Object-oriented programming

Lecture №10

I/O Streams

PhD, Alexander Vlasov

Question?

- exception
- try – catch – finally
- throw
- class Throwable
- checked and unchecked exceptions
- Interface AutoCloseable

Using Path Names

- ***Path name***—gives name of file and tells which directory the file is in
- ***Relative path name***—gives the path starting with the directory that the program is in
- Typical UNIX path name:

```
/user/smith/home.work/java/FileClassDemo.java
```

- Typical Windows path name:

```
D:\Work\Java\Programs\FileClassDemo.java
```

- When a backslash is used in a quoted string it must be written as two backslashes since backslash is the escape character:

```
"D:\\Work\\Java\\Programs\\FileClassDemo.java"
```

- Java will accept path names in UNIX or Windows format, regardless of which operating system it is actually running on.

File Methods

- canRead()
- canWrite()
- exists()
- getParent()
- isDirectory()
- isFile()
- lastModified()
- length()

```
File numFile = new File(“numbers.txt”);
 if (numFile.exists())
  System.out.println(numfile.length());
```

File Methods for Modifying

- createNewFile()
- delete()
- makeDir()
- makeDirs()
- renameTo()
- setLastModified()
- setReadOnly()

```
import java.io.*;
import java.util.*;
public class DirList {
   public static void main(String[] args) {
      File path = new File(".");
      String[] list;
      System.out.println(path.getAbsolutePath());
      if(args.length == 0)
         list = path.list();
      else
         list = path.list(new DirFilter(args[0]));
      for (int i = 0; i < list.length; i++)
         System.out.println(list[i]);
   }
}
```

Overview

- IO provides communication with devices (files, console, networks etc.)
- Communication varies (sequential, random-access, binary, char, lines, words, objects, …)
- Java provides a “mix and match” solution based around byte-oriented and character-oriented I/O streams – ordered sequences of data (bytes or chars).
- System streams System.in, (out and err) are available to all Java programs (console I/O) – System.in is an instance of the InputStream class, System.out is an instance of PrintStream
- So I/O involves creating appropriate stream objects for your task.

The IO Zoo

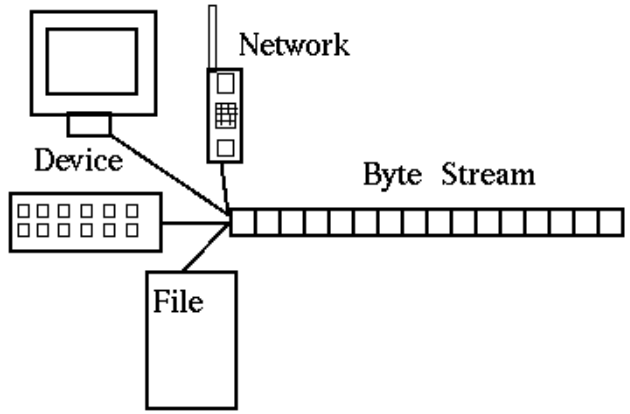
- More than 60 different stream types.
- Based around four abstract classes: InputStream, OutputStream, Reader and Writer.
- Unicode characters (two bytes per char) are dealt with separately with Reader/Writers (and their subclasses).
- Byte oriented I/O is dealt with by InputStream, OutputStream and their subclasses.

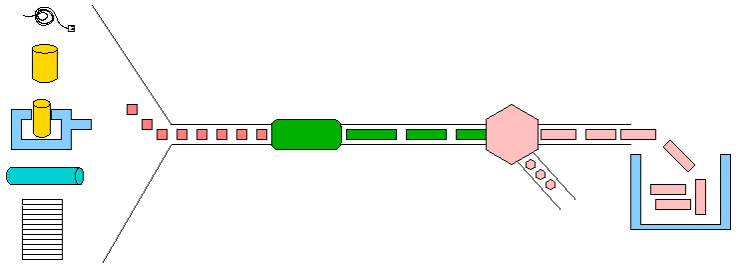
Input and output streams

- **stream**: an abstraction of a source or target of data
  - 8-bit bytes flow to (output) and from (input) streams

- can represent many data sources:
  - files on hard disk
  - another computer on network
  - web page
  - input device (keyboard, mouse, etc.)

