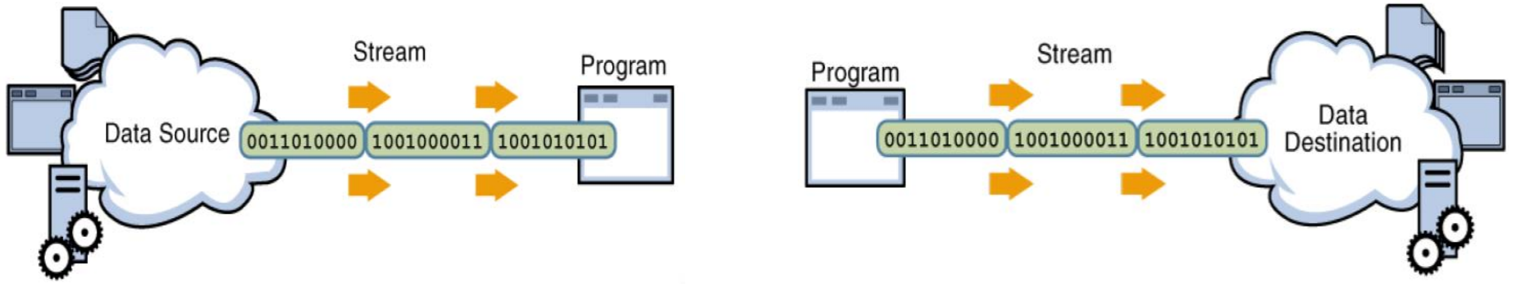
Architecture of software systems

Course 7: Streams, serialization, externalization, network communication

David Šišlák
david.sislak@fel.cvut.cz

Streams

» represents an input source or an output destination



» represent different kinds of data sources
  - » disk files, devices, other programs, network connection, memory arrays

» support different kinds of data
  - » bytes, primitives, localized characters, objects

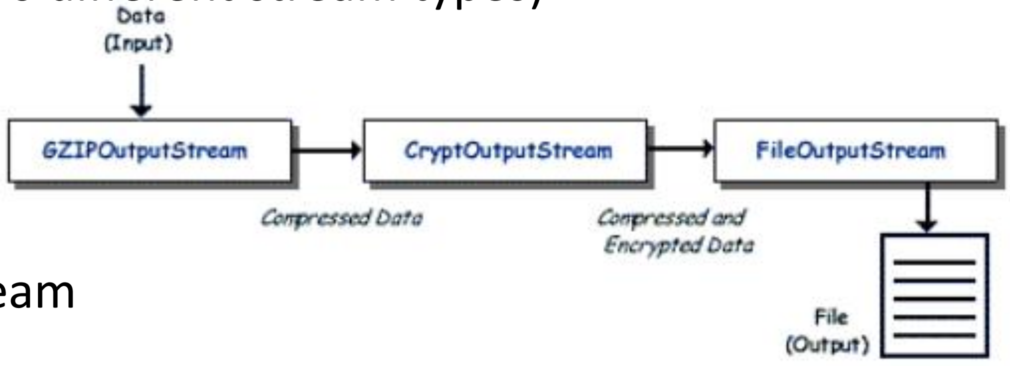
» can just pass data or manipulate/transform in useful ways

» use simple model for usage
  - » sequence of data elements

» streams can be chained

Streams

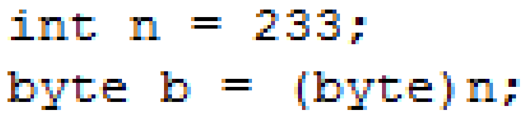
- types
  - byte vs. character
  - input vs. output
    - source or destination
  - node vs. filter (processing)
    - reading/writing from a specific location like files, memory, pipes
    - or transformation, managing data in the stream
- typical layered usage (more than 60 different stream types)
  - one node stream
  - chained with several filter/processing streams
  - user manipulates with top stream
- system streams (console I/O)
  - System.in – is instance of InputStream
  - System.out and System.out – is instance of PrintStream



Streams – byte stream

- java.io defines two basic root abstract stream classes for byte streams (8-bit values)
  - InputStream
    - int read(), int read(byte b[]), int read(byte b[], int off, int len)
    - long skip(long n)
    - int available()
    - close()
    - boolean markSupported(), mark(int readlimit), reset()
  - OutputStream
    - write(int b), write(byte b[]), write(byte b[], int off, int len)
    - flush() – force buffered output to be written
    - close()

Streams – byte stream



```
int n = 233;
byte b = (byte)n;
```

» is n = b ?

Streams – byte stream

```
int n = 233;
byte b = (byte)n;
```

- is n = b ?

