Java Programming Course
IO

Session objectives

- What is an I/O stream?
- Types of Streams
- Stream class hierarchy
- Control flow of an I/O operation using Streams
- Byte streams
- Character streams
- Buffered streams
- Standard I/O streams
- Data streams
- Object streams
- File class

I/O stream

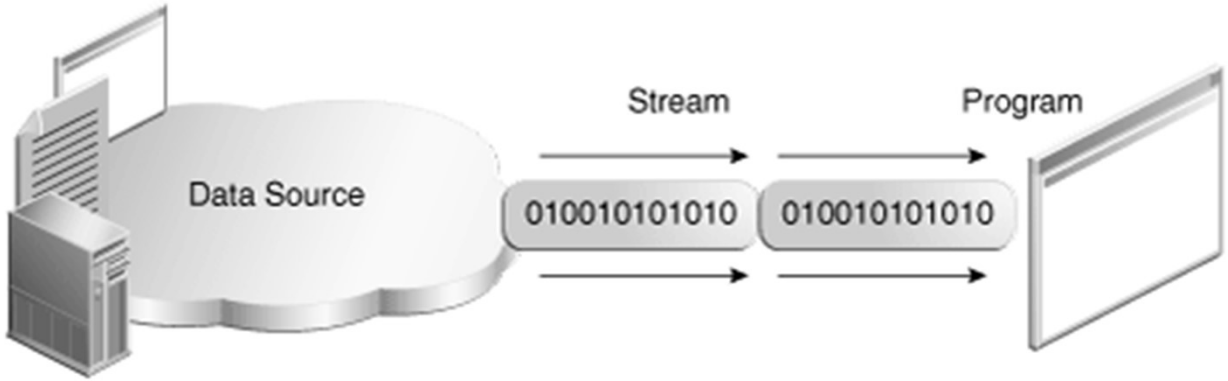
I/O Streams

- 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
  - simple bytes, primitive data types, localized characters, and objects.
- Some streams simply pass on data; others manipulate and transform the data in useful ways.

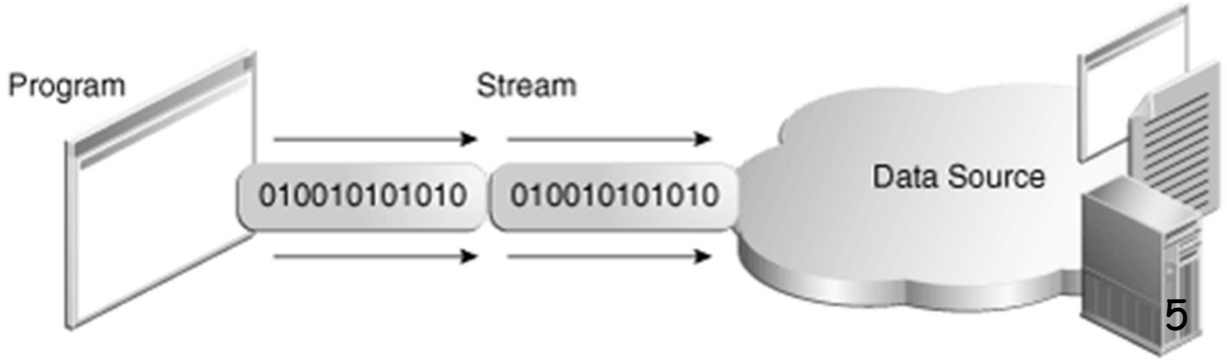
I/O Streams

A stream is a sequence of data.

- A program uses an ***input stream*** to read data from a source, one item at a time:



- A program uses an ***output stream*** to write data to a destination, one item at time:



Types of Streams

General Stream Types

- Character and Byte Streams
- Input and Output Streams
- Based on source or destination
- Node and Filter Streams
- Whether the data on a stream is manipulated or transformed or not.