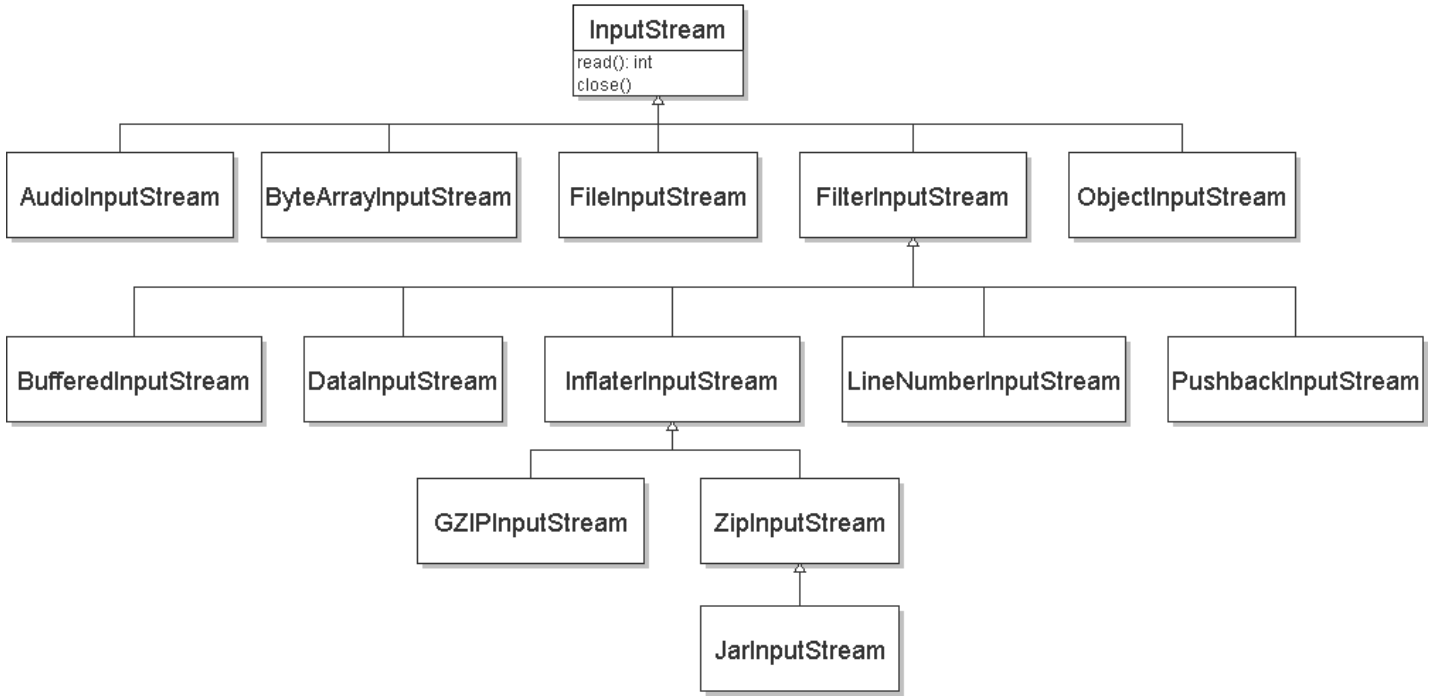
- represented by `java.io` classes
  - `InputStream`
  - `OutputStream`





Streams and inheritance

- all input streams extend common superclass `InputStream`;
  all output streams extend common superclass `OutputStream`
  - guarantees that all sources of data have the same methods
  - provides minimal ability to read/write one byte at a time



Input streams

- constructing an input stream:

Constructor
`public FileInputStream(String name) throws IOException`
`public ByteArrayInputStream(byte[] bytes)`
`public SequenceInputStream(InputStream a, InputStream b)`

(various objects also have methods to get streams to read them)

- methods common to all input streams:

Method	Description
`public int` **`read`**`() throws IOException`	reads/returns a byte (-1 if no bytes remain)
`public void` **`close`**`() throws IOException`	stops reading

Output streams

- constructing an output stream:

Constructor
`public FileOutputStream(String name) throws IOException`
`public ByteArrayOutputStream()`
`public PrintStream(File file)` `public PrintStream(String fileName)`

- methods common to all output streams:

Method	Description
`public void` **`write`**`(int b) throws IOException`	writes a byte
`public void` **`close`**`() throws IOException`	stops writing (also flushes)
`public void flush() throws IOException`	forces any writes in buffers to be written