  NO

- byte is signed 8-bit type with values from -128 to 127
- sign bit can be set even if original value is not negative

```
int n = 233; //binary 00000000 00000000 00000000 11101001
byte b = (byte)n; //binary 11101001, sign bit is set
```

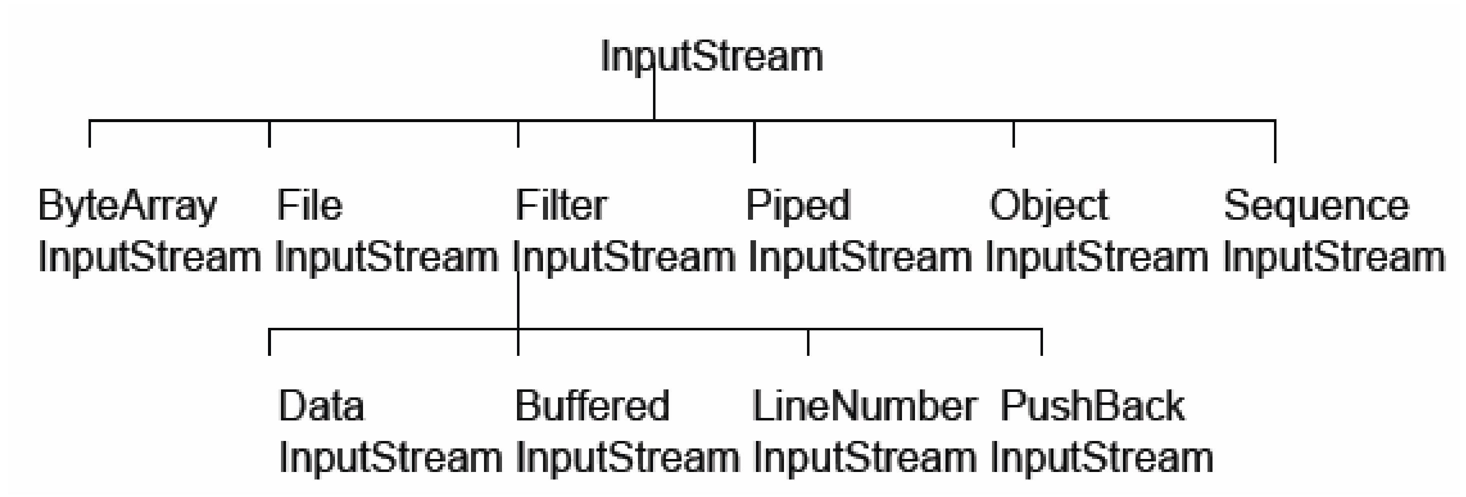
- read/write use **int** to allow signal -1 (EOF)
- reader should test value and if not -1 then it should cast to a byte !

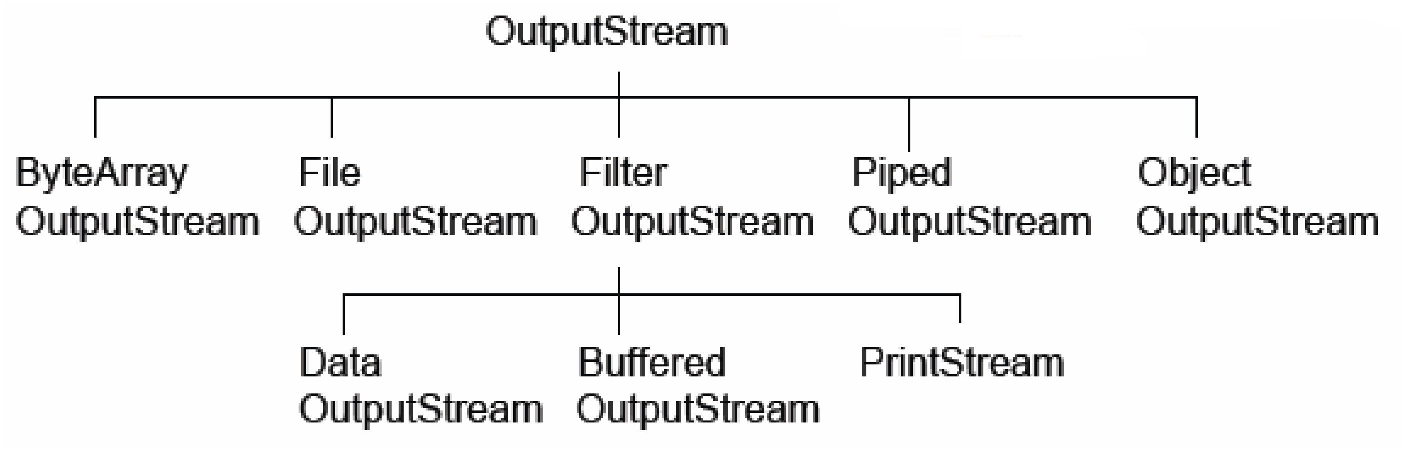
Streams – character stream

- character streams (similar methods but works with 16-bit chars) two root abstract classes
  - Reader
    - int read(), int read(char c[]), int read(char c[], int off, int len)
    - int read (CharBuffer b)
    - long skip(long n), close()
    - boolean markSupported(), mark(int readlimit), reset()
  - Writer
    - write(int c), write(char c[]), write(char c[], int off, int len)
    - write(String s), write(String s, int off, int len)
    - Writer append(char c), two append with CharSequence
    - flush(), close()

