DM550 / DM857
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM550/

http://imada.sdu.dk/~petersk/DM857/

IN & OUTPUT USING STREAMS

Streams

- streams are ADTs for representing input and output
- source for input can e.g. be files, keyboard, network resources
- output can go to e.g. files, terminal, network resources
- four categories of streams in java.io package:

	Input	Output
byte	InputStream	OutputStream
character	Reader	Writer

- byte streams are for machine-readable data
  - reading one unit is reading one byte (= 8 bits)
- character streams are for human-readable data
  - reading one unit is reading one character (= 16 bits)
  - readers/writers translate 8-bit files etc. into 16-bit unicode

InputStream ADT: Specification

- data = potentially infinite stream of bytes
- operations are given by the following interface:

```
public interface InputStreamADT {
    public int available();          // how much more can be read?
    public void close();             // close the stream
    public int read();               // next byte of the stream
    public int read(byte[] b);       // read n bytes into b and return n
    public int read(byte[] b, int off, int len);     // max len from b[off]
    public long skip(long n);        // skip n bytes
}
```

- all input byte streams are subclasses of java.io.InputStream

InputStream ADT: Example

- Example (reading up to 1024 bytes from a file):

```
InputStream input = new FileInputStream(new File("test.txt"));
byte[] data = new byte[1024];
int readSoFar = 0;
do {
    readSoFar += input.read(data, readSoFar, 1024-readSoFar);
} while (input.available() > 0 && readSoFar < 1024);
input.close();
System.out.println("Got "+readSoFar+" bytes from test.txt!");
```

- if you think that is horrible …
- … you now understand, why we used java.util.Scanner ☺

OutputStream ADT: Specification

- data = potentially infinite stream of bytes
- operations are given by the following interface:

```
public interface OutputStreamADT {
    public void close();            // close the stream
    public void write(int b);       // write b to the stream
    public void write(byte[] b);// write b.length bytes from b
    public void write(byte[] b, int off, int len);  // len bytes from b[off]
    public void flush();            // forces buffers to be written
}
```

- all output byte streams are subclasses of java.io.OutputStream

OutputStream ADT: Example

- Example (copying a file):

```
InputStream in = new FileInputStream(new File("test.txt"));
OutputStream out = new FileOutputStream(new File("test.out"));
int total = 0;
byte[] block = new byte[4096];
while (true) {
    int read = in.read(block);
    if (read == -1) {  break;  }
    out.write(block, 0, read);
    total += read;
}     in.close();        out.close();
System.out.println("Copied "+total+" bytes from test.txt!");
```

Reader ADT: Specification

- data = potentially infinite stream of characters
- operations are given by the following interface:

```
public interface ReaderADT {
    public boolean ready();      // input available?
    public void close();         // close the stream
    public int read();           // next character of the stream
    public int read(char[] c);   // read n characters into c and return n
    public int read(char[] c, int off, int len);     // max len from c[off]
    public int read(CharBuffer target);              // read into CharBuffer
    public long skip(long n);    // skip n characters
}
```

- all input character streams are subclasses of java.io.Reader

Reader ADT: Example

- Example (reading characters from a file):

```
Reader input = new FileReader(new File("test.txt"));
StringBuffer buffer = new StringBuffer();
while (true) {
    int ch = input.read();
    if (ch == -1) {  break;  }
    buffer.append((char)ch);
}
input.close();
System.out.println("Read the following content:");
System.out.println(buffer.toString());
```

- less horrible … but we still prefer java.util.Scanner ☺

Writer ADT: Specification

- data = potentially infinite stream of characters
- operations are given by the following interface:

```
public interface WriterADT {
    public void close();              // close the stream
    public void write(int c);         // write one character to the stream
    public void write(char[] c);// write c.length characters
    public void write(char[] c, int off, int len);  // len chars from c[off]
    public void write(String s); // write s.length() characters
    public void write(String s, int off, int len);  // len chars from s at off
    public void flush();              // forces buffers to be written
}
```

- all output character streams are subclasses of java.io.Writer

Writer ADT: Example

- Example (copying a text file character by character):

```
Reader in = new FileReader(new File("test.txt"));
Writer out = new FileWriter(new File("test.out"));
while (true) {
    int ch = in.read();
    if (ch == -1) {  break;  }
    out.write(ch);
}
in.close();
out.close();
System.out.println("Done!");
```

Character vs Byte Streams

- Java has classes to convert between character and byte streams
- characters are converted according to specified char set
- default char set is 16-bit unicode

	Input	Output
byte -> char	InputStreamReader	DataOutputStream
char -> byte	DataInputStream	OutputStreamWriter

- InputStreamReader reads characters from byte stream
- DataOutputStream can be used to write primitive types + String
- OutputStreamWrite write characters to byte stream
- DataInputStream can be used to read primitive types + String