Character and Byte Streams

- Byte streams
  - For binary data
  - Root classes for byte streams:
    - The InputStream Class
    - The OutputStream Class
    - Both classes are abstract
- Character streams
  - For Unicode characters
  - Root classes for character streams:
    - The Reader class
    - The Writer class
    - Both classes are abstract

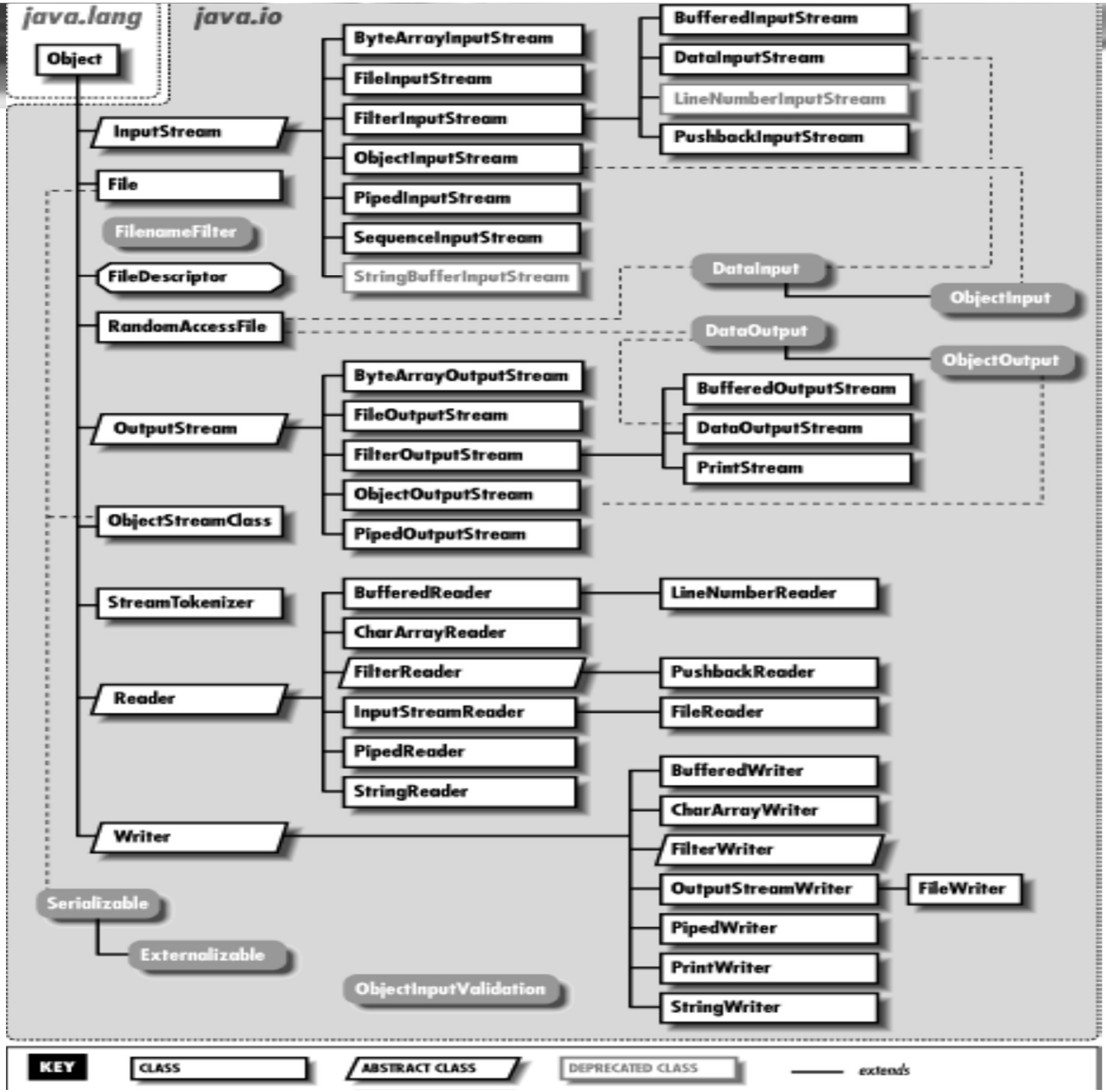
Input and Output Streams

- Input or source streams
  - Can read from these streams
  - Root classes of all input streams:
    - The InputStream Class
    - The Reader Class
- Output or sink (destination) streams
  - Can write to these streams
  - Root classes of all output streams:
    - The OutputStream Class
    - The Writer Class

Node and Filter Streams

- Node streams (Data sink stream)
  - Contain the basic functionality of reading or writing from a specific location
  - Types of node streams include files, memory and pipes
- Filter streams (Processing stream)
  - Layered onto node streams between threads or processes
  - For additional functionality- altering or managing data in the stream
- Adding layers to a node stream is called stream chaining

Java IO Stream hierarchy



Control Flow of an I/O operation

1. Create a stream object and associate it with a data-source (data-destination)
2. Give the stream object the desired functionality through stream chaining
3. while (there is more information)
4. read(write) next data from(to) the stream
5. close the stream

Byte Stream

Byte Stream

- 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
  - FileInputStream and FileOutputStream
- They are used in much the same way; they differ mainly in the way they are constructed

When Not to Use Byte Streams?

