

IO in Java

IO Overview

- Input and output of bytes from files/sockets/etc
- The Java language has evolved over the years to accommodate programmers' needs. There are:
 - Initial IO classes/objects (in package `java.io`)
 - Java 1.0
 - Blocking read/write
 - NIO (new IO) classes/objects (in package `java.nio`)
 - Java 1.4
 - Non-blocking read/write
 - NIO2 (new new IO) classes/object (also in package `java.nio`)
 - Java 1.7
 - Better File/Path abstractions

Input/Output Streams

- `InputStream (java.io.InputStream)`
 - Blocking read bytes from some source (file, network, etc)
- `OutputStream (java.io.OutputStream)`
 - Blocking write bytes to some destination (file, network, etc)
- `InputStream/OutputStream` are fairly low-level. Reading/writing a single byte or array of bytes at a time.
 - Do not take into account file encodings/character sets
 - Cannot be (easily) interrupted
- Java provides abstractions on top of `InputStream/OutputStream` (see Core Java Volume II Figures 1.1, 1.2 and 1.3)
 - Most common and the ones we'll be using are
 - `FileInputStream / FileOutputStream`
 - `BufferedInputStream / BufferedOutputStream`
 - `PrintStream`
 - `Reader / Writer`
 - Buffered & File variants
 - `Closeable / Autocloseable / Flushable / Appendable` interfaces

InputStream Usage

```
1 public class InputStreams {
2
3     public void read(InputStream stream) {
4
5         // can read one byte
6         try {
7             int read = stream.read();
8             if (read == -1) {
9                 System.out.printf("Stream is closed\n");
10            } else {
11                // read is a single byte
12                byte valueRead = (byte) read;
13                System.out.printf("Read byte value %s\n", Byte.toString(valueRead));
14            }
15        } catch (IOException ioe) {
16            System.out.printf("Failed to read - %s\n", ioe.getMessage());
17        }
18
19        // check for availability and read into an array
20        try {
21            int availableAmount = stream.available();
22            byte[] into = new byte[availableAmount];
23            int read = stream.read(into, 0, into.length);
24            if (read == -1) {
25                System.out.printf("Stream is closed\n");
26            } else {
27                System.out.printf("Read %d bytes into buffer\n", read);
28            }
29        } catch (IOException ioe) {
30            System.out.printf("Failed to read - %s\n", ioe.getMessage());
31        }
32    }
33 }
```

OutputStream Usage

```
1 public class OutputStreams {
2
3     public void write(OutputStream stream) {
4
5         // can write one byte
6         try {
7             stream.write(0x01);
8         } catch (IOException ioe) {
9             System.out.printf("Failed to write - %s\n", ioe.getMessage());
10        }
11
12        // write bytes from an array and then flush
13        try {
14            byte[] from = new byte[] { 0x1, 0x2, 0x3, 0x4 };
15            stream.write(from, 0, from.length);
16            stream.flush();
17        } catch (IOException ioe) {
18            System.out.printf("Failed to read - %s\n", ioe.getMessage());
19        }
20
21    }
22}
```

FileInputStream

- Extends InputStream to assist in reading byte values from a File

```
1  public class FileInputStreams {
2
3      public void read(String filePath) {
4          read(new File(filePath));
5      }
6
7      public void read(File file) {
8
9          try (FileInputStream stream = new FileInputStream(file)) {
10             int availableAmount = stream.available();
11             byte[] into = new byte[availableAmount];
12             int read = stream.read(into, 0, into.length);
13             if (read == -1) {
14                 System.out.printf("Stream is closed%n");
15             } else {
16                 System.out.printf("Read %d bytes into buffer%n", read);
17             }
18         } catch (IOException ioe) {
19             System.out.printf("Failed to read - %s%n", ioe.getMessage());
20         }
21     }
22 }
```

BufferedInputStream

- Extends InputStream to provide buffered reading (or writing in analogous BufferedOutputStream).

```
1  public class BufferedInputStreams {
2
3      public void read(InputStream input) {
4
5          try (BufferedInputStream stream = new BufferedInputStream(input)) {
6              int availableAmount = stream.available();
7              byte[] into = new byte[availableAmount];
8              int read = stream.read(into, 0, into.length);
9              if (read == -1) {
10                 System.out.printf("Stream is closed%n");
11             } else {
12                 System.out.printf("Read %d bytes into buffer%n", read);
13             }
14         } catch (IOException ioe) {
15             System.out.printf("Failed to read - %s%n", ioe.getMessage());
16         }
17     }
18 }
19
```