- bridge from byte stream to character streams – do character translation
  - java.io.InputStreamReader
  - java.io.OutputStreamWriter

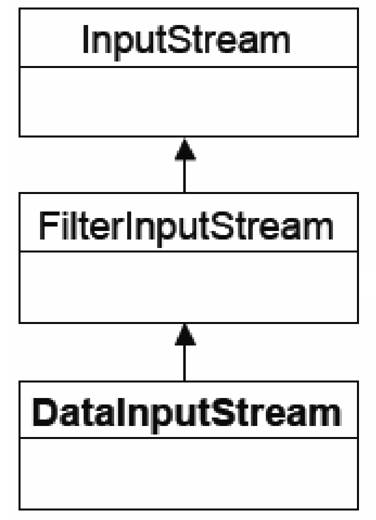
Streams – class hierarchy

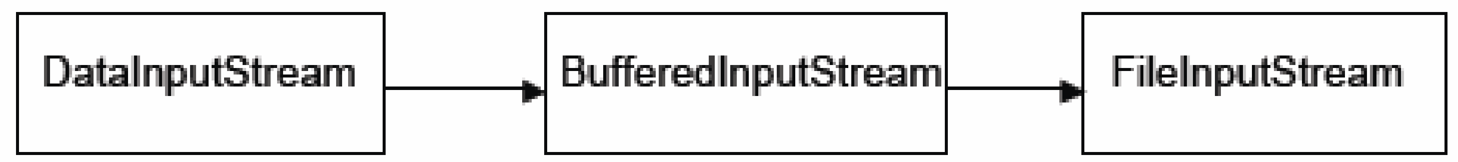




Streams

- each class has very focused responsibility
  - you need combine several streams together (through constructor)
    - decorator (wrapping idiom) pattern is used
  - e.g. FileInputStream with DataInputStream, usage of buffered stream





Streams – file example

```
InputStream myIn = new FileInputStream("input.bin");
boolean done = false;

while (!done) {
    int next = myIn.read();
    if (next == -1) {
        done = true;
    } else {
        byte b = (byte)next;
        // process input…
    }
}
myIn.close();
```

Streams – file example 2

```
FileReader in = new FileReader("in.txt");
FileWriter out = new FileWriter("out.txt");

BufferedReader inputStream = new BufferedReader(in);
PrintWriter outputStream = new PrintWriter(out);

String l;
while ((l = inputStream.readLine()) != null) {
    System.out.println(l);
    outputStream.println(l);
}
in.close();
out.close();
```

Serialization

- usefull for
  - persisting object graphs – all members to disk or database
  - network transmission
  - other – e.g. compute object signature
- key classes:
  - java.io.Serializable (no method definitions, only marker)
  - ObjectInputStream
  - ObjectOutputStream
- class implementing Serializable is required to has no-arg constructor
  - error thrown at runtime
- produce special binary stream
  - serialization uses reflection of all non-static members except **transient**
  - class definition is not saved !!!
  - store field names
- constructor and members can be also private, sub-classes requires protected/public constructors

Serialization

- all subclasses are automatically Serializable
  - it requires accessible no-arg constructor in parent
- non-serializable class can be made serializable in any sub-type
  - but there has to be accessible no-arg constructor
  - data from parent are not automatically serialized !
- identification of non-serializable object when traversing a graph
  - NotSerializableException

```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("test.dat"));
out.writeObject(serializableObject);
out.close();
```

```
ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("test.dat"));
serializableObject = in.readObject();
in.close();
```

Serialization

- each Seraializable class has

```
private static final long serialVersionUID = 7106358172580524456L;
```

  - generated based on class name, modifiers, interfaces, methods, etc.
- BEWARE of changes of class definitions
  - InvalidClassException – different serialVersionUID !
- define own serialVersionUID using **serialver** tool

- define serialization fields – can be used for evolving objects
  - non-transient and non-static
  - serialPersistentFields (ObjectStreamField[])
    - suitable for compatibility with old versions

```
private final static ObjectStreamField[] serialPersistentFields = {
  new ObjectStreamField("numberPrimitive", Integer.TYPE),
  new ObjectStreamField("doubleObject", Double.class),
  new ObjectStreamField("myObject", Test.class)
};
```

Serialization

- special handling of classes (exact signature)
  - additional information
  - initialization of non-serialized fields
  - solve incompatibility of versions

private void writeObject(ObjectOutputStream out) throws IOException

- can call out.defaultWriteObject (default nebo serialPersistentFields)

private void readObject(ObjectInputStream in) throws IOException

- can call in.defaultReadObject (default nebo serialPersistentFields)

private void readObjectNoData() throws ObjectStreamException

- given class is not listed as a superclass of deserialized object
- receiver’s version extends classes that are not extended by the sender’s version

