Java I/O

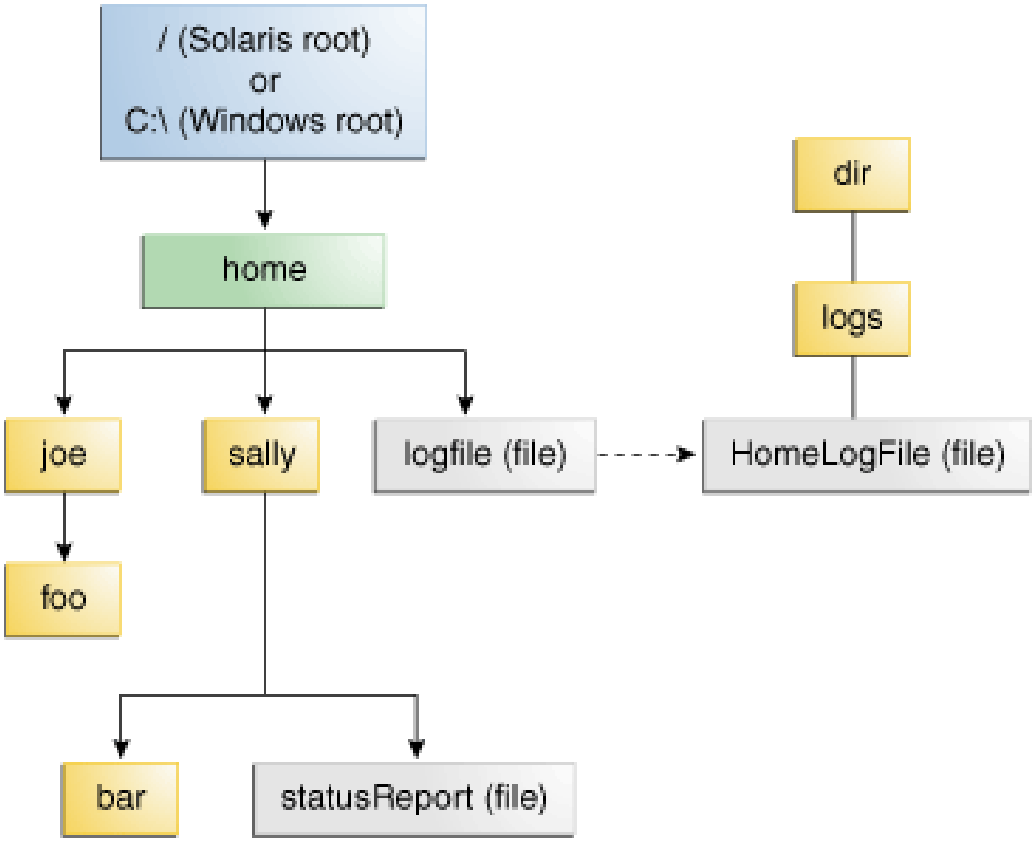
Оперативни системи 2014
Аудиториски вежби

Java I/O

- The Java Input/Output (I/O) is a part of the `java.io` package.
- The `java.io` package contains a fairly large number of classes that deal with Java input and output.
- The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow data to be read from and written to files or other input and output sources.
- `InputStream` and `OutputStream` are central classes in the package which are used for reading from and writing to byte streams, respectively.

File System Facts

- What is a Path?
  - Relative or Absolute?
  - Symbolic Links



The `File` Class

- Before getting into the classes that actually read and write data to streams, we'll look at a library utility that assist you with file and directory manipulation.
- The `File` Class doesn't refer to a file for input or output, but it is used for manipulation of the file system that holds the files and directories.