Reader

- Let's read a file!
 - File norwegian-names.txt
 - Should print the persons names

```
1  public static void main(String[] args) throws IOException {  
2  
3      try (FileInputStream stream = new FileInputStream(args[0])) {  
4          int read;  
5          while ((read = stream.read()) != -1) {  
6              System.out.printf("%c", (char) read);  
7          }  
8      }  
9  
10 }
```


Reader (cont)

- WAT?!?!?
- InputStream / OutputStream (and their subclasses) do not take character encodings into account.
- What are character encodings?
 - ways of representing characters as bytes
 - US-ASCII
 - ISO-8859-1
 - UTF-8
 - UTF-16
 - UTF-16BE
 - UTF-16BL

Reader (cont) - Charset

- Java handles character encoding using the Charset abstraction.
 - Each platform / machine has a default Charset (or character encoding).
 - Many machines use UTF-8 as the default character encoding.

```
1 public class CharSets {  
2  
3     public static void main(String[] args) {  
4         String charsetName = args[0];  
5  
6         Charset defaultCharset = Charset.defaultCharset();  
7  
8         Charset charset = Charset.forName(charsetName);  
9  
10        System.out.printf("%s %s %s", defaultCharset, (defaultCharset.equals(charset) ? "==" : "!="), charset);  
11    }  
12 }  
13  
14 }
```

Reader (cont)

- Reader!
 - Has subclasses like **InputStreamReader**, **FileReader**, **BufferedReader** / etc

```
1  public static void main(String[] args) throws IOException {  
2  
3      try (FileReader reader = new FileReader(args[0])) {  
4          int read;  
5          while ((read = reader.read()) != -1) {  
6              System.out.printf("%c", (char) read);  
7          }  
8      }  
9  
10 }
```

Closeable / Autocloseable

- Must close all of the classes we've mentioned.
 - Remember to always wrap with a try/finally
 - Or (if Java 1.7) and the class implements Autocloseable use the try-with-resources

```
1 public void openAndClose(String file) {
2
3     FileInputStream stream = null;
4     try {
5         stream = new FileInputStream(file);
6         // do something
7     } catch (IOException ioe) {
8         // handle exception
9     } finally {
10        if (stream != null) {
11            try {
12                stream.close();
13            } catch (IOException ioe) {
14                // print or do nothing
15            }
16        }
17    }
18
19 }
```

NIO (non-blocking, memory-mapped)

- IO calls block the invoking thread until the IO operation completes (or fails). Additionally, they don't (always) leverage platform specific efficient operations (like memory mapped files, direct file copy, etc).
- NIO (in Java) provided mechanisms to perform non-blocking IO operations, memory mapped files and more access to platform specific operations (like file copying).
- NIO is built on different paradigms (than IO which is stream based).
 - Buffers
 - Channels
 - Selectors