  - anyway serialization continue with superclass serialization automatically

Serialization

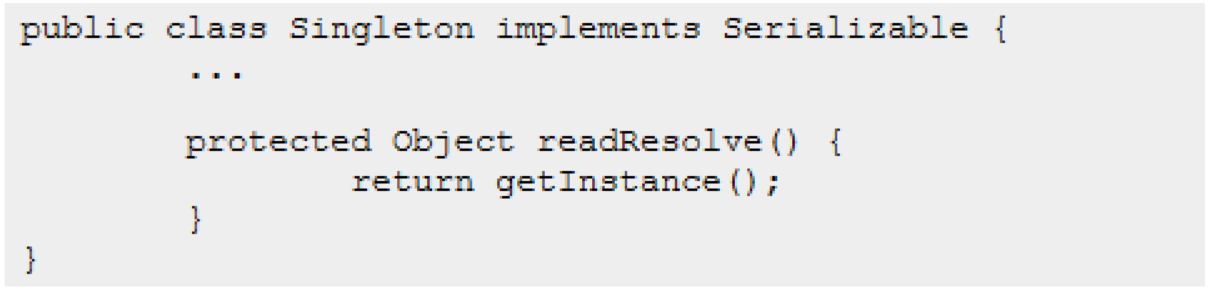
» use alternative objects

ANY-MODIFIER Object writeReplace() throws ObjectStreamException

- serialize different object than this

ANY-MODIFIER Object readResolve() throws ObjectStreamException

- after deserialization the object is replaced



```
public class Singleton implements Serializable {
        ...
        protected Object readResolve() {
                return getInstance();
        }
}
```

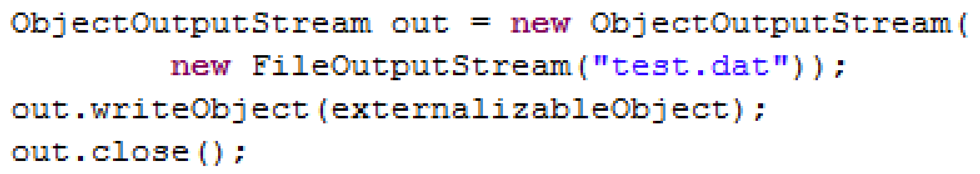
Externalizable

» faster than Serialization
» usually produce shorter binary stream
» control object graph traversal, but what about repeating objects?
» but you loose flexibility, add more bugs, class object is usually longer

» Externalization doesn’t continue with superclass serialization automatically!
» requires public no-arg constructor

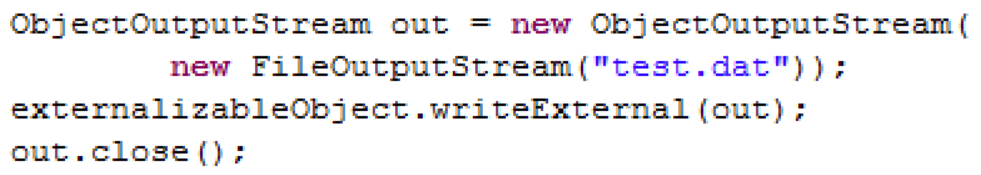
  public void writeExternal(ObjectOutput out) throws IOException

  public void readExternal(ObjectInput in) throws IOException



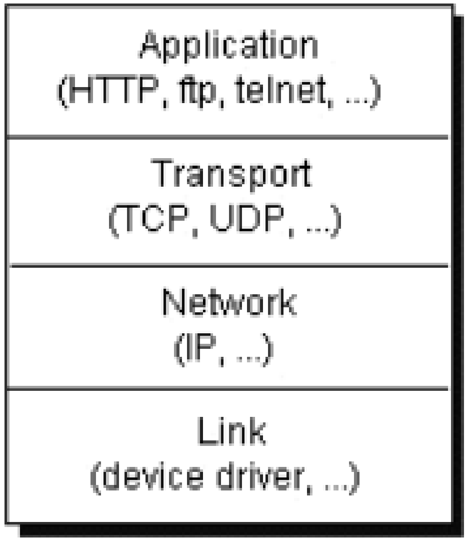
```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("test.dat"));
out.writeObject(externalizableObject);
out.close();
```

VS.