The `File` Class operations

`public String getName()`	`public boolean canRead()`
`public String getParent()`	`public boolean canWrite()`
`public String getPath()`	`public boolean exists()`
`public long lastModified()`	`public boolean isDirectory()`
`public long length()`	`public boolean isFile()`
`public boolean delete()`	`public boolean isHidden()`
`public String[] list() (FilenameFilter filter)`	`public boolean setReadOnly()`
`public File[] listFiles() (FilenameFilter fil)`	
`public boolean mkdir()`	
`public boolean mkdirs()`	
`public boolean renameTo(File dest)`	
`public boolean setLastModified(long time)`	

File Operations …

```
File f=new File(name);
// Tests whether the file or directory denoted by this abstract pathname exists.
boolean exists = f.exists();
// Tests whether the file denoted by this abstract pathname is a directory.
boolean isDirectory = f.isDirectory();
// The following lines gives the permissions of the process over the file
boolean canRead = f.canRead();
boolean canWrite = f.canWrite();
boolean canExecute = f.canExecute();

// Returns the name of the file or directory denoted by this abstract pathname.
String fileName = f.getName();
//Tests whether the file or directory denoted by this abstract pathname exists.
String absolutPath = f.getAbsolutePath();
// Returns the pathname string of this abstract pathname's parent, or null if this
// pathname does not name a parent directory.
String parentPath = f.getParent();
// Returns the pathname string of this abstract pathname's parent, or null if this
// pathname does not name a parent directory.
f.length();
```

… File Operations

```
// Atomically creates a new, empty file named by this abstract pathname if and only
// if a file with this name does not yet exist.
boolean created = f.createNewFile();
// Deletes the file or directory denoted by this abstract pathname.
boolean deleted = f.delete();
// Returns the pathname string of this abstract pathname's parent, or null if
// this pathname does not name a parent directory.
f.mkdir();
// Creates the directory named by this abstract pathname, including any necessary
// but nonexistent parent directories.
f.mkdirs();
// Renames the file denoted by this abstract pathname.
boolean renamed=f.renameTo(new File("newFilePath"));
// Returns the pathname string of this abstract pathname's parent, or null if
// this pathname is not name a parent directory.
String[] files = f.list();
```

Directory Listing

- Suppose you want to see a directory listing.
- The `File` object can be used in two ways:
  - for a full list of what the directory denoted by the `File` object contains, use `list()` without arguments;
  - for a restricted list, use a “directory filter”;

Directory Listing – Example 1 …

- List the directory content, with or without a filter;

```
public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        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]);
    }
}
```

… Directory Listing – Example 1

- List the directory content, with or without a filter;

```
class DirFilter implements FilenameFilter {
    String afn;

    DirFilter(String afn) {
        this.afn = afn;
    }

    public boolean accept(File dir, String name) {
        // Strip path information:
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
}
```

Directory Listing – Example 2

- Recursively list all subdirectories from a starting point in the file system, and print the file permissions;

```
public static void listFile(String absolutePath, String prefix) {
    File file = new File(absolutePath);

    if (file.exists()) {
        File[] subfiles = file.listFiles();
        for (File f : subfiles) {
            // print the permissions in unix like format
            System.out.println(prefix + getPermissions(f) + "\t"
                    + f.getName());

            // Recursively show the content of sub-directories
            if (f.isDirectory()) {
                listFile(f.getAbsolutePath(), prefix + "\t");
            }
        }
    }
}

public static String getPermissions(File f) {
    return String.format("%s%s%s", f.canRead() ? "r" : "-",
            f.canWrite() ? "w" : "-", f.canExecute() ? "x" : "-");
}
```

Checking for and creating directories

- The `File` class is more than just a representation of an existing file or directory.
- You can also use a `File` object to create a new directory or an entire directory path if it doesn't exist.
- The following example shows some of the other methods available with the `File` class.

