ID2212 Network Programming with Java
Lecture 3

Java I/O.
Overview of New I/O (NIO)

Leif Lindbäck, Vladimir Vlassov

KTH/ICT/SCS

HT 2016

Outline

- **Java I/O**
  - I/O using Streams
  - Types of streams
  - Standard streams
  - Accessing files
  - File channels
- **Overview of New I/O**
  - Buffers
  - Channels
  - Selectors

I/O in Java

- Package `java.io`
- I/O sources and destinations:
  - standard input, standard output, standard err
  - Files, streams of TCP socket and URL connections
- Input and output streams
  - Java provides different types of stream APIs, e.g. byte streams, character streams, object streams, etc.
  - Different stream reading and writing primitives, e.g. read/write, print
  - Basic streams: byte streams
  - Other streams are built on top of byte streams

I/O in Java (cont’d)

- For example:

```java
try (BufferedReader r = new BufferedReader(
                    new InputStreamReader(
                    socket.getInputStream())) {
  String str;
  while ((str = r.readLine()) != null) {
    //process the line read
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

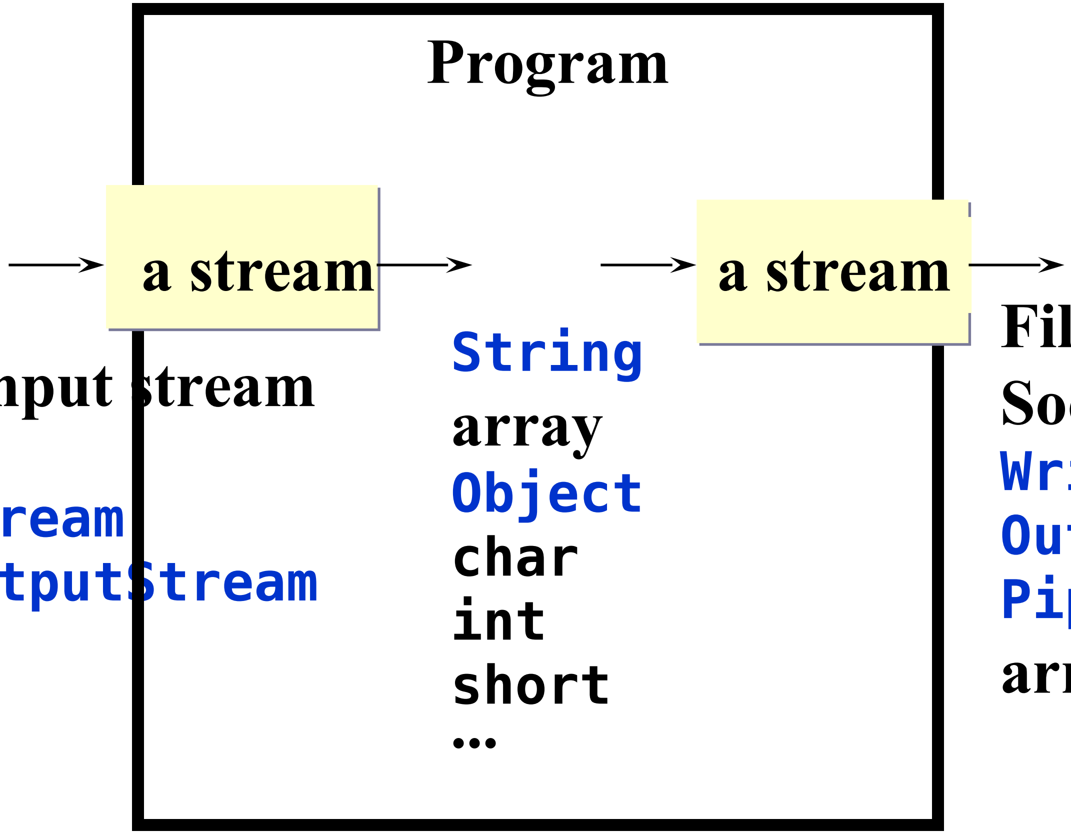
Streams

- *Streams* pass data from/to programs.
  - Input can be performed by different types of input streams, e.g. byte input stream, character input stream (reader)
  - Output can be performed by different types of output streams, e.g. byte output stream, character output stream (writer)
  - If a stream handles characters on the program side, then it is called a *reader* or a *writer*.

Streams

A source can be:

A destination can be:



Program

a stream

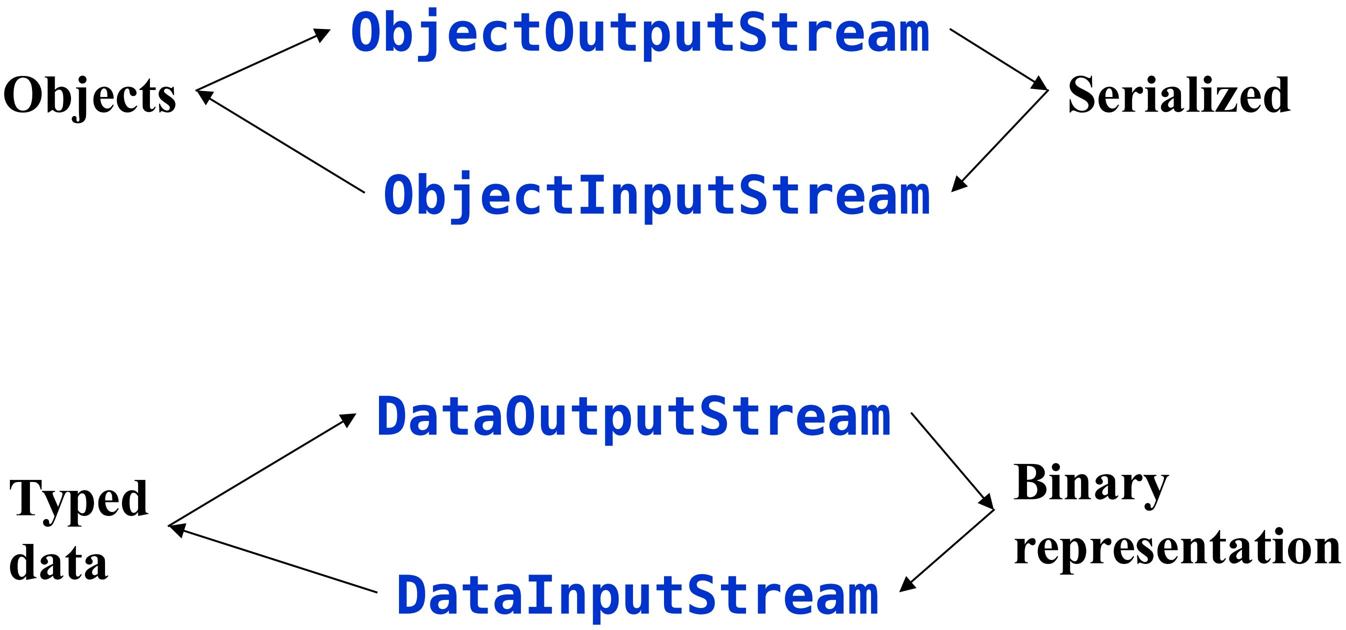
a stream

File
Socket input stream
Reader
InputStream
PipedOutputStream
array

String
array
Object
char
int
short
…

File
Socket output stream
Writer
OutputStream
PipedInputStream
array

Some Types of Streams



Objects

`ObjectOutputStream`

`ObjectInputStream`

Serialized

Typed data

`DataOutputStream`

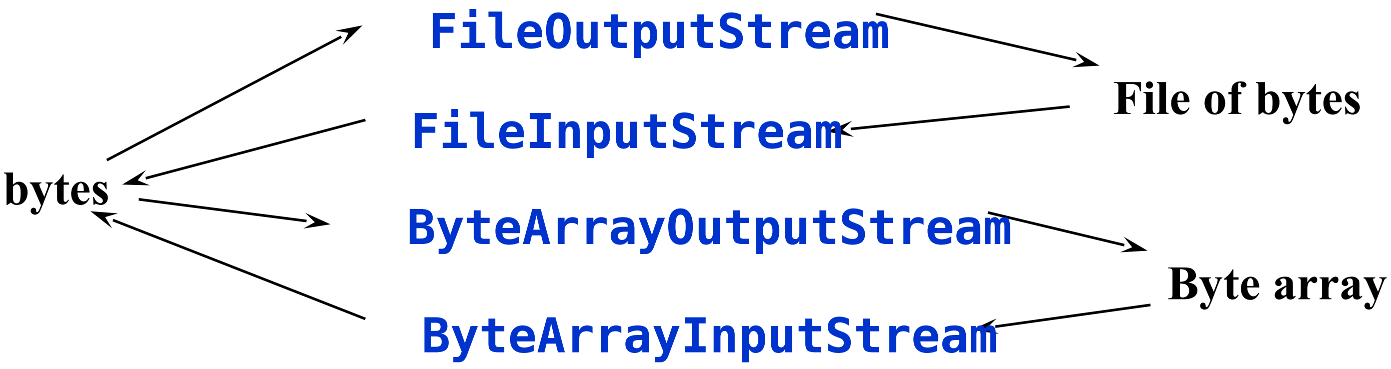
`DataInputStream`

Binary representation

DataInputStream Example