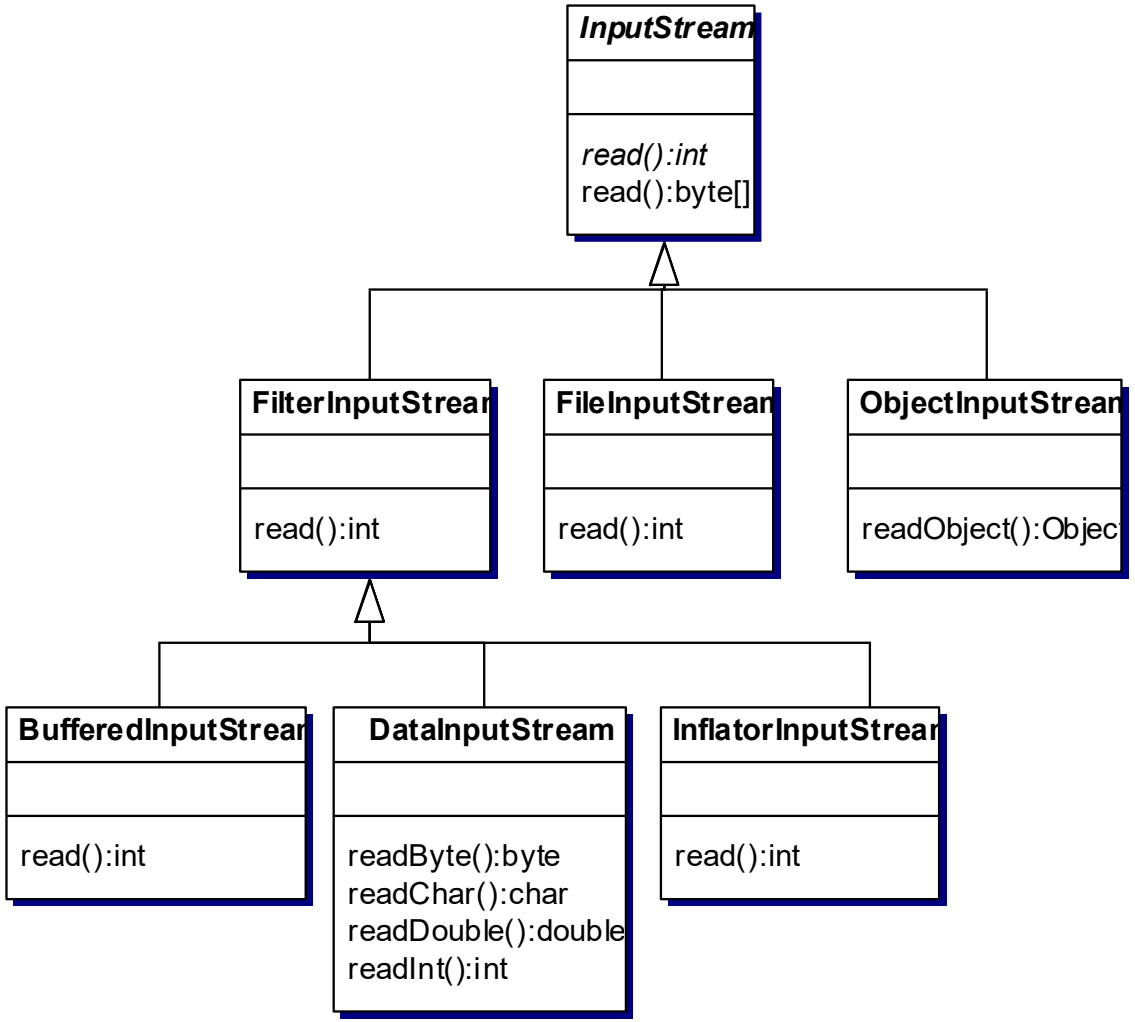
Handling IOException

- `IOException` cannot be ignored
  - either handle it with a catch block
  - or defer it with a `throws`-clause

We will put code to open the file and write to it in a `try`-block and write a `catch`-block for this exception :

```
catch(IOException e)
{
  System.out.println("Problem with output...“);
}
```