Directory Manipulation – Example …

```
public class MakeDirectories {
    private final static String usage = "Usage:MakeDirectories path1 ...\n"
            + "Creates each path\n" + "Usage:MakeDirectories -d path1 ...\n"
            + "Deletes each path\n" + "Usage:MakeDirectories -r path1 path2\n"
            + "Renames from path1 to path2\n";

    private static void usage() {
        System.err.println(usage);  System.exit(1);
    }

    private static void fileData(File f) {
        System.out.println("Absolute path: " + f.getAbsolutePath()
                + "\n Can read: " + f.canRead() + "\n Can write: "
                + f.canWrite() + "\n getName: " + f.getName()
                + "\n getParent: " + f.getParent() + "\n getPath: "
                + f.getPath() + "\n length: " + f.length()
                + "\n lastModified: " + f.lastModified());
        if (f.isFile())
            System.out.println("it's a file");
        else if (f.isDirectory())
            System.out.println("it's a directory");
    }
```

… Directory Manipulation – Example …

```
public static void main(String[] args) {
    if (args.length < 1)        usage();
    if (args[0].equals("-r")) {
        if (args.length != 3)       usage();
        File old = new File(args[1]), rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if (args[0].equals("-d")) {
        count++;
        del = true;
    }
```

… Directory Manipulation – Example

```
        for (; count < args.length; count++) {
            File f = new File(args[count]);
            if (f.exists()) {
                System.out.println(f + " exists");
                if (del) {
                    System.out.println("deleting..." + f);
                    f.delete();
                }
            } else { // Doesn't exist
                if (!del) {
                    f.mkdirs();
                    System.out.println("created " + f);
                }
            }
            fileData(f);
        }
    }
}
```

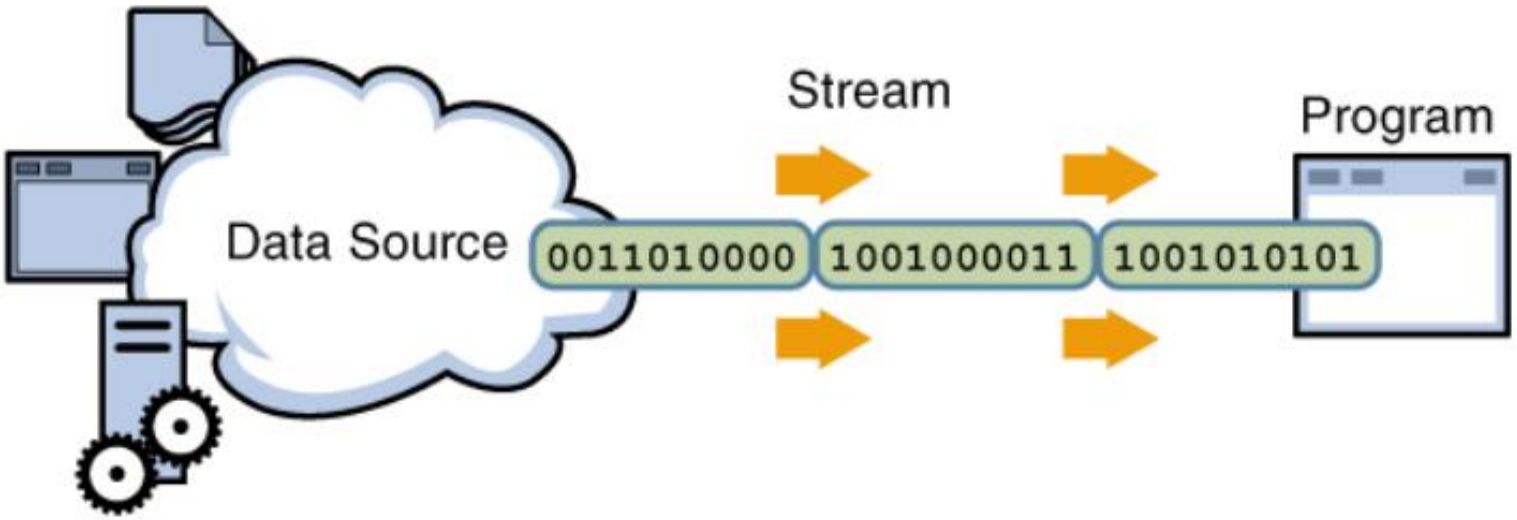
Input and Output

- The basic organization of the `java.io` classes, consisting of:
  - Input and Output streams (byte oriented streams);
  - Readers and Writers (character oriented streams);
  - Data and Object I/O streams;