```
try (DataInputStream inData = new DataInputStream(
                              new FileInputStream(
                              fileName))) {
  while (true) {
    no = inData.readInt();
    System.out.println("No " + no);
  }
} catch (EOFException done) {
} catch (FileNotFoundException e) {
  System.err.println("file " + fileName +
                     " is mising");
} catch (IOException e) {
  e.printStackTrace();
}
```

Types of Streams (cont'd)



bytes

FileOutputStream

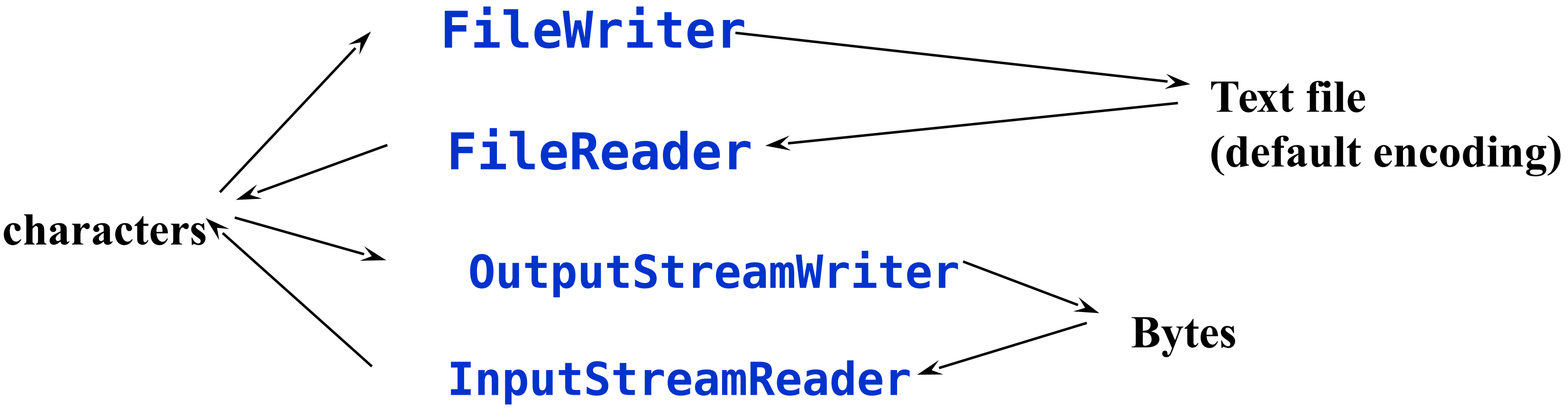
FileInputStream

ByteArrayOutputStream

ByteArrayInputStream

File of bytes

Byte array



characters

FileWriter

FileReader

OutputStreamWriter

InputStreamReader

Text file
(default encoding)

Bytes

Standard Streams

- Static fields in the `java.lang.System` class:
  - `public static final PrintStream err;`
    - The “standard” error output stream;
  - `public static final PrintStream out;`
    - The “standard” output stream;
  - `public static final InputStream in;`
    - The “standard” input stream.
  - All the streams are already open and ready to supply/accept data

```
System.out.println(“your output “ + result);
```

Files (java.io package)

- File class supports platform-independent usage of file- and directory names.
  - Instances of this class represent the name of a file or a directory on the host file system.
- Some constructors:

```
File(String path)
File(String dir, String fileName)
File(File dir, String fileName)
```

- Some interesting methods of File:

```
public boolean exists();
public boolean isDirectory();
public boolean isFile();
public long length();
public String[] list();
public String[] list(FileNameFilter f);
public boolean mkdir();
public boolean renameTo(File dest);
public boolean createNewFile()
```

File Streams

- Used to access files (for reading and writing) as a continues stream of bytes or characters
- `FileInputStream` and `FileOutputStream`
  - for reading and writing bytes to the file
- `FileReader` and `FileWriter`
  - for reading and writing character files
- Provide read and write methods
- Can be created by constructors given a file name or an object of `File`