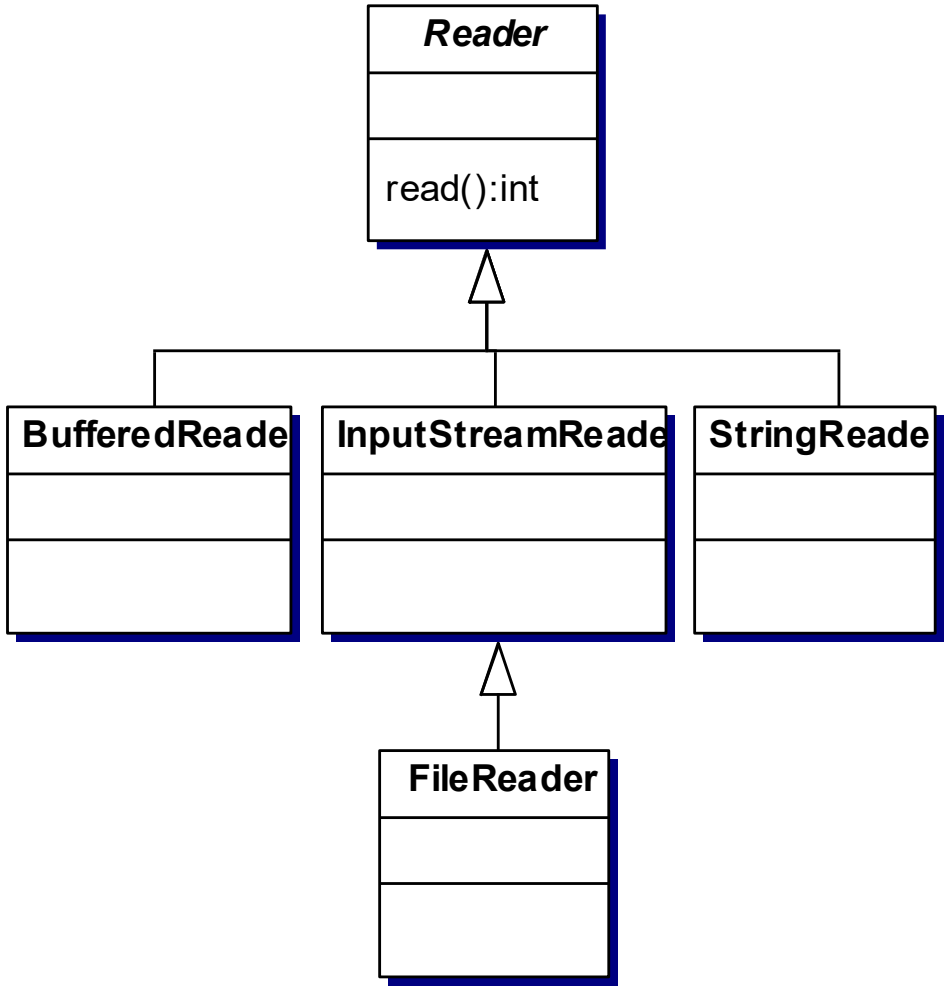
More on Input



FilterInputStream

- A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
- The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the contained input stream.
- Subclasses of FilterInputStream may further override some of these methods and may also provide additional methods and fields.

Readers


Reader
read():int
BufferedReade
InputStreamReade
StringReade
FileReader

BufferedReader

```
BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));

InputStreamReader isr = new
        InputStreamReader(new FileInputStream("FileInput.java")); //slow:
   unbuffered
```

This is easier (if we're happy with the default character encoding and buffer size):

```
 InputStreamReader isr = new
        FileReader(" FileInput.java");
```

OutputStreams and Writers

- Basically, a “mirror image” of InputStreams and Readers.
- Wrapping is the same, e.g.,

```
BufferedWriter bw =
        new BufferedWriter(new OutputStreamWriter(System.out));
    String s;
    try {
        while ((s = br.readLine()).length() != 0) {
        bw.write(s, 0, s.length());
        bw.newLine();
        bw.flush();
        }
    }
```