- An I/O Stream represents an input source or an output destination.
- A stream can represent different kinds of sources and destinations:
  - disk files, devices, other programs, memory arrays
- A stream supports different kinds of data:
  - simple bytes, primitive data types, localized characters, objects

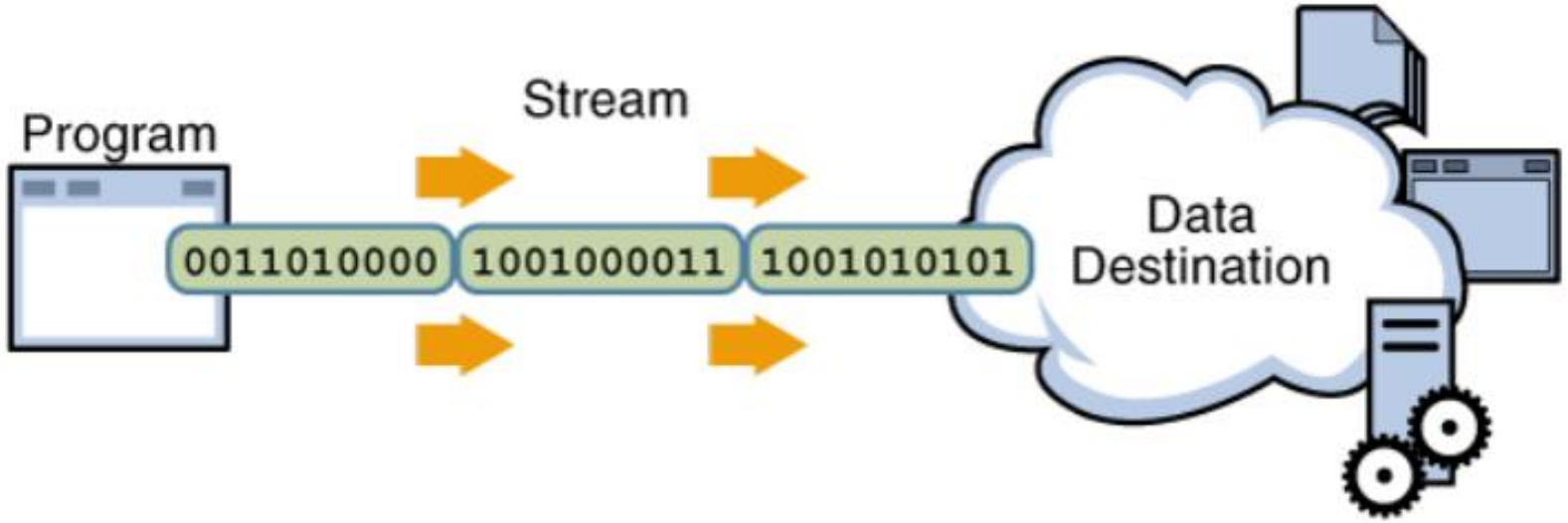
Input

- Reading information into a program



Output

- Writing information from a program



Input and Output

- Everything is derived from:
  - `InputStream` or `Reader` classes have basic methods called `read()`, used for reading a single byte (char), or an array of bytes (chars);
  - `OutputStream` or `Writer` classes have basic methods called `write()`, used for writing a single byte (char), or an array of bytes (chars);

Types of `InputStream`

- In the `InputStream` class, bytes can be read from different sources:
  - An array of bytes;
  - A `String` object;
  - A file;
  - A pipe;
  - A sequence of other streams, so you can collect them together into a single stream;
  - Other sources, such as an Internet connection;