```
FileInputStream inf = new FileInputStream(filename);
```

File Descriptor

- `FileDescriptor` class is a platform-independent representation of a handle of an open file or an open socket.
- Objects of this class
  - are returned by `getFD()` of `FileInputStream`, `FileOutputStream`, `RandomAcessFile`, ...
  - passed to (used by) `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`, ...

Random Access File

- `RandomAccessFile` class – provides an API similar to the file API in C
  - Instances of this class represent the file opened in a given mode, e.g.
    - “`r`” – for reading only
    - “`rw`”– for reading and writing
  - Methods of this class provide means for reading from file, writing into file and changing current file access position.
  - All methods (including constructors) of this class may throw `IOException`.
  - Contains object of the `FileDescriptor` class as a handle of the file.

An Overview of New I/O

Use of the new I/O API when performing course programming assignments is optional

New I/0 (`java.nio.`*...)

- **New I/O APIs introduced in JDK v 1.4**
- **NIO APIs supplements `java.io`**
  - **provides a new I/O model based on channels, buffers and selectors**
  - **enables non-blocking scalable I/O**
  - **allows improving performance of distributed applications (mostly for the server side)**

Features in NIO APIs

- ***Buffers*** **for data of primitive types, e.g. char, int**
- ***Channels*****, a new primitive I/O abstraction**
- ***A multiplexed, non-blocking I/O facility*** **(selectors, selection keys, selectable channels) for writing scalable servers**
- ***Character-set encoders and decoders***
- ***A pattern-matching facility*** **based on Perl-style regular expressions (`java.util`)**
- **A file interface that supports locks and memory mapping**

NIO Packages

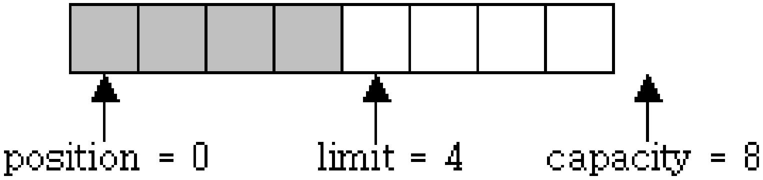
`java.nio`	Buffers, which are used throughout the NIO APIs.
`java.nio.channels`	Channels and selectors.
`java.nio.charset`	Character encodings.
`java.nio.channels.spi`	Service-provider classes for channels.
`java.nio.charset.spi`	Service-provider classes for charsets.
`java.util.regex`	Classes for matching character sequences against patterns specified by regular expressions.

NIO Programming Abstractions

- *Buffers*
  - Containers for data
  - Can be filled, drained, flipped, rewind, etc.
  - Can be written/read to/from a channel
- *Channels* of various types
  - Represent connections to entities capable of performing I/O operations, e.g. pipes, files and sockets
  - Can be selected when ready to perform I/O operation
- *Selectors* and *selection keys*
  - together with selectable channels define a multiplexed, non-blocking I/O facility. Used to select channels ready for I/O
- *Charsets* and their associated *decoders* and *encoders*
  - translate between bytes and Unicode characters

Buffers

- *Buffer* is a container for a fixed amount of data of a specific primitive type; Used by channels
  - Content, data
  - Capacity, size of buffer; set when the buffer is created; cannot be changed
  - Limit, the index of the first element that should not be read or written; limit ≤ capacity
  - Position, the index of the next element to be read or written
  - Mark, the index to which its position will be reset when the reset method is invoked
  - Buffer invariant: 0 ≤ mark ≤ position ≤ limit ≤ capacity



Buffer Classes

Buffer	Superclass for other buffers; clear, flip, rewind, mark/reset
ByteBuffer	provides views as other buffers, e.g. IntBuffer get/put, compact, views; allocate, wrap
MappedByteBuffer	Subclass of the ByteBuffer A byte buffer mapped to a file
CharBuffer DoubleBuffer FloatBuffer IntBuffer LongBuffer	absolute (index-based) and relative (position-based) get/put, compact, allocate, wrap

Some Buffer’s methods

`static allocateDirect()`	Allocates a new direct byte buffer. With direct `ByteBuffer`, JVM avoid intermediate buffering when performing native I/O operations directly upon the direct buffer.
`static allocate()`	allocate a buffer of a given capacity
`clear()`	clear the buffer, i.e. prepare the buffer for writing data into it by channel-reads or relative puts (limit = capacity; position = 0)
`flip()`	prepare the buffer for reading data from it by channel-writes or relative gets (limit = position; position = 0)
`rewind()`	prepare the buffer for re-reading data from it (position = 0)
`mark()`	set this buffer's mark at its position (mark = position)
`reset()`	reset this buffer's position to the previously-marked position (position = mark)

Some Buffer’s methods (cont’d)

static wrap()	wrap a given array into a buffer; returns the buffer.
get/put	absolute (index-based) and relative (position-based) get/put data from/into the buffer; position = position –/+ 1;
asIntBuffer() asCharBuffer()…	create a view of this byte buffer as another primitive type buffer, e.g. as an IntBuffer, as a CharBuffer, etc.
slice()	create a new buffer that shares part of this buffer's content starting at this buffer's position.
duplicate()	creates a new byte buffer that shares the this buffer's content.
compact()	copy data between position and limit to the beginning of the buffer; position is set to the number of data items copied.
boolean hasRemaining()	check whether there are any elements between the current position and the limit.

Creating Buffers

- Allocation
  - Create an empty buffer on top of a backing Java array

```
Bytebuffer buf1 = ByteBuffer.allocate(100);
IntBuffer buf2 = intBuffer.allocate(100);
```

- Direct allocation (only ByteBuffer)
  - Direct buffers (using DMA)

```
ByteBuffer buf3 = ByteBuffer.allocateDirect(100);
```

- Wrapping
  - Wrap a buffer around existing data array