- Byte Stream represents a kind of low-level I/O that you should avoid
  - If the data 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
- All other streams are based on byte stream

Byte Stream example

```
3  import java.io.FileInputStream;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  public class CopyBytes {
7      public static void main(String[] args) throws IOException {
8          FileInputStream in = null;
9          FileOutputStream out = null;
10         try {
11             in = new FileInputStream("data.txt");
12             out = new FileOutputStream("copy_data.txt");
13             int c;
14             while ((c = in.read()) != -1) {
15                 out.write(c);
16             }
17         } finally {
18             if (in != null) in.close();
19             if (out != null)out.close();
20         }
21     }
22 }
```

Character Stream

Character Stream

- 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.
- 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.

Character Stream

- 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.
  - A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.
  - If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues.
  - Later, if internationalization becomes a priority, your program can be adapted without extensive recoding.

Character Stream example

```
3  import java.io.FileReader;
4  import java.io.FileWriter;
5  import java.io.IOException;
6
7  public class CopyCharacters {
8      public static void main(String[] args) throws IOException{
9          FileReader inputStream = null;
10         FileWriter outputStream = null;
11         try {
12             inputStream = new FileReader("data.txt");
13             outputStream = new FileWriter("dataoutput.txt");
14             int c;
15             while ((c = inputStream.read()) != -1) {
16                 outputStream.write(c);
17             }
18         } finally {
19             if (inputStream != null) inputStream.close();
20             if (outputStream != null) outputStream.close();
21         }
22     }
23 }
```

Character Stream and Byte Stream

- 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.

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").

Line-Oriented I/O example

```
3  import java.io.FileReader;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.io.PrintWriter;
7  import java.util.Scanner;
8
9  public class CopyLines {
10     public static void main(String[] args) throws IOException {
11         PrintWriter outputStream = null;
12         Scanner input=null;
13         try {
14             input = new Scanner(new FileReader("data.txt"));
15             outputStream = new PrintWriter(new FileWriter("dataoutput.txt"));
16             String line;
17             while (input.hasNextLine()) {
18                 line=input.nextLine();
19                 outputStream.println(line);
20             }
21         } finally {
22             if (input != null) input.close();
23             if (outputStream != null) outputStream.close();
24         }
25     }
26 }
```

Buffered Stream

The necessary of Buffered Streams

- An unbuffered I/O 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; 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 Stream Classes

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

Buffered Stream Classes - Reading demo

```
11  public void readFromFile(String filename) throws Exception{
12      BufferedInputStream bufferedInput = null;
13      byte[] buffer = new byte[1024];//buffer size
14      try {
15          //Construct the BufferedInputStream object
16          bufferedInput = new BufferedInputStream(new FileInputStream(filename));
17          int bytesRead = 0;
18          //Keep reading from the file while there is any content
19          //when the end of the stream has been reached, -1 is returned
20          while ((bytesRead = bufferedInput.read(buffer)) != -1) {
21              //Process the chunk of bytes read
22              //in this case we just construct a String and print it out
23              String chunk = new String(buffer, 0, bytesRead);
24              System.out.print(chunk);
25          }
26      } catch (Exception ex) {
27          ex.printStackTrace();
28      } finally {
29          //Close the BufferedInputStream
30              if (bufferedInput != null)
31                  bufferedInput.close();
32      }
33  }
```

Buffered Stream Classes - Writing demo

```
39 public void writeBuffered(String filename)throws Exception{
40     BufferedOutputStream output=null;
41     try {
42         output=new BufferedOutputStream(new FileOutputStream(filename));
43         String line="this is a sample text";
44         byte[]b=line.getBytes();
45         output.write(b, 3/*offset*/, 6/*length*/);
46         //result will be "s is a"
47     } finally{
48         if(output!=null)
49             output.close();
50     }
51 }
```

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.