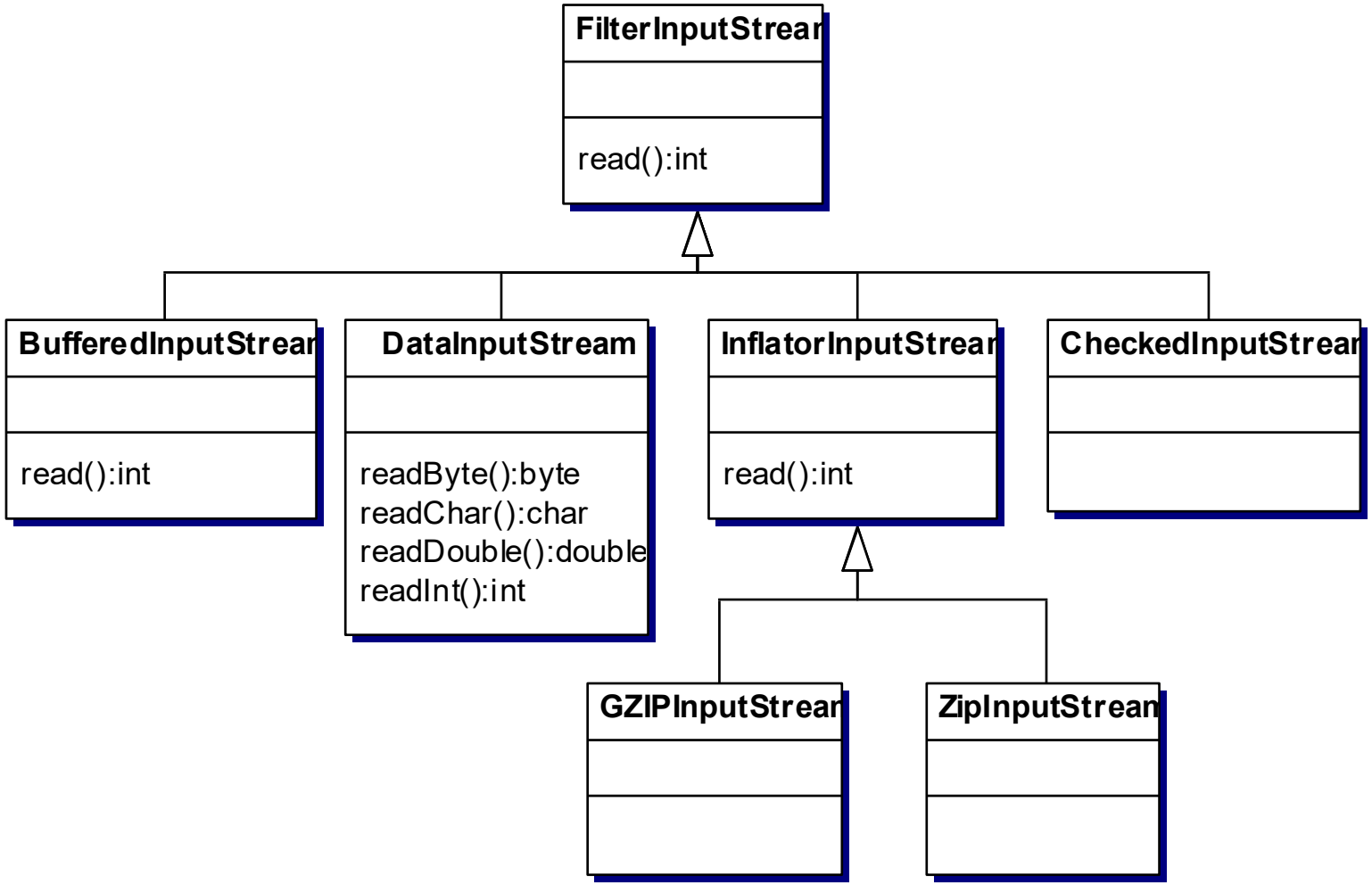
FileWriter

- Again, basically the same. The constructors are
  - FileWriter(File file)
  - FileWriter(FileDescriptor fd)
  - FileWriter(String s)
  - FileWriter(String s, boolean append)
- The last one allows appending, rather than writing to the beginning (and erasing an existing file!).
- These *will* create files!
- There is also PrintWriter

PrintWriter

```
PrintWriter out =
   new PrintWriter(new BufferedWriter(new FileWriter("Test.txt")));
String s;
try {
       while ((s = br.readLine()).length() != 0) {
       bw.write(s, 0, s.length());
       bw.newLine();
       bw.flush();
       out.println(s);
       //out.flush();
       }
}
catch(IOException e) {
}
out.close();    // also flushes
```

The Compression Input Classes



GZIP Example (1st part)

```
import java.io.*;
import java.util.zip.*;
public class GZIPCompress {
   public static void main(String[] args) throws IOException {
      BufferedReader in = new BufferedReader(
         new FileReader(args[0]));
      BufferedOutputStream out = new BufferedOutputStream(
          new GZIPOutputStream(new FileOutputStream("test.gz")));
      System.out.println("Writing file");
      int c;
      while((c = in.read()) != -1)
         out.write(c);
      in.close();
      out.close();
```

GZIP Example (2nd part)

```
        System.out.println("Reading file");
        BufferedReader in2 =
            new BufferedReader(
                new InputStreamReader(
                    new GZIPInputStream(
                        new FileInputStream("test.gz"))));  // whew!
        String s;
        while((s = in2.readLine()) != null)
            System.out.println(s);
    }
}
```