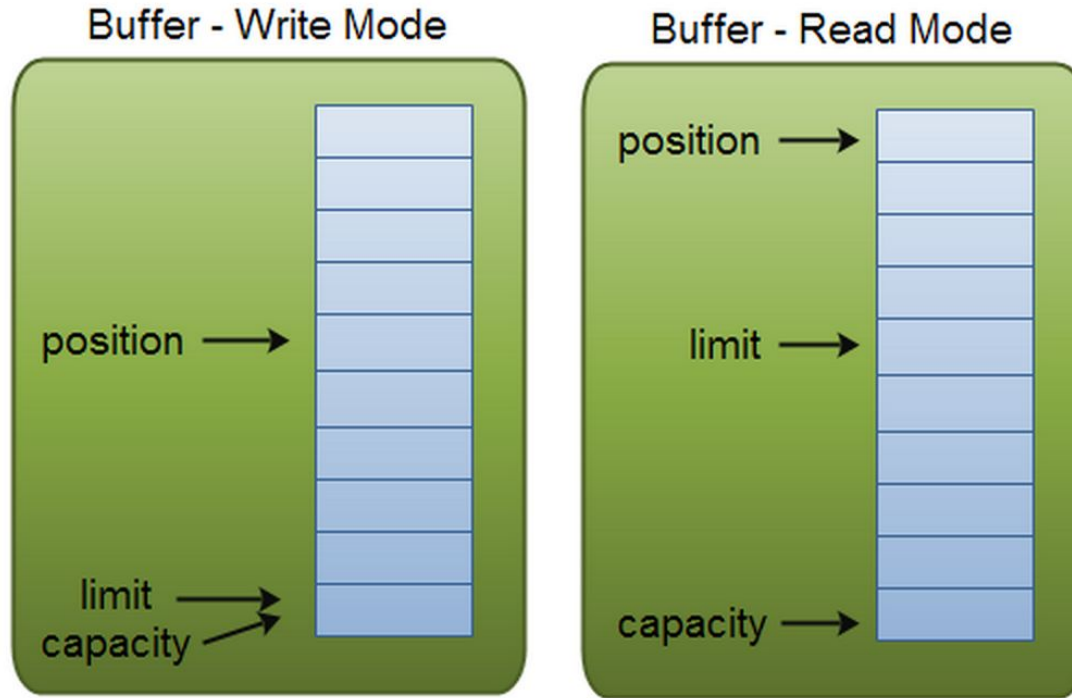
NIO - Buffer

- “These classes represent a contiguous extent of memory, together with a small number of data transfer operations. Although theoretically these are general-purpose data structures, the implementation may select memory for alignment or paging characteristics, which are not otherwise accessible in Java. Typically, this would be used to allow the buffer contents to occupy the same physical memory used by the underlying operating system for its native I/O operations, thus allowing the most direct transfer mechanism, and eliminating the need for any additional copying. In most operating systems, provided the particular area of memory has the right properties, transfer can take place without using the CPU at all. The NIO buffer is intentionally limited in features in order to support these goals.” http://en.wikipedia.org/wiki/Non-blocking_I/O_%28Java%29

NIO - Buffer (cont)

```
1  public void read(ByteBuffer buffer) {
2
3      // moves the 'position' pointer to the start of the underlying buffer
4      buffer.flip();
5
6      while (buffer.hasRemaining()) {
7          System.out.printf("%d", buffer.get());
8      }
9
10     // moves the 'position' pointer to 0 and the 'limit' pointer to the 'capacity' pointer
11     buffer.clear();
12
13 }
```

NIO - Buffer (cont)



Buffer capacity, position and limit in write and read mode.

NIO - Channel

- Provide bulk data transfer to/from NIO Buffer objects.
 - Obtained from the higher-level objects (FileInputStream, sockets, etc)

```
1  public void fastTransfer(File from, File to) throws IOException {  
2  
3      try (FileInputStream fromStream = new FileInputStream(from);  
4           FileOutputStream toStream = new FileOutputStream(to)) {  
5  
6          FileChannel fromChannel = fromStream.getChannel();  
7          FileChannel toChannel = toStream.getChannel();  
8  
9          // JVM will attempt to do this with native I/O methods  
10         fromChannel.transferTo(0, fromChannel.size(), toChannel);  
11  
12     }  
13
```

NIO - Channel (cont)

```
1 public void write(File to, byte[] values) throws IOException {  
2  
3     try (FileChannel channel = new FileInputStream(to).getChannel()) {  
4         ByteBuffer buffer = ByteBuffer.wrap(values);  
5         channel.write(buffer);  
6     }  
7  
8 }
```

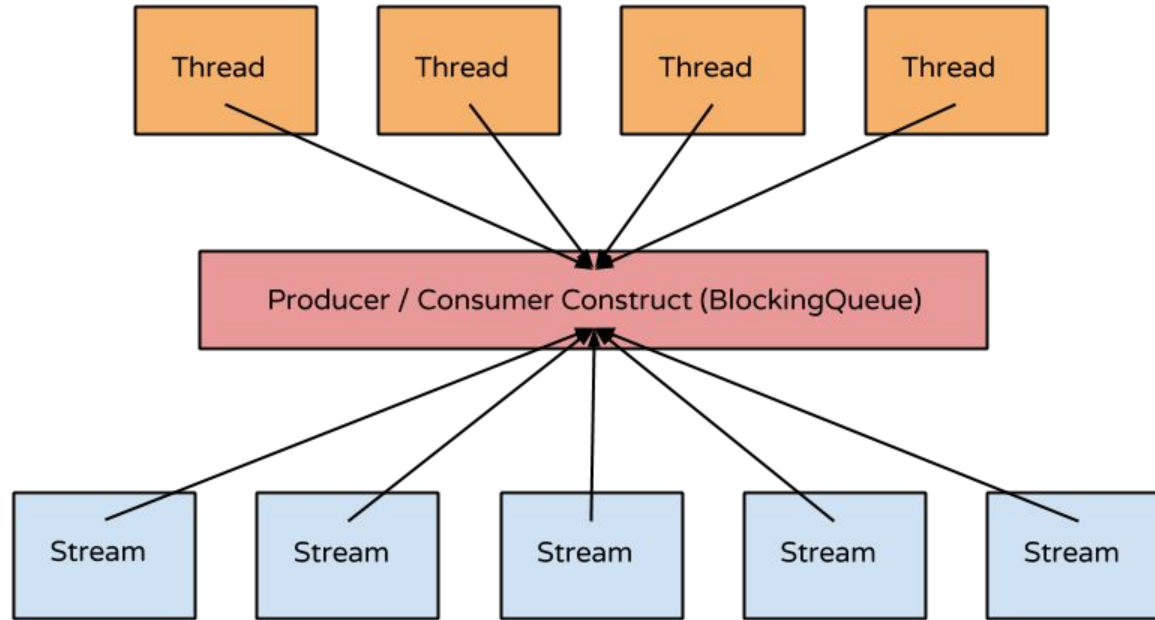
NIO - Memory Mapped Files

```
1 public ByteBuffer mapForReading(File to) throws IOException {  
2     FileChannel channel = new FileInputStream(to).getChannel();  
3     return channel.map(FileChannel.MapMode.READ_ONLY, 0L, channel.size());  
4 }  
5  
6 public ByteBuffer map(File to) throws IOException {  
7     FileChannel channel = new FileInputStream(to).getChannel();  
8     return channel.map(FileChannel.MapMode.READ_WRITE, 0L, channel.size());  
9 }
```