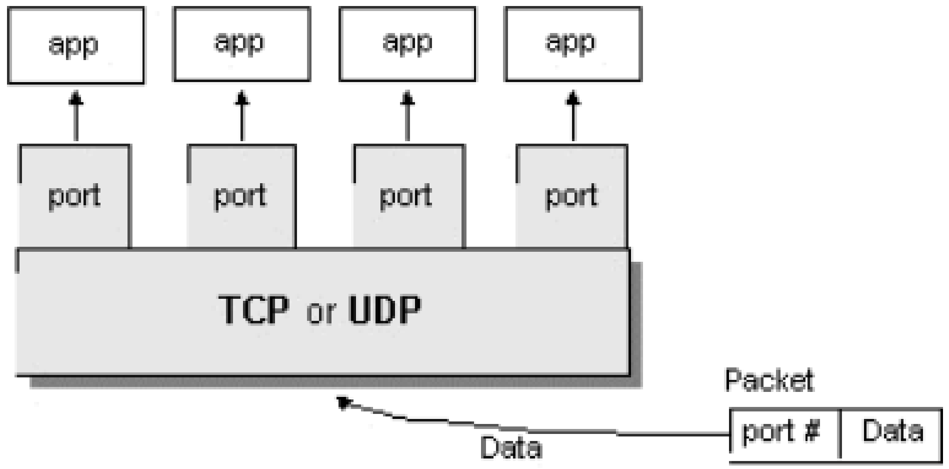
```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("test.dat"));
externalizableObject.writeExternal(out);
out.close();
```

Network communication - introduction



- » UDP/IP (User Datagram Protocol)
  - » datagram (packet) oriented
  - » order, delivery is not guaranteed
- » TCP/IP (Transmission Control Protocol)
  - » connection-based protocol
  - » reliable bi-directional point-to-point channel

Network communication - introduction



- ports – 16-bit number
- IPv4
    - IP – 32-bit address
- IPv6
    - IP – 128-bit address (64-bit site, 64-bit host)

Network communication – JAVA mapping

- java.net package
- addressing
  - InetAddress, InetSocketAddress
- UDP
  - DatagramPacket
  - DatagramSocket
  - MulticastSocket
- TCP
  - URL
  - URLConnection, HttpURLConnection
  - Socket
  - ServerSocket

JAVA networking

- InetAddress
  - get by name - InetAddress InetAddress.getByName(“google.com”)
  - get by address - InetAddress InetAddress.getByAddress(byte ip[])
  - get special - InetAddress InetAddress.getLocalHost()
- InetSocketAddress
  - IP with port – complete address
  - new InetSocketAddress(ia, port)
  - InetSocketAddress.createUnresolved(“www.google.com”, 80)
  - nonspecified address, automatic port – new InetSocketAddress(0)
- NetworkInterface
  - NetworkInterface.getAll(), NetworkInterface.getByName(“eth0”)
  - methods
    - getDisplayName(), getHardwareAddress(), getInetAddresses()

JAVA networking - URL

- URL (java.net.URL) – Uniform Resource Locator
  - protocol – most used http(s), ftp
  - host – DNS name, IP
  - port
  - file

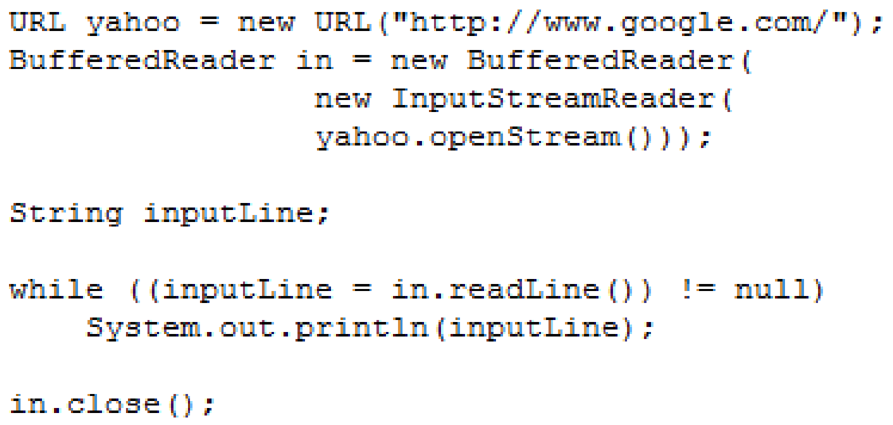
    http://www.google.com/search?q=a
  - support creation (also relative from other), getters for different parts

  - direct reading

    InputStream url.openStream()

    or

    Object url.getContent()



```
URL yahoo = new URL("http://www.google.com/");
BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                        yahoo.openStream()));

String inputLine;

while ((inputLine = in.readLine()) != null)
    System.out.println(inputLine);

in.close();
```

JAVA networking – URL connection

- » URLConnection
  - » URL Connection url.openConnection()
  - » can set timeouts, request properties, set input (POST data)
  - » can read content type and other parameters
  - » HttpURLConnection – connect(), getInputStream, getOutputStream

```
String paramEnc = URLEncoder.encode(param, "UTF-8");

URL url = new URL(where);
URLConnection connection = url.openConnection();
connection.setDoOutput(true);

OutputStreamWriter out = new OutputStreamWriter(
                              connection.getOutputStream());
out.write("param=" + paramEnc);
out.close();

BufferedReader in = new BufferedReader(
                new InputStreamReader(
                connection.getInputStream()));

String result;
while ((result = in.readLine()) != null) {
    System.out.println(result);
}
in.close();
```