Object Streams

- **`ObjectOutputStream`** class can save a entire objects to disk
- **`ObjectOutputStream`** class can read objects back in from disk
- Objects are saved in binary format; hence, you use streams and not writers

Write out an object

- The object output stream saves all instance variables

```
BankAccount b = . . .;

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

out.writeObject(b);
```

Read in an object

- **`readObject`** returns an **`Object`** reference
- Need to remember the types of the objects that you saved and use a cast

```
ObjectInputStream in = new ObjectInputStream(
  new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

Exceptions

- **`readObject`** method can throw a **`ClassNotFoundException`**
- It is a checked exception
- You must catch or declare it

Writing an Array

- Usually want to write out a collection of objects:

```
BankAccount[] arr = new BankAccount[size];

// Now add size BankAccount objects into arr
out.writeObject(arr);
```

Reading an Array

- To read a set of objects into an array

```
BankAccount[] ary = (BankAccount[]) in.readObject();
```

Object Streams

- Very powerful features
  - Especially considering how little we have to do

- The **`BankAccount`** class as is actually will not work with the stream
  - Must implement **`Serializable`** interface in order for the formatting to work

Object Streams

```
class BankAccount implements Serializable
{
  . . .
}
```

- IMPORTANT: **`Serializable`** interface has no methods.
- No effort required

Custom serialization

```
public class Logon implements Externalizable {
    private String login;
    private String password;
    public Logon() {
    }
    public Logon(String login, String password) {
        this.login = login;
        this.password = password;
    }
   @Override
    public String toString() {
        return "Logon{" +
                "login='" + login + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
```

```
   @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(login);
    }
@Override
    public void readExternal(ObjectInput in)
                                throws IOException, ClassNotFoundException {
        login = (String) in.readObject();
    }
}
```

Serialization

- Serialization: process of saving objects to a stream
  - Each object is assigned a serial number on the stream
  - If the same object is saved twice, only serial number is written out the second time
  - When reading, duplicate serial numbers are restored as references to the same object

Serialization

- Why isn’t everything serializable?
  - Security reasons – may not want contents of objects printed out to disk, then anyone can print out internal structure and analyze it
  - Example: Don’t want SSN ever being accessed
  - Could also have temporary variables that are useless once the program is done running

Tokenizing

- Often several text values are in a single line in a file to be compact

  `"25 38 36 34 29 60 59"`

- The line must be broken into parts (i.e. ***tokens***)

  `"25"`
  `"38"`
  `"36"`

- tokens then can be parsed as needed

  `"25"` can be turned into the integer `25`

Tokenizing

- Inputting each value on a new line makes the file very long
- May want a file of customer info – name, age, phone number all on one line
- File usually separate each piece of info with a delimiter – any special character designating a new piece of data (space in previous example)

StringTokenizer

```
StringTokenizer st = new StringTokenizer("this is a test");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
```

```
String[] result = "this is a test".split("\\s");
    for (int x=0; x<result.length; x++)
        System.out.println(result[x]);
```

output:

this

is

a

test

Scanner

```
public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int number = sc.nextInt();
  }
```

```
Scanner sc = new Scanner(new File("myNumbers"));
     while (sc.hasNextLong()) {
        long aLong = sc.nextLong();
     }
```

Exception in thread "main" java.util.InputMismatchException at java.util.Scanner.throwFor(Scanner.java:864) at java.util.Scanner.next(Scanner.java:1485) at java.util.Scanner.nextInt(Scanner.java:2117) at java.util.Scanner.nextInt(Scanner.java:2076) at Main.main(Main.java:10) Process finished with exit code 1

```
String input = "1 fish 2 fish red fish blue fish";
    Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
    System.out.println(s.nextInt());
    System.out.println(s.nextInt());
    System.out.println(s.next());
    System.out.println(s.next());
    s.close();
```

nextInt(),nextLine(), nextByte(), nextShort(), nextLong(), nextFloat(), nextDouble()

hasNextInt(),hasNextLine(), hasNextByte(), hasNextShort(), hasNextLong(), hasNextFloat(), hasNextDouble()

java.nio.*

- Java NIO: File, Path
  - Exceptions
  - More function (symbols links, access right and etc.)
- Java NIO: Channels and Buffers
  - In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.
- Java NIO: Non-blocking IO
  - Java NIO enables you to do non-blocking IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.
- Java NIO: Selectors
  - Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.