```
byte[] data = “Some data”.getBytes(“UTF-8”);
ByteBuffer buf4 = ByteBuffer.wrap(data);
char[] text = “Some text”.toCharArray();
CharBuffer buf5 = CharBuffer.wrap(text);
```

Filling/Draining Buffers

- **Filling using wrap or put**

```
String s = ”Some String”;
CharBuffer buf1 = CharBuffer.wrap(s);
CharBuffer buf2 = CharBuffer.allocate(s.length());
// put reversed s in to buf2
for (int i = s.length() - 1; i >= 0; i--) {
  buf2.put(s.charAt(i)); // relative put
} // position in buf2 should be 11 after the loop
```

- **Draining using get**

```
buf2.flip(); // limit = position; position = 0
String r = ””;
while (buf2.hasRemaining())
    r += buf2.get();
}
```

Reading/Writing Buffers from/to Channels

- Reading from a channel to a buffer

```
while (buf.hasRemaining() && channel.read(buf) != -1)
  {
  // process the buffer’s content
}
```

- Writing to a channel from a buffer

```
while (buf.hasRemaining() &&
       channel.write(buf) != -1) ;
```

Channels

- ***Channels*** **represent connections to various I/O sources, such as pipes, sockets, files, datagrams;**
  - **operate with buffers and I/O sources: move (read/write) data blocks into / out of buffers from / to the I/O sources;**
  - **can be open or closed;**
  - **can be blocking/non-blocking, selectable (socket, pipe), interruptible (file);**
  - **enable *non-blocking I/O operations***

Channels versus Streams

Channels (new I/O)	Streams (traditional I/O)
Write/read data to/from buffers; similar to buffered streams; buffers can be directly allocated in memory – efficient implementation	Write data onto output streams and reading data from input streams
Block-based: a stream of blocks from/to buffers	Byte-based: a continues stream of bytes
Bi-directional: tend to support both reading and writing on the same object (source, buffer)	Uni-directional: input streams and output streams

Some Channel Classes

- For TCP connections
  - `SocketChannel`