`InputStream` methods

- Various methods are included in the `InputStream` class:
  - `read()` - reads a single byte, an array, or a subarray of bytes. It returns the bytes read, the number of bytes read, or -1 if end-of-file has been reached;
  - `skip()` - which takes long, skips a specified number of bytes of input and returns the number of bytes actually skipped;
  - `available()` - returns the number of bytes that can be read without blocking. Both the input and output can block threads until the byte is read or written;
  - `close()` - closes the input stream to free up system resources;

Types of `InputStream`

Class	Function
`ByteArrayInputStream`	Allows a buffer in memory to be used as an `InputStream`
`StringBufferInputStream`	Converts a `String` into an `InputStream`
`FileInputStream`	For reading information from a file
`PipedInputStream`	Produces the data that's being written to the associated `PipedOutputStream`. Implements the "piping" concept
`SequenceInputStream`	Converts two or more `InputStream` objects into a single `InputStream`
`FilterInputStream`	Abstract class that is an interface for decorators that provide useful functionality to the other `InputStream` classes

Types of `OutputStream`

- Bytes can be written to three different types of sinks:
  - An array of bytes
  - A file
  - A pipe

`OutputStream` methods

- The `OutputStream` class provides several methods:
  - `void write()` - method writes an integer byte, a byte array or subarray of bytes;
  - `void flush()` - method forces any buffered output to be written;
  - `void close()` - method closes the stream and frees up system resources;
    - It is important to close your output files, because sometimes the buffers do not get completely flushed and, as a consequence, the write is not complete.

Types of `OutputStream`

Class	Function
`ByteArrayOutputStream`	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer
`FileOutputStream`	For sending information to a file
`PipedOutputStream`	Any information you write to this automatically ends up as input for the associated `PipedInputStream`. Implements the "piping" concept
`FilterOutputStream`	Abstract class that is an interface for decorators that provide useful functionality to the other `OutputStream` Classes

Reading from `InputStream` with `FilterInputStream`

- `FilterInputStream` classes allow reading different types of primitive data, as well as `String` objects
  - All the methods start with 'read', such as `readByte()`, `readFloat()`, etc.
- `FilterInputStream` classes can modify the way an `InputStream` behaves internally:
  - whether it's buffered or unbuffered;
  - whether it keeps track of the lines it's reading;
  - whether you can push back a single character;

Types of `FilterInputStream`

Class	Function
`DataInputStream`	Used in concert with `DataOutputStream`, so you can read primitives (`int`, `char`, `long`, etc.) from a stream in a portable fashion
`BufferedInputStream`	Use this to prevent a physical read every time you want more data
`LineNumberInputStream`	Keeps track of line numbers in the input stream; you can call `getLineNumber()` and `setLineNumber(int)`
`PushbackInputStream`	Has a one-byte pushback buffer so that you can push back the last character read

Writing from `OutputStream` with `FilterOutputStream`

- `FilterOutputStream` classes allow the user:
  - to format each of the primitives types and `String` objects onto a stream in such a way that any `DataInputStream` can read them;
    - All the methods start with 'write', such as `writeByte()`, `writeFloat()`, etc .
  - to print all of the primitive data types and `String` objects in a viewable format;

Types of `FilterOutputStream`

Class	Function
`DataOutputStream`	Used in concert with `DataInputStream` so you can write primitives (`int`, `char`, `long`, etc.) to a stream in a portable fashion
`PrintStream`	For producing formatted output. While `DataOutputStream` handles the storage of data, `PrintStream` handles display
`BufferedOutputStream`	Use this to prevent a physical write every time you send a piece of data. You can call `flush()` to flush the buffer

`Readers` and `Writers`

- What are `Readers`?
  - `Readers` are character-based input streams that read Unicode characters.
- What are `Writers`?
  - `Writers` are character-based output streams that write character bytes and turn Unicode into bytes.