JAVA networking – Socket

- Socket
  - end-point of network TCP/IP connection
  - is bound to particular IP and port
  - each TCP/IP connection is uniquely identified by its two end-points
  - provides input/output streams

```
Socket echoSocket = null;
PrintWriter out = null;
BufferedReader in = null;

try {
    echoSocket = new Socket("taranis", 7);
    out = new PrintWriter(echoSocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(
                                echoSocket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Don't know about host: taranis.");
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for "
                       + "the connection to: taranis.");
    System.exit(1);
}
```

JAVA networking – Socket

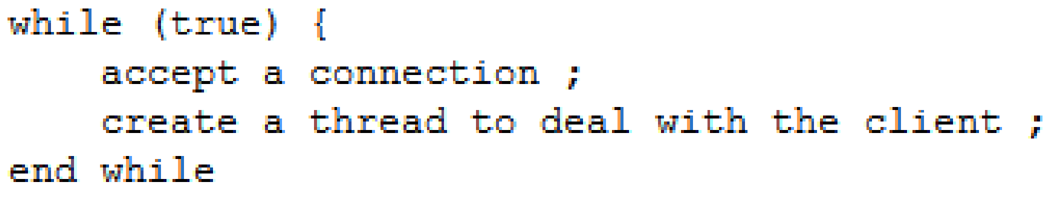
```
BufferedReader stdIn = new BufferedReader(
                                 new InputStreamReader(System.in));
String userInput;

while ((userInput = stdIn.readLine()) != null) {
     out.println(userInput);
     System.out.println("echo: " + in.readLine());
}

out.close();
in.close();
stdIn.close();
echoSocket.close();
```

JAVA networking – ServerSocket

- ServerSocket
  - special socket representing listening TCP/IP end-point
  - within constructor you specify the port, and optionally IP where it has to be bound
  - wait for establishing connection using method

    Socket accept()

- handle multiple clients



```
while (true) {
    accept a connection ;
    create a thread to deal with the client ;
end while
```

JAVA networking – DatagramPacket, DatagramSocket

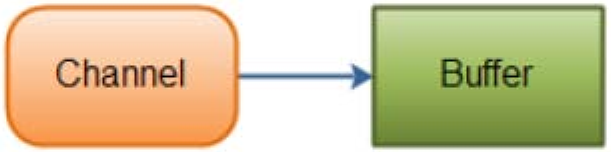
- DatagramPacket
  - independent, self-contained message sent over the network
  - like packet
    - InetAddress address, int port – destination
    - byte data[], int length, int offset
    - SocketAddress sa – sender
- DatagramSocket
  - sending or receiving point for a packet delivery service
  - can be bound to any available port (using default constructor)
  - connect(InetAddress,int) – can sent or receive packets only specified host, if not set in DatagramPacket automatically fill
  - send(DatagramPacket p), receive(DatagramPacket p) – blocking IO
- MulticastSocket
  - additional capabilities for joining/leaving multicast groups, loopback
  - multicast IP (IGMP – Internet Group Management Protocol)

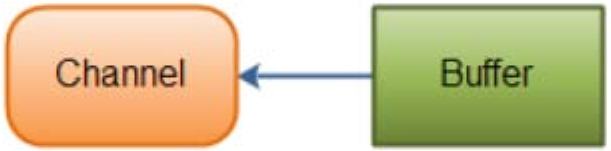
    224.0.0.0 – 239.255.255.255

JAVA networking – NIO

- NIO – new IO – implemented in java.nio starting from Java 1.4
- API for
  - scalable I/O – asynchronous I/O requests and polling
  - high-speed block-oriented binary and character I/O working – including mapping files to the memory, using channels and selectors
  - regular expressions
  - charset conversion
  - improved files ystem interface

- some functions are dependent on the underlying OS
- Channel is like a bit stream