PrintWriter & PrintStream

- classes that extend Writer and OutputStream
- add comfortable methods for printing and formatting data
- provide methods such as for example
  - print – like in System.out.print
  - println – like in System.out.println
  - printf – like in System.out.printf
- in fact, System.out is an instance of PrintStream
- Example (writing comfortably to a file):

```
File file = new File("test.out");       String name = "Peter";
PrintStream out = new PrintStream(new FileOutputStream(file));
out.printf("Hej %s! How are you?\n", name);
out.close();
```

NETWORKING & MULTI-THREADING

Accessing Network Resources

- like File represents files, URL represents network resources
- Example 1 (downloading course web site into file):

```
URL url = new URL("http://imada.sdu.dk/~petersk/DM537/");
InputStream input = url.openStream();
OutputStream output = new FileOutputStream("dm537.html");
byte[] block = new byte[4096];
while (true) {
    int read = input.read(block);
    if (read == -1) {  break;  }
    output.write(block, 0, read);
}
input.close();  output.close();
```

Accessing Network Resources

- like File represents files, URL represents network resources
- Example 2 (downloading course web site into file):

```
URL url = new URL("http://imada.sdu.dk/~petersk/DM537/");
Reader in = new InputStreamReader(url.openStream());
PrintStream output = new PrintStream(
    new FileOutputStream("dm537.html"));
BufferedReader input = new BufferedReader(in);
while (true) {
  String line = input.readLine();
  if (line == null) {  break;  }
  output.println(line);
}     input.close();   output.close();
```

TCP/IP Sockets

- URL provides high-level abstraction
- for general TCP/IP connection, *sockets* are needed
- once socket connection is established, normal byte streams
- client-server model where server waits for client to connect
- for sockets, IP adress and port number needed
- Example: IP 130.225.157.85, Port 80 (IMADA web server)

- listening sockets implemented by class ServerSocket
- Example: `ServerSocket ss = new ServerSocket(2342);`

- connection between client and server instance of Socket
- Example: `Socket sSock = ss.accept();`
  `Socket sock = new Socket("127.0.0.1", 2342);`

Example: TCP/IP Server

```
public class MyServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(2343);
        while (true) {
            Socket sock = server.accept();
            InputStream in = sock.getInputStream();
            OutputStream out = sock.getOutputStream();
            while (true) {
                int read = in.read();
                if (read == -1) {  break;  }
               out.write(Character.toUpperCase((char)read));
            }   }   }   }
```

Example: TCP/IP Client

```
public class MyClient {
    public static void main(String[] args) throws IOException {
        Socket sock = new Socket("127.0.0.1", 2343);
        InputStream in = sock.getInputStream();
        OutputStream out = sock.getOutputStream();
        String userInput = new Scanner(System.in).nextLine();
        StringBuffer result = new StringBuffer();
        for (char ch : userInput.toCharArray()) {
            out.write(ch);
            result.append((char)in.read());
        }
        System.out.println(result);   }   }
```

Example: Simple Chat Server

```
public class ChatServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(2343);
        while (true) {
            Socket sock = server.accept();
            Scanner in = new Scanner(sock.getInputStream());
            PrintStream out = new PrintStream(sock.getOutputStream());
            while (true) {
                System.out.println(in.nextLine());
                out.println(new Scanner(System.in).nextLine());
            }
        } } }
```

Example: Simple Chat Client

```
public class ChatClient {
    public static void main(String[] args) throws IOException {
        Socket sock = new Socket("127.0.0.1", 2343);
        Scanner in = new Scanner(sock.getInputStream());
        PrintStream out = new PrintStream(sock.getOutputStream());
        while (true) {
            out.println(new Scanner(System.in).nextLine());
            System.out.println(in.nextLine());
        }
    }
}
```

Theory and Practice

- our client-server implementations work fine
- BUT:
  - network connections are not reliable
  - there can be many clients
  - answering queries can be time consuming
- multi-threading can solve these problems
- Idea:
  - create a thread for each client connection
  - the server is immediately responsive
  - starving threads can be disposed of after some timeout

Multi-Threading

- threads can be started by creating instances of Thread
- Example (two threads counting up to 1 000 000):

```
public class Counter extends Thread {
    String name;
    public Counter(String name) {  this.name = name;  }
    public void run() {
        for (int i=1; i<=1000000; i++) {
            System.out.printf("%s: %d\n", name, i);
        }
    }
    …
```

Multi-Threading

- Example (continued):

```
…
public static void main(String[] args) {
    Counter c1 = new Counter("Fred");
    Counter c2 = new Counter("George");
    c1.start();
    c2.start();
}
}
```

- start() creates a new thread and runs the run() method

Multi-Threaded Server

```
public class MultiServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(2343);
        while (true) {
            Socket sock = server.accept();
            new MultiServerHandler(sock).start();
        }
    }
}
```

Multi-Threaded Server

```
public class MultiServerHandler extends Thread {
    private Socket sock;
    public MultiServerHandler(Socket sock) {
        this.sock = sock;
    }
    public void run() {
        try {
            Scanner in = new Scanner(sock.getInputStream());
            PrintStream out = new PrintStream(sock.getOutputStream());
            while (true) {  out.println(in.nextLine().toUpperCase());  }
        } catch (IOException e) {}
    }   }
```

THE END