  - `ServerSocketChannel`
- For UDP communication
  - `DatagramChannel`
- For file access
  - `FileChannel`

FileChannel

- `java.nio.channels.FileChannel`
  - A channel for reading, writing, mapping, and manipulating a file.
  - Similar to `RandomAccessFile`
- Can be mapped to a buffer in the main memory
  - `MappedByteBuffer()`
- Has a current position within its file which can be both queried and modified.
- The file itself contains a variable-length sequence of bytes that can be read and written and whose current size can be queried.

Some methods of FileChannel

`read (dst, pos)`
`write (src, pos)`
	Read or write at an absolute position in a file without affecting the channel's position.

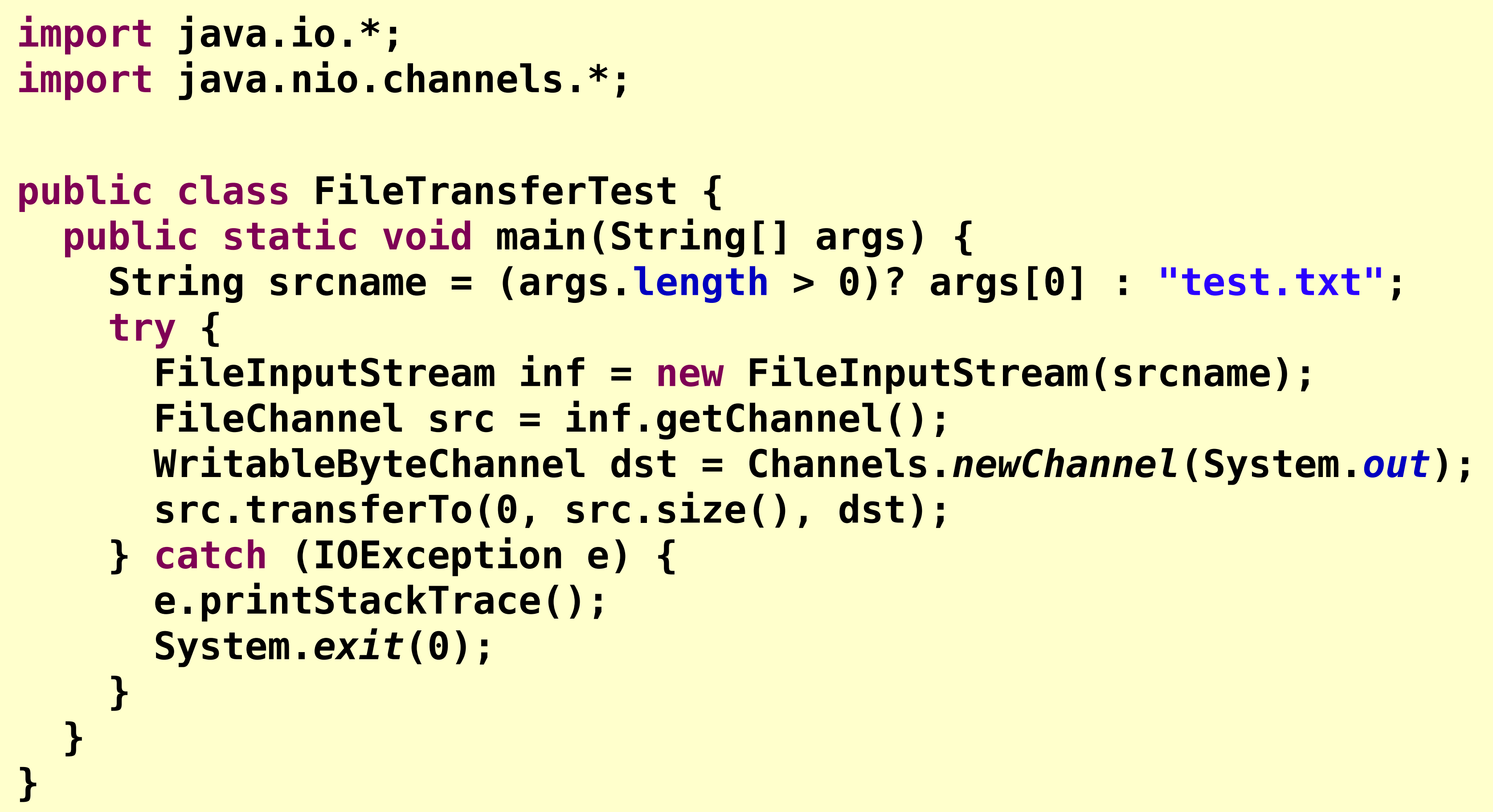
`MappedByteBuffer()`	Map a region of a file directly into memory.
`force()`	Force out file updates to the underlying storage device, in order to ensure that data are not lost in the event of a system crash.

`transferTo()`
`transferFrom()`
	Bytes can be transferred from a file to some other channel, and vice versa, in a way that can be optimized by many OSs into a very fast transfer directly to or from the file system cache.

FileChannel Example

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class FileChannelTest {
  public static void main(String[] args) {
    String filename = (args.length > 0)? args[0] : "test.txt";
    try {
      FileInputStream inf = new FileInputStream(filename);
      FileChannel channel = inf.getChannel();
      MappedByteBuffer buffer =
          channel.map(FileChannel.MapMode.READ_ONLY,
                      0, channel.size());
      WritableByteChannel out = Channels.newChannel(System.out);
      while (buffer.hasRemaining() && out.write(buffer) != -1) {
        System.out.println("Writing the file " + filename);
      }
      channel.close();
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

Using transfer method