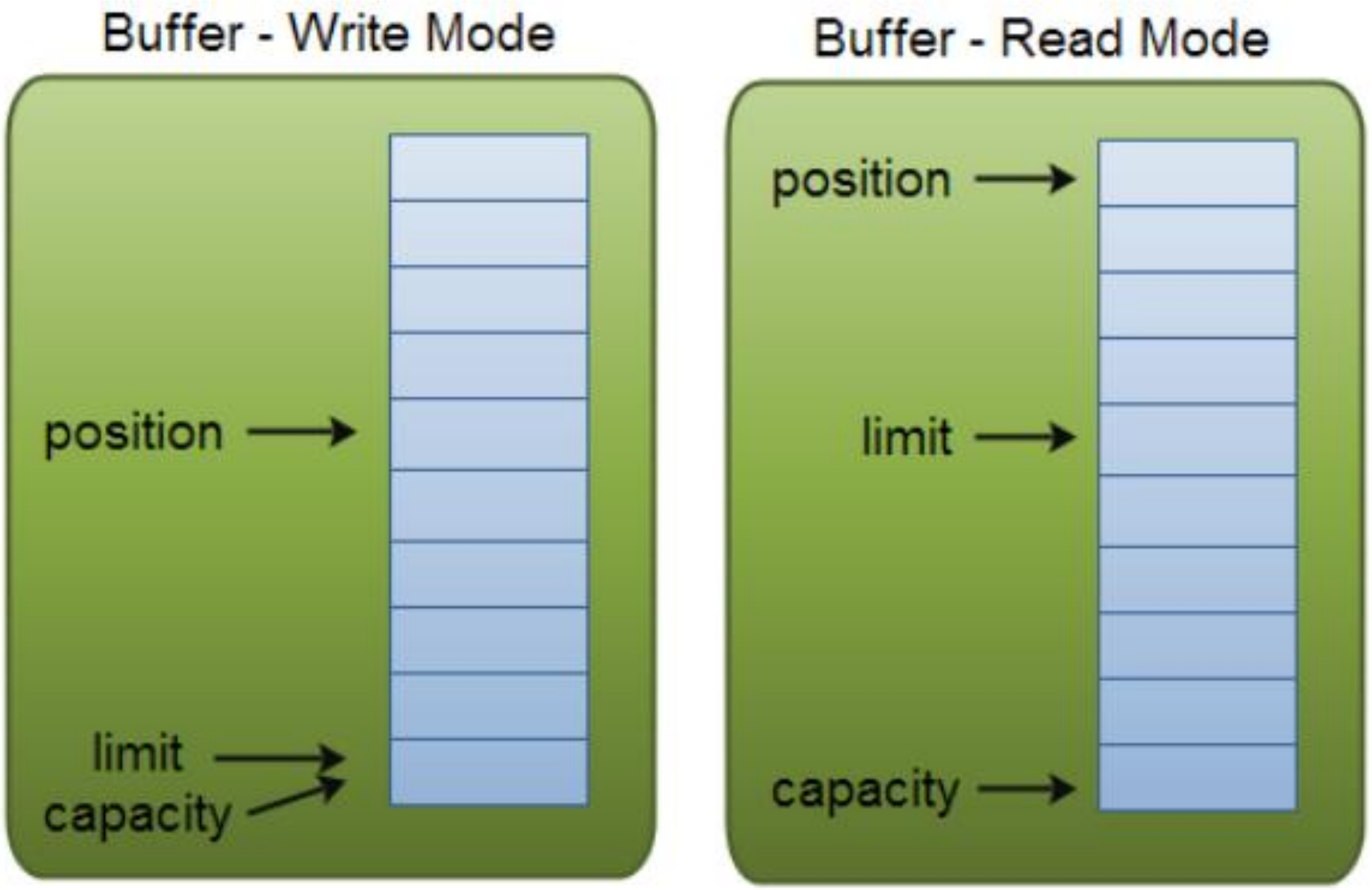




JAVA networking – NIO - Buffer

- java.nio.Buffer
  - linear, finite sequence of elements of a specific primitive type
    - ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer, MappedByteBuffer {FileChannel.map(…)}
  - not thread safe, multi mode for the same buffer (read, write)
  - key properties – 0 <= mark <= position <= limit <= capacity
    - capacity – numbers of elements, never changing !
    - limit – index of the first element that should not be read or written
    - position – index of the next element to be read or written
    - mark – index to which its position is set after reset()
  - initial content is undefined !!!
  - clear() – position=0, limit=capacity => ready for channel read (put)
  - flip() – limit=position, position=0 => ready for channel write (get)
  - rewind() – limit unchanged, position=0 => ready for re-reading
  - mark() – mark = position
  - reset() – position=mark

JAVA networking – NIO - Buffer



» write mode – channel.read(buf); buf.put(…);
» read mode – channel.write(buf); … buf.get();

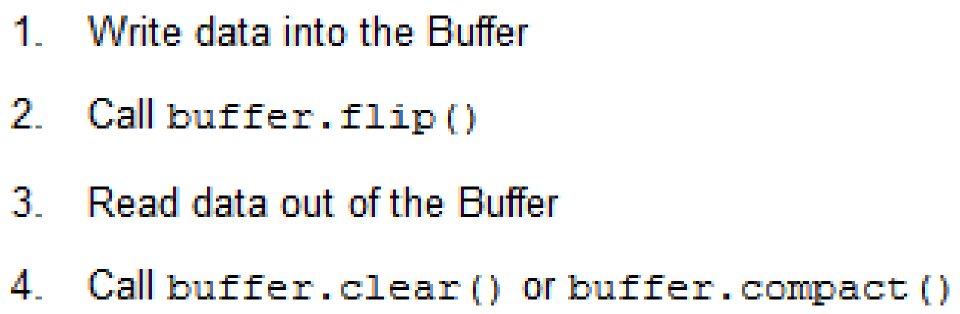
JAVA networking – NIO - Buffer

- java.nio.Buffer
  - isReadOnly() – can be read-only
  - hasArray() – is backed by an accessible array (array())
  - equals(), compareTo() – compare remainder sequence

  - can be allocated to physical memory – direct OS operation over it !

