Java NIO (New I/O) and Non-Blocking Sockets

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

- New I/O APIs introduced in JDK 1.4
- Provides a new I/O model based on channels, buffers and selectors
- Enables non-blocking 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 Programming Abstractions

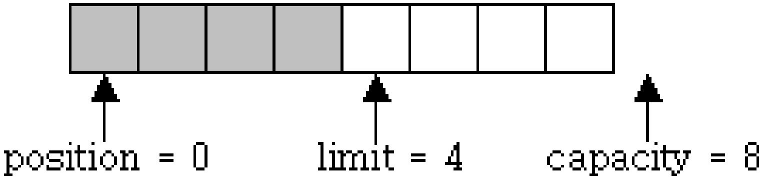
- *Buffers*
  - Containers for data
  - Can be filled, drained, flipped, rewind, etc.
  - Can be written/read to/from a channel
- *Channels* of various types
  - Represent connections to entities capable of performing I/O operations, e.g. pipes, files and sockets
  - Can be selected when ready to perform 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

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`	**Pattern matching using regular expressions.**

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



Some Buffer methods

- `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.**
- `allocate()` **Allocate a new buffer.**
- `clear()` **Clear the buffer, i.e. prepare the buffer for writing data by channel-reads or relative puts (limit = capacity; position = 0)**
- `flip()` **Prepare the buffer for reading data by channel-writes or relative gets (limit = position; position = 0)**
- `rewind()` **Prepare the buffer for re-reading data (position = 0)**
- `mark()` **Set this buffer's mark equal to its position (mark = position)**
- `reset()` **Reset this buffer's position equal to its mark (position = mark)**
- `wrap()` **Wrap a given array in a buffer**
- `get(), put()` Absolute (index-based) and relative (position-based) get/put data from/into the buffer
- `hasRemaining()` **Check whether there are any elements between the current position and the limit**

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();
}
```

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 blocking/non-blocking, enable *non-blocking I/O operations*

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.
- 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.

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.

`map()`	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

```
public class FileChannelTest {
  public static void main(String[] args) {
    String filename = "test.txt";
    try {
      FileInputStream inf = new FileInputStream(filename);
      try (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);
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

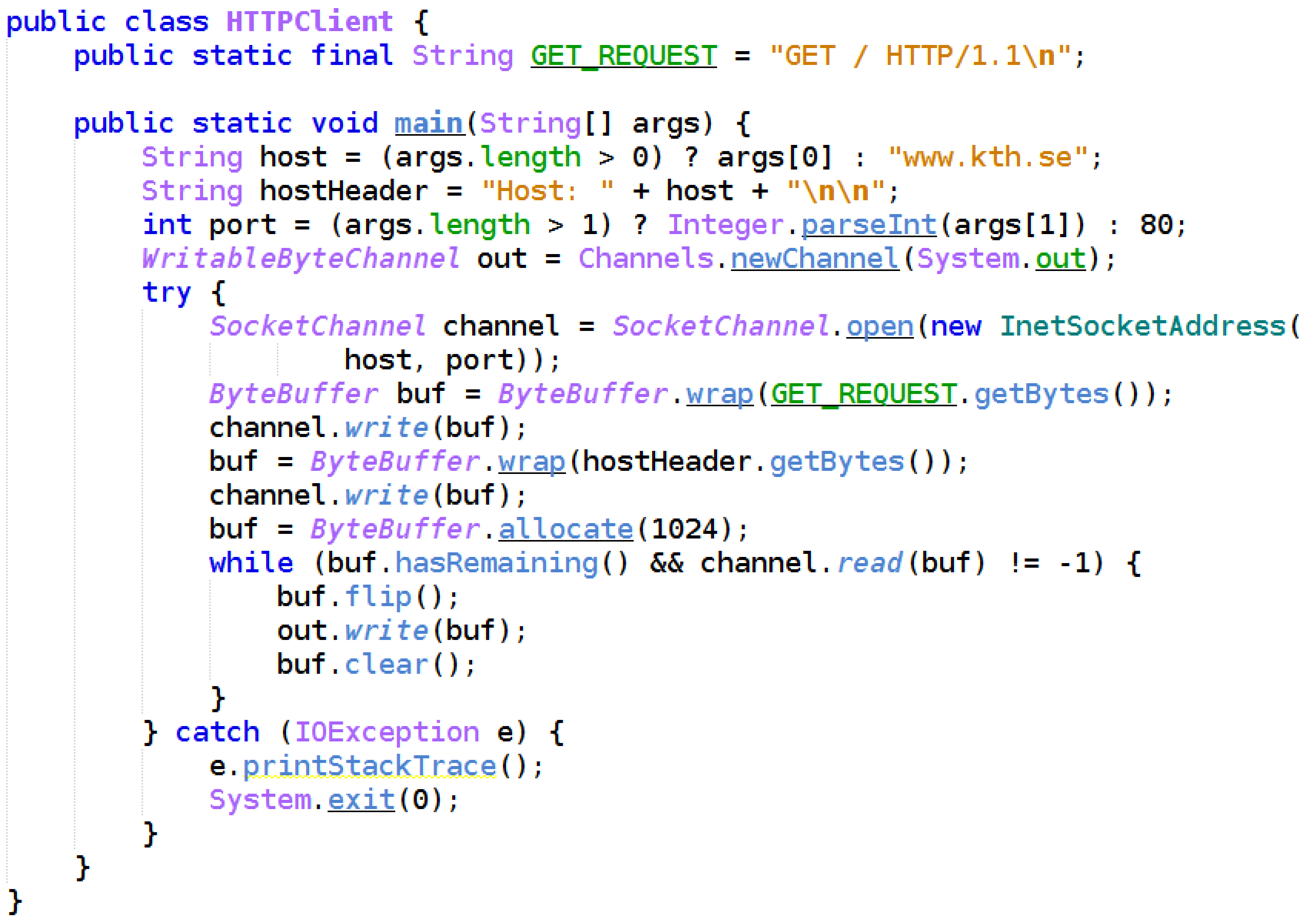
SocketChannel

- A selectable channel for TCP sockets.
  - Reads from and writes to a TCP socket.
- Each `SocketChannel` is associated with a `Socket` object

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

Non blocking

Socket Channel Example



```
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 *TCP listening sockets.*
- Each `ServerSocketChannel` is associated with a `ServerSocket` object

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
ServerSocket socket = serverChannel.socket();
socket.bind(new InetSocketAddress(port));
```

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); non-Blocking
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

  - Register as many channels as you have/need

Selector methods

- `select()` blocking select, returns a set of keys whose channels are ready for I/O.
- `selectNow()` non-blocking select, returns zero if no channels are ready
- `selectedKeys()` returns 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);
```

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

```
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 Echo 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_WRITE);
```

Non-Blocking Echo 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);
      }
    }
  }
}
```