```java
import java.io.*;
import java.nio.channels.*;

public class FileTransferTest {
  public static void main(String[] args) {
    String srcname = (args.length > 0)? args[0] : "test.txt";
    try {
      FileInputStream inf = new FileInputStream(srcname);
      FileChannel src = inf.getChannel();
      WritableByteChannel dst = Channels.newChannel(System.out);
      src.transferTo(0, src.size(), dst);
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

SocketChannel

- *A selectable channel* for stream-oriented *connecting sockets*.
  - Reads from and writes to a TCP socket.
  - Uses `ByteBuffer` for reading and writing
  - Does not have public constructors
- Each `SocketChannel` is associated with a peer `Socket` object
  - Binding, closing, and manipulation of socket options must be done through the associated Socket object

```
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.connect(new InetSocketAddress(host,
                                     port));
```

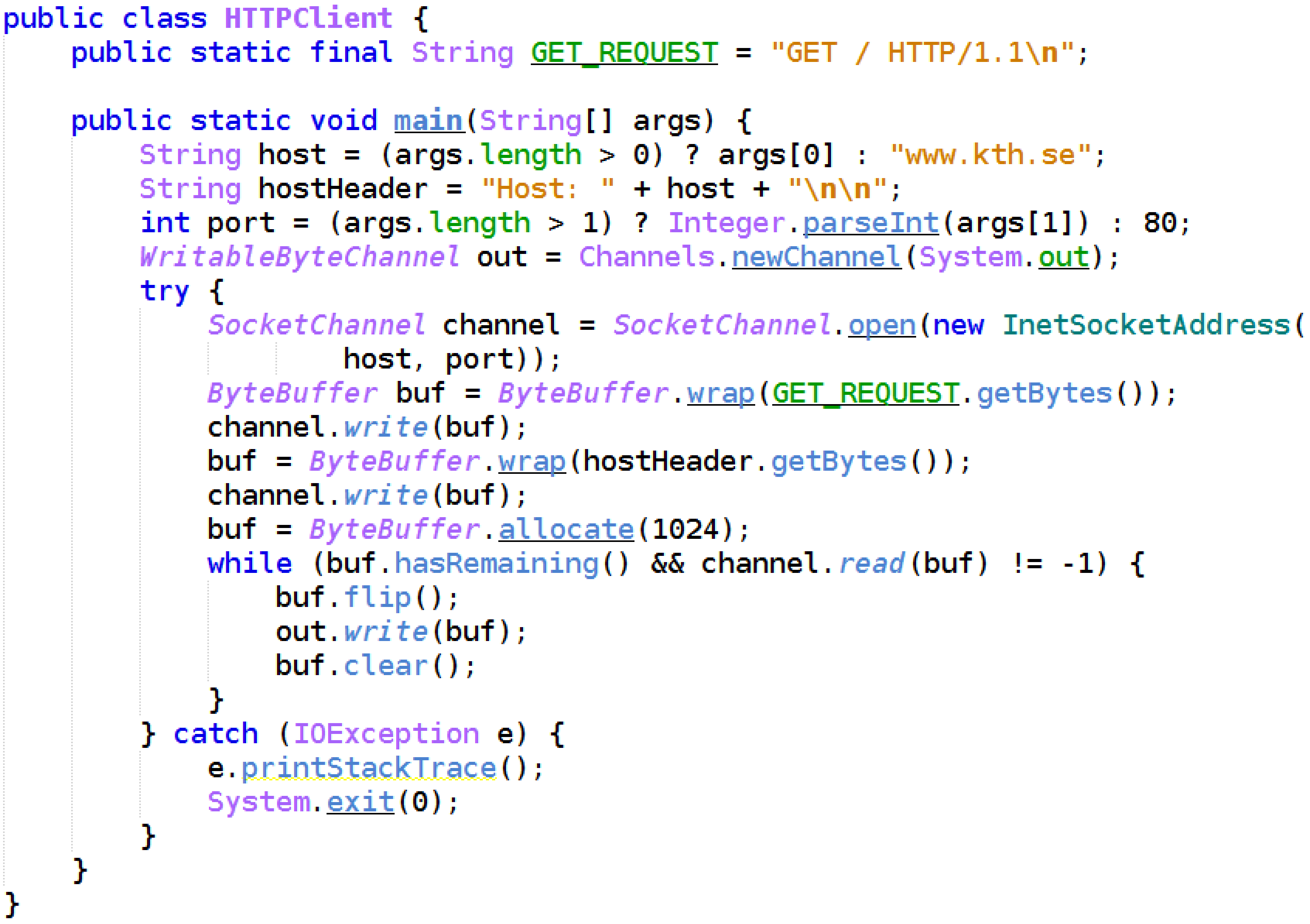
SocketChannel Example 1

```java
import java.io.IOException;
import java.nio.channels.*;
import java.net.*;

public class SocketChannelTest {
  public static void main(String[] args) {
    String host = (args.length > 0)? args[0] : "www.oracle.com";
    int port = (args.length > 1) ? Integer.parseInt(args[1]) : 80;
    try {
      SocketChannel channel = SocketChannel.open();
      channel.configureBlocking(false);
      channel.connect(new InetSocketAddress(host, port));
      //can do something here while connecting
      while (!channel.finishConnect()) {
        System.out.println("Connecting to " + host + " on port " + port);
        // can do something here while connecting
      }
      System.out.println("Connected to " + host + " on port " + port);
      // communication with the server via channel
      channel.close();
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

Example 2

```
import java.io.IOException;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;

public class HTTPClient {
  public static final String GET_REQUEST = "GET /index.html HTTP/1.0\n\n";
  public static void main(String[] args) {
    String host = (args.length > 0) ? args[0] : "www.oracle.com";
    int port = (args.length > 1) ? Integer.parseInt(args[1]) : 80;
    WritableByteChannel out = Channels.newChannel(System.out);
    try {
      SocketChannel channel = SocketChannel.open(new InetSocketAddress(
                                                 host, port));
      ByteBuffer buf = ByteBuffer.wrap(GET_REQUEST.getBytes());
      channel.write(buf);
      buf = ByteBuffer.allocate(1024);
      while (buf.hasRemaining() && channel.read(buf) != -1) {
        buf.flip();
        out.write(buf);
        buf.clear();
      }
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

