOutBlips {

    public static void main (String args[]) throws IOException {

        Blip b = new Blip ("Hello class", 21);
        b.printout();

        ObjectOutputStream out =
           new ObjectOutputStream (
            new FileOutputStream ("blips.dat"));

        out.writeObject(b);
        out.close();
    }
}
```

Create Blip

Write blip out. Note that we don t have to know anything about its behavior (serializable, transient, externalizable)

S10: Externalizable

```
import java.io.*;

public class SerializeInBlips {

    public static void main (String args[])
       throws IOException, ClassNotFoundException {

         ObjectInputStream in =
            new ObjectInputStream (
             new FileInputStream ("blips.dat"));

         Blip b = (Blip)in.readObject();
         b.printout();
         in.close();
    }
}
```

Read Blip back, just as any other serializable object. Transparently to SerializeInBlips, the Blip s default constructor is called automatically.

SUMMARY

We have seen the Java STREAM mechanism and class organization.

We have designed and build Stream Chains.

We have made our own Streams and included them in other Stream Chains.

We have Serialized objects.

We have controlled the serialization process through the transient and externalizable mechanisms.