- Input and output done with these character-based streams automatically translates to and from the local character set.

Readers and Writers

Sources and Sinks
InputStream
OutputStream
FileInputStream
FileOutputStream
StringBufferInputString
ByteArrayInputStream
ByteArrayOutputStream
PipedInputStream
PipedOutputStream

Modifying string behavior

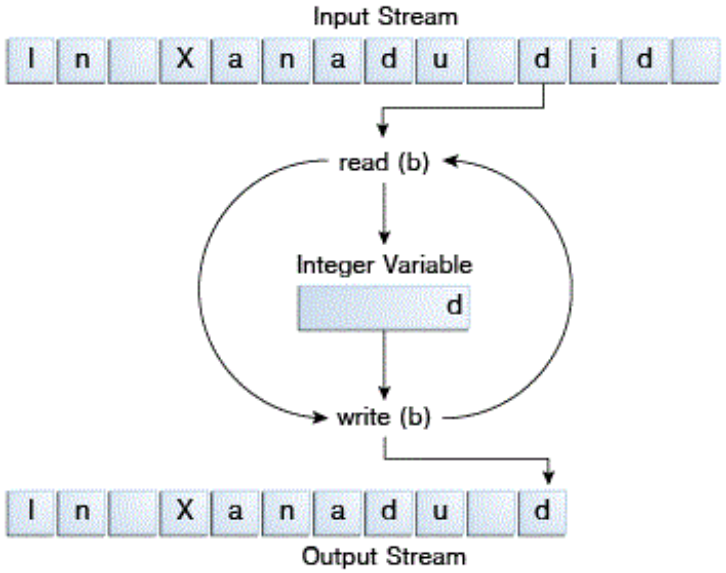
Filters
FilterInputStream
FilterOutputStream
BufferedInputStream
BufferedOutputStream
DataInputString
PrintStream
LineNumberInputStream
StreamTokenizer
PushbackInputStream

Typical uses of I/O Streams

- You can combine the I/O stream classes in many different ways.
- But, you'll probably just use a few combinations.
- The following examples can be used as a basic reference for typical I/O usage:
  - Buffered input file
  - Input from memory
  - Formatted memory input
  - Basic file output
  - Text file output shortcut
  - Storing and recovering data
  - Random access file

Read and Write, byte by byte

```
public static void copyStream(InputStream in, OutputStream out)
        throws IOException {
    try {
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } finally {
        // why are the streams closed in finally block?
        if (in != null) {
            // all streams must be closed
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```



Most common `InputStream` Mistake ...

```
public static void badReading(InputStream in) throws IOException {
    try {
        byte[] buffer = new byte[100];
        // this is wrong !!!
        int bytesRead = in.read(buffer);
        doSomethingWithReadData(buffer);
    } finally {
        if (in != null) {
            in.close();
        }
    }
}
```

… Most common `InputStream` Mistake

```
public static void correctReading(InputStream in) throws IOException {
    try {
        byte[] buffer = new byte[100];
        // this is the right way
        while (in.read(buffer) != -1) {
            doSomethingWithReadData(buffer);
        }
    } finally {
        if (in != null) {
            in.close();
        }
    }
}
```

Reading a Text File

```
public static String readTextFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(
            new FileInputStream(path), "UTF-8"));

    String line = null;

    // this approach will fail for huge files
    StringBuilder sb = new StringBuilder();

    while ((line = br.readLine()) != null) {
        sb.append(line).append("\n");
    }
    br.close();
    return sb.toString();
}
```

Reading from the Standard Input

```
public static void stdinRead() throws IOException {
    BufferedReader stdin = new BufferedReader(new InputStreamReader(
            System.in));
    String s;
    System.out.print("Enter a line:");
    while ((s = stdin.readLine()) != null && s.length() != 0)
        System.out.println(s);
    // An empty line or Ctrl-Z terminates the program
}
```

Write Text File & Text File Content Copy