Standard Streams

Standard Streams on Java Platform

- 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
- System.out and System.err are defined as PrintStream objects

Data Streams

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
- *DataInputStream* and *DataOutputStream* are most widely-used implementations of these interfaces

The DataOutputStream class

- A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.
- Constructor:

  DataOutputStream(OutputStream out)

  Creates a new data output stream to write data to the specified underlying output stream.

DataOutputStream demo

```
3  import java.io.BufferedInputStream;
4  import java.io.BufferedOutputStream;
5  import java.io.DataInputStream;
6  import java.io.DataOutputStream;
7  import java.io.FileInputStream;
8  import java.io.FileOutputStream;
9
10 public class DemoDataStream {
11     static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
12     static final int[] units = { 12, 8, 13, 29, 50 };
13     static final String[] descs = {
14         "Java T-shirt","Java Mug","Duke Juggling Dolls",
15         "Java Việt Nam","Java Key Chain"
16     };
17     public void write(String dataFile)throws Exception{
18         DataOutputStream out=new DataOutputStream(
19                 new BufferedOutputStream(new FileOutputStream(dataFile)));
20         for (int i = 0; i < prices.length; i ++) {
21             out.writeDouble(prices[i]);
22             out.writeInt(units[i]);
23             out.writeUTF(descs[i]);
24         }
25         out.close();
26     }
```

dataFile.data

```
1 @3♦p♦♦
2 =    Java T-shirt@#♦♦G♦{
3  Duke Juggling Dolls@♦♦Q♦
```

The DataInputStream class

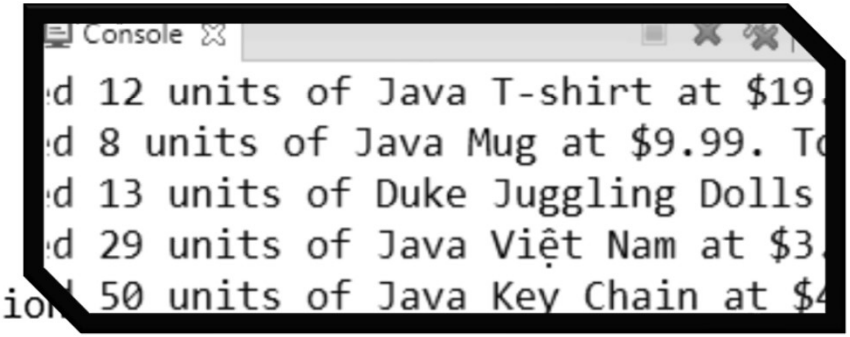
- A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.
- Constructor :

```
DataInputStream(InputStream in)
```

Creates a DataInputStream that uses the specified underlying InputStream.

The DataInputStream exemple

```
28  public void read(String dataFile)throws Exception{
29      DataInputStream in = new DataInputStream(new
30              BufferedInputStream(new FileInputStream(dataFile)));
31      double total = 0.0;
32      while (in.available()!=0) {
33          double price = in.readDouble();
34          int unit = in.readInt();
35          String desc = in.readUTF();
36          total += unit * price;
37          System.out.format("You ordered %d" + " units of %s at $%.2f."
38                  +" Total: $%.2f.%n",unit, desc, price,total);
39      }
40      in.close();
41  }
```




```
public static void main(String[] args) throws Exception
    new DemoDataStream().write("dataFile.data");
    new DemoDataStream().read("dataFile.data");
}
```

Object Streams

Object Serialization

- Persistence is the concept that an object can exist separate from the executing program that creates it
- Java contains a mechanism called object serialization for creating persistent objects executing program that creates it.
- When an object is serialized, it is transformed into a sequence of bytes; this sequence is raw binary representation of the object. Later, this representation can be restored to the original object. Once serialized, the object can be stored in a file for later use.

Object Streams

- Object streams support I/O of objects
  - Like Data streams support I/O of primitive data types
  - The object has to be Serializable type
- The object stream classes are ObjectInputStream and ObjectOutputStream
  - These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput
  - An object stream can contain a mixture of primitive and object values

Object Serialization

- Any object we want to serialize must implement the Serializable interface.
- To *serialize* an object, we invoke the `writeObject` method of an ObjectOutputStream.
- To *deserialize* the object, we invoke the `readObject` method of an ObjectInputStream.
- The actual data streams to which the serialized object is written can represent a file, network communication, or some other type of stream.