Example 2



```
public class HTTPClient {
    public static final String GET_REQUEST = "GET / HTTP/1.1\n";

    public static void main(String[] args) {
        String host = (args.length > 0) ? args[0] : "www.kth.se";
        String hostHeader = "Host: " + host + "\n\n";
        int port = (args.length > 1) ? Integer.parseInt(args[1]) : 80;
        WritableByteChannel out = Channels.newChannel(System.out);
        try {
            SocketChannel channel = SocketChannel.open(new InetSocketAddress(
                    host, port));
            ByteBuffer buf = ByteBuffer.wrap(GET_REQUEST.getBytes());
            channel.write(buf);
            buf = ByteBuffer.wrap(hostHeader.getBytes());
            channel.write(buf);
            buf = ByteBuffer.allocate(1024);
            while (buf.hasRemaining() && channel.read(buf) != -1) {
                buf.flip();
                out.write(buf);
                buf.clear();
            }
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(0);
        }
    }
}
```

ServerSocketChannel

- *A selectable channel* **for stream-oriented** *listening sockets.*
  - Abstraction for listening network sockets.
  - Listens to a port for TCP connections.
  - Does not have public constructors
- Each `ServerSocketChannel` is associated with a peer `ServerSocket` object
  - Binding and the manipulation of socket options must be done through the associated `ServerSocket` object;
- `accept` on a ready `ServerSocketChannel` returns `SocketChannel`

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
ServerSocket socket = serverChannel.socket();
socket.bind(new InetSocketAddress(port));
serverChannel.configureBlocking(false);
selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

Selectors

- *Selector* is an object used to select a channel ready to communicate (to perform an operation)
  - Used to operate with several non-blocking channels
  - Allows readiness selection
    - Ability to choose a selectable channel that is ready for some of network operation, e.g. accept, write, read, connect

Selectable Channels

- *Selectable channels* include:
  - `DatagramChannel`
  - `Pipe.SinkChannel`
  - `Pipe.SourceChannel`
  - `ServerSocketChannel`
  - `SocketChannel`
- Channels are registered with a selector for specific operations, e.g. accept, read, write
- Registration is represented by a *selection key*

Selection Keys

- **A selector operates with set of selection keys**
- ***Selection key*** **is a token representing the registration of a channel with a selector**
- **The selector maintains three sets of keys**
  - ***Key set*** **contains the keys with registered channels;**
  - ***Selected-key set*** **contains the keys with channels ready for at least one of the operations;**
  - ***Cancelled-key set*** **contains cancelled keys whose channels have not yet been deregistered.**
  - **The last two sets are sub-sets of the Key set.**

Use of Selectors

- Create a selector

```
 Selector selector = Selector.open();
```

- Configure a channel to be non-blocking

```
channel.configureBlocking(false);
```

- Register a channel with the selector for specified operations (accept, connect, read, write)

```
 ServerSocketChannel serverChannel =
                    ServerSocketChannel.open();
 ServerSocket serverSocket = serverChannel.socket();
 serverSocket.bind(new InetSocketAddress(port));
 serverChannel.configureBlocking(false);
 serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

  - Register as many channels as you have/need

Use of Selectors (cont'd)

- `Select()` on the selector to perform the selection of keys with ready channels
  - Selects a set of keys whose channels are ready for I/O.
- `selectNow()` – non-blocking select: returns zero if no channels are ready
- `selectedKeys()` on the selector to get the selected-key set
- Iterate over the selected-key set and handle the channels ready for different I/O operations, e.g. read, write, accept

SelectionKey

- Upon registration, each of the registered channels is assigned a selection key.

```
SelectionKey clientKey = clientChannel.register(selector,
  SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

- Selection key allows attaching of a single arbitrary object to it
  - Associate application data (e.g. buffer, state) with the key (channel)

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
clientKey.attach(buffer);
```

- Get the channel and attachment from the key

```
SocketChannel clientChannel =
                    (SocketChannel) key.channel();
ByteBuffer buffer = (ByteBuffer) key.attachment();
```

Non-Blocking Server

```
while (true) {
  selector.select();
  Iterator<SelectionKey> keys = selector.selectedKeys().iterator();

  while (keys.hasNext()) {
    SelectionKey key = keys.next();
    keys.remove();

    if (key.isAcceptable()) { // accept connection.
      ServerSocketChannel server =
          (ServerSocketChannel) key.channel();
      SocketChannel channel = server.accept();
      channel.configureBlocking(false);
      channel.register(selector, SelectionKey.OP_READ,
                       ByteBuffer.allocate(1024));

    } else if (key.isReadable()) {  // read from a channel.
      SocketChannel channel = (SocketChannel) key.channel();
      ByteBuffer buffer = (ByteBuffer) key.attachment();
      channel.read(buffer);
      key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

Non-Blocking Server, Cont'd

```
    } else if (key.isWritable()) {  // write buffer to channel.
      SocketChannel channel = (SocketChannel) key.channel();
      ByteBuffer buffer = (ByteBuffer) key.attachment();
      buffer.flip();
      channel.write(buffer);
      if (buffer.hasRemaining()) {
        buffer.compact();
      } else {
        buffer.clear();
      }
      key.interestOps(SelectionKey.OP_READ);
    }
  }
}
```