```
public static void writeTextFile(String path, String text, boolean append)
        throws IOException {
    BufferedWriter br = new BufferedWriter(new FileWriter(path, append));
    br.write(text);
    br.close();
}

public static void memorySafeTextFileCopy(String from, String to)
        throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(from));
    BufferedWriter bw = new BufferedWriter(new FileWriter(to));

    String line = null;

    while ((line = br.readLine()) != null) {
        bw.write(line + "\n");
    }
    br.close();
    bw.close();
}
```

File Output with Line Numbers

```
public static void fileOutputWithLineNumbers() throws IOException {
    String outFile = "BasicFileOutput.out";
    BufferedReader in = null;       PrintWriter out = null;
    try {
        in = new BufferedReader(new FileReader("src/BasicFileOutput.java"));
        out = new PrintWriter(new BufferedWriter(new FileWriter(outFile)));
        // Here's the shortcut for the previous line:
        // PrintWriter out = new PrintWriter(outFile);
        int lineCount = 1;
        String s;
        while ((s = in.readLine()) != null)
            out.println(lineCount++ + ": " + s);
    } finally {
        if (out != null)        out.close();
        if (in != null)         in.close();
    }
    // Show the stored file (use the method from the previous example):
    System.out.println(readTextFile(outFile));
}
```

Storing and Retrieving Data

```
public static void dataReadWrite() throws IOException {
    DataOutputStream out = null;        DataInputStream in = null;
    try {
        out = new DataOutputStream(new BufferedOutputStream(new FileOutputStream("Data.txt")));
        out.writeDouble(3.14159);
        out.writeUTF("That was pi");
        out.writeDouble(1.41413);
        out.writeUTF("Square root of 2");
        in = new DataInputStream(new BufferedInputStream(new FileInputStream("Data.txt")));
        System.out.println(in.readDouble());
        // Only readUTF() will recover the Java-UTF String properly:
        System.out.println(in.readUTF());
        System.out.println(in.readDouble());
        System.out.println(in.readUTF());
    } finally {
        if (out != null)    out.close();
        if (in != null)     in.close();
    }
}
```


Random Access File

```
static String filePath = "rtest.dat";

static void display() throws IOException {
    RandomAccessFile rf = null;
    try {
        rf = new RandomAccessFile(filePath, "r");
        for (int i = 0; i < 7; i++) {   System.out.println("Value " + i + ": " + rf.readDouble()); }
        System.out.println(rf.readUTF());
    } finally {     if(rf!=null)    rf.close();     }
}

public static void randomAccess() throws IOException {
    RandomAccessFile rf = null;
    try {
        rf = new RandomAccessFile(filePath, "rw");
        for (int i = 0; i < 7; i++) {   rf.writeDouble(i * 1.414); }
        rf.writeUTF("The end of the file");
    } finally {     if (rf != null)     rf.close();     }
    display();
    try {
        rf = new RandomAccessFile(filePath, "rw");
        // Sets the file-pointer offset, measured from the beginning of this
        // file, at which the next read or write occurs
        rf.seek(5 * 8);
        rf.readDouble();
        rf.writeDouble(47.0001);
    } finally { if (rf != null)     rf.close();     }
    display();
}
```

Redirecting Standard I/O

```
public static void redirect() throws IOException {
    InputStream consoleIn = System.in;  BufferedInputStream in = null;
    PrintStream console = System.out;   PrintStream out = null;

    try {
        in = new BufferedInputStream(new FileInputStream("src/MemoryInput.java"));
        out = new PrintStream(new BufferedOutputStream(new FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String s;
        while ((s = br.readLine()) != null)
            System.out.println(s);
    } finally {
        if (in != null)         in.close();
        if (out != null)        out.close();
        System.setIn(consoleIn);
        System.setOut(console);
    }

}
```