    ByteBuffer ByteBuffer.allocateDirect(int capacity)

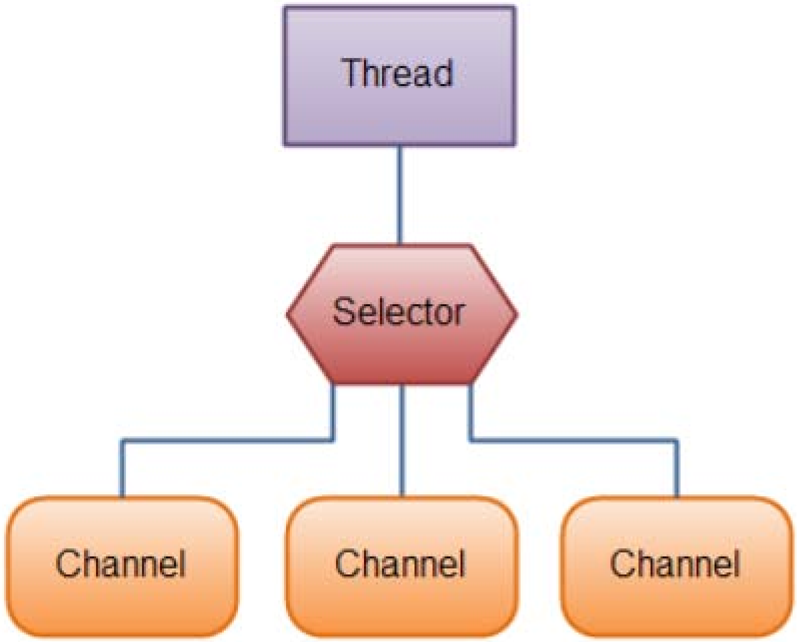
  - typical us



1. Write data into the Buffer
2. Call `buffer.flip()`
3. Read data out of the Buffer
4. Call `buffer.clear()` or `buffer.compact()`

Note: compact() – bytes between position and limit are copied to the beginning of the buffer.

JAVA networking – NIO – Channel, Selector



- » one thread works with multiple channels at the same time
- » Channel – cover UDP+TCP network IO, file IO
  - » FileChannel – from Input/OutputStream or RandomAccessFile
  - » DatagramChannel
  - » SocketChannel
  - » ServerSocketChannel
  - » still missing MulticastDatagramChannel

JAVA networking – NIO – Channel

- Channel
  - read/write at the same time (streams are only one-way)
  - always read/write from/to a buffer
  - channel.transferFrom(int pos, int count, Channel source), transferTo …

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "r");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocateDirect(48);

int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {

  System.out.println("Read " + bytesRead);
  buf.flip();

  while(buf.hasRemaining()){
      System.out.print((char) buf.get());
  }

  buf.clear();
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

JAVA networking – NIO – Selector

- Selector
  - Selector Selector.open();
  - only channels in non-blocking mode can be registered

    ```
    channel.configureBlocking(false);
    SelectionKey channel.register(selector, SelectionKey.OP_READ);
    ```
  - FileChannel doesn’t support non-blocking mode !

- SelectionKey – events you can listen for (can be combined together)
  - OP_CONNECT
  - OP_ACCEPT
  - OP_READ
  - OP_WRITE

- events are filled by channel which is ready with operation

JAVA networking – NIO – Selector

- SelectionKey – returned from register method
  - interest set – your configured ops
  - ready set – which ops are ready, sk.isReadable(), sk.isWritable(), …
  - the channel
  - selector
  - optional attached object – sk.attach(Object obj); Object sk.attachment()

    SelectionKey channel.register(selector, ops, attachmentObj);

- Selector with registered one or more channels
  - int select() – blocks until at least one channel is ready
  - int select(long timeout) – with timeout milliseconds
  - int selectNow() – doesn’t block at all, returns immediately

  - return the number of channels which are ready from the last call !

    Set<SelectionKey> selector.selectedKeys();

JAVA networking – NIO – Selector

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
}
```

JAVA networking – NIO – Selector, SocketChannel

- Selector (cont.)
  - wakeUp() – different thread can “wake up” thread blocked in select()
  - close() – invalidates selector, channels are not closed

- SocketChannel
  - can be configured as non-blocking before connecting
  - SocketChannel socket.getChannel();
  - SocketChannel SocketChannel.open();
  - sch.connect(…)

  - write(…) and read(…) may return without having written/read anything for non-blocking channel !

JAVA networking – NIO

- ServerSocketChannel
  - can be configured as non-blocking
  - can be created directly using open() or from ServerSocket
  - accept() – returns SocketChannel in the same mode

- DatagramChannel
  - can be configured as non-blocking
  - can be created directly using open() or from DatagramSocket
  - receive(…), send(…)

- FileChannel
  - cannot be non-blocking !
  - support – direct buffers, mapped files, locking