Object serialization demo

ObjectSerialization.java

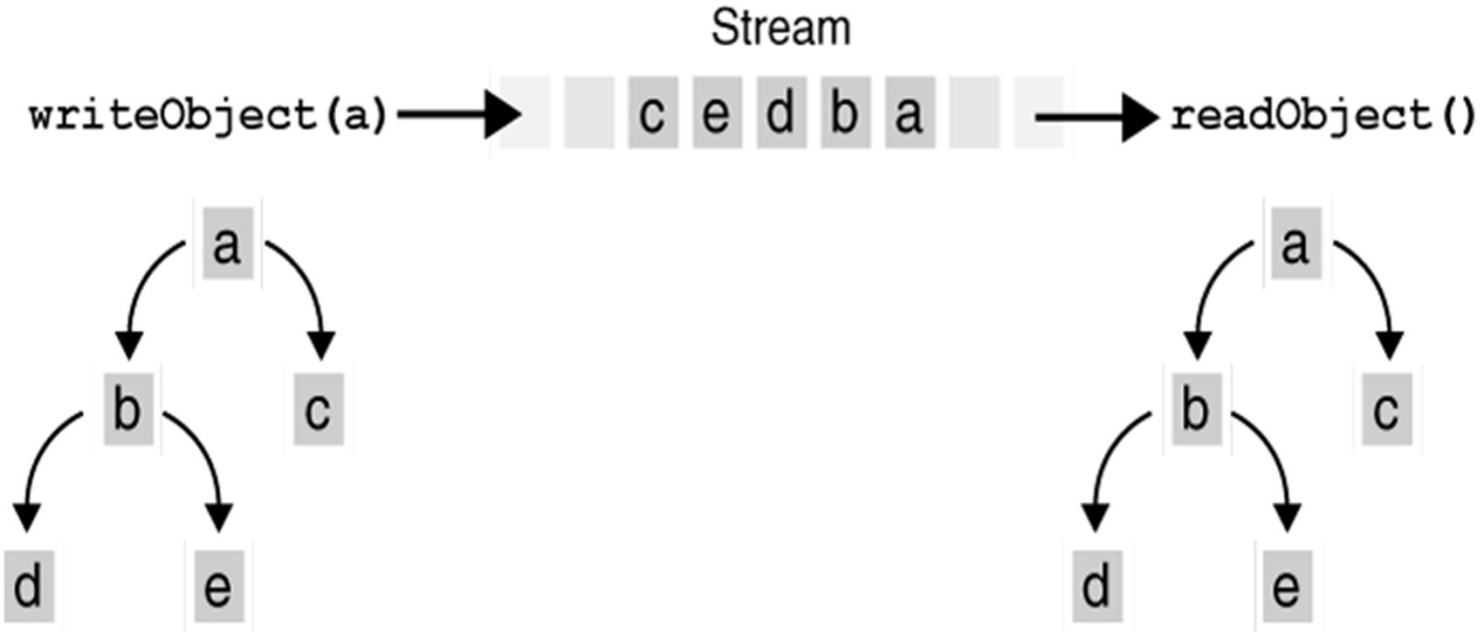
```
1  package myio;
2
3  import java.io.FileInputStream;
4  import java.io.FileOutputStream;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7
8  public class ObjectSerialization {
9      public void serialize2file(Object obj, String dataFile)throws Exception{
10         FileOutputStream fos=new FileOutputStream(dataFile);
11         ObjectOutputStream oos=new ObjectOutputStream(fos);
12         oos.writeObject(obj);
13         oos.close();
14     }
15     public Object deserialize(String datafile)throws Exception{
16         Object obj=null;
17         FileInputStream fis=new FileInputStream(datafile);
18         ObjectInputStream ois=new ObjectInputStream(fis);
19         obj=ois.readObject();
20         ois.close();
21         return obj;
22     }
23 }
```

Output and Input of Complex Objects

- The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic
  - This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects.
- If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to.
  - These additional objects might have their own references, and so on.

I/O of multiple referred-to objects

- Object a contains references to objects b and c, while b contains references to d and e
- Invoking writeObject(a) writes not just a, but all the objects necessary to reconstitute a, so the other four objects in this web are written also
- When a is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



NIO - New I/O

NIO.2 - New I/O version 2

- Is aimed at simplifying I/O in Java
- A new file system and path abstraction
  - Based on java.nio.file.Path
  - Bulk access to file attributes
  - File system specific support (e.g. Symbolic links)
  - Extra providers (e.g. zip files treated as normal files)
- Asynchronous (non-blocking) I/O
  - For sockets and files
  - Mainly utilises java.util.concurrent.Future
- Socket/Channel construct
  - Binding, options and multicast

NIO.2 - Path

- `java.nio2.file.Path` interface
  - Typically represents a file on a file system
  - `normalize()` method removes constructs such as . and .. in a Path
  - `relativize(Path)` constructs relative Path between this path and the one given
  - Lots of other expected methods defined in the interface
  - Dealing with real path, absolute path etc
  - Can convert `java.io.File` objects via the `toFile()` method
- `java.nio2.file.Paths`
  - Helper class, provides `get(URI)` method to return you a Path
  - Uses *FileSystems* helper class under the hood
  - Uses the default file system unless you specify otherwise

NIO.2 - Files

- java.nio2.file.Files helper class
  - Day to day class for performing simple file I/O
- Many common methods
  - copy
  - delete
  - move
  - createDirectory/File/Link
  - write
  - readAllLines
  - walkFileTree (recurse over a directory)
  - newInputStream/OutputStream
- Combined with try-with-resources makes code concise

URLStream to file - Java 6 vs. Java 7

```
void URLstream2FileInJava6()throws Exception{
    URL url = new URL("http://www.java7developer.com/blog/?page_id=97");
    try (FileOutputStream fos = new FileOutputStream(new File("output.txt"));
            InputStream is = url.openStream() ){
        byte[] buf = new byte[4096];
        int len;
        while ((len = is.read(buf)) > 0){
            fos.write(buf, 0, len);
        }
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

```
void URLstream2FileInJava7()throws Exception{
    URL url = new URL("http://www.java7developer.com/blog/?page_id=97");
    try (InputStream in = url.openStream()){
        Files.copy(in, Paths.get("output.txt"));
    }
    catch(IOException ex){
        ex.printStackTrace();
    }
}
```

NIO.2 - Asynchronous I/O

- The ability to perform read/write operations in the background
  - Without having to write your own `java.util.concurrent` code
- Is available for:
  - file I/O (`AsynchronousFileChannel`)
  - networking I/O (`AsynchronousSocketChannel`)
    - And `AsynchronousServerSocketChannel`
- Can work in two styles
  - Future based (order coffee, do something else, collect coffee)
  - Callbacks (order coffee, do something else, have coffee thrown at you)

NIO.2 - Future based file I/O example

```
try {
      Path file = Paths.get("/usr/karianna/foobar.txt");
      AsynchronousFileChannel channel =
        AsynchronousFileChannel.open(file);
      ByteBuffer buffer = ByteBuffer.allocate(100_000_000);
      Future<Integer> result = channel.read(buffer, 0);
      while(!result.isDone()) {
        System.out.println("Do some other stuff");
      }
      Integer bytesRead = result.get();
      //...
    } catch (IOException | ExecutionException | InterruptedException e){
      // Deal with exception
    }
```

NIO.2 - Callback based file I/O example

```
try {
    Path file = Paths.get("/usr/karianna/foobar.txt");
    AsynchronousFileChannel channel = AsynchronousFileChannel.open(file);
    ByteBuffer buffer = ByteBuffer.allocate(100_000_000);

    channel.read(buffer, 0, buffer,
        new CompletionHandler<Integer, ByteBuffer>(){
            @Override
            public void completed(Integer result,
                    ByteBuffer attachment) {
                System.out.println("Bytes read [" + result + "]");
            }
            @Override
            public void failed(Throwable exc, ByteBuffer attachment) {
                // Deal with exception
            }
    });
} catch(Exception xxx){/*Deal with exception*/}
```

Summary

- What is an I/O stream?
- Types of Streams
- Stream class hierarchy
- Control flow of an I/O operation using Streams
- Byte streams
- Character streams
- Buffered streams
- Standard I/O streams
- Data streams
- Object streams
- File class