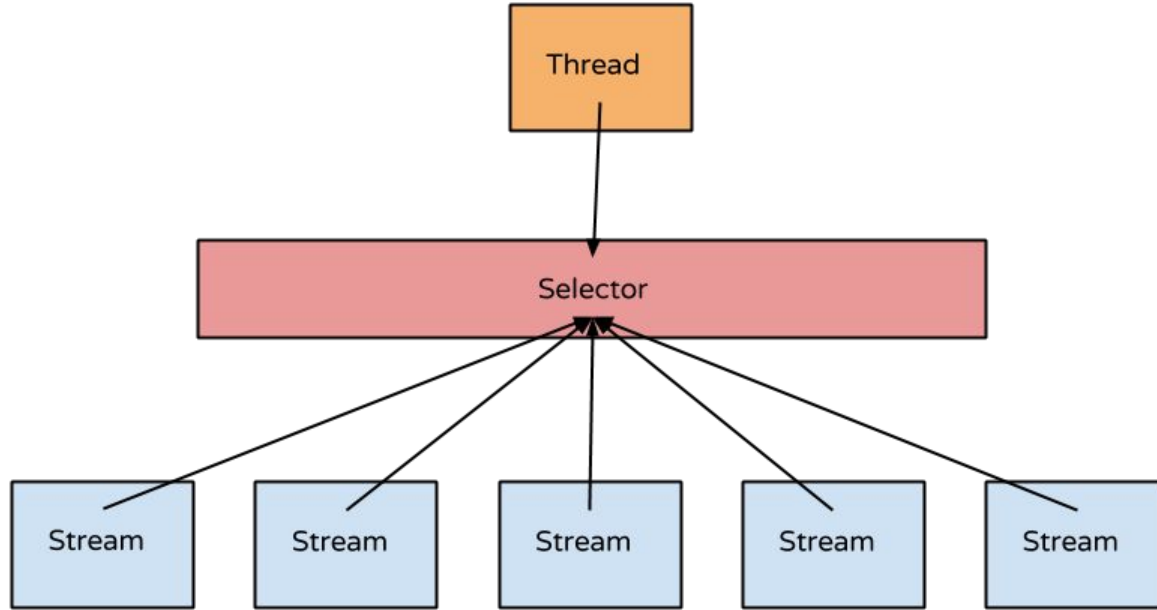
NIO - Selector

- Provides a way of waiting on Channel objects and recognizing when one or more become available for data transfer.
- Register a Selector with a Channel
- Enables non-blocking IO.
 - Multiplexing behavior which doesn't require threads. The Selector can provide a more efficient implementation using lower-level operating system constructs.

Handling Network Sockets (blocking)



Handling Network Sockets (non-blocking)



Non-blocking Example


```
1 public void handle(SelectableChannel... channels) throws IOException {
2
3     Selector selector = Selector.open();
4
5     for (SelectableChannel channel : channels) {
6         channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
7     }
8
9     while (!Thread.currentThread().isInterrupted()) {
10
11         int readyChannels = selector.select();
12         if (readyChannels == 0) {
13             continue;
14         }
15
16         Set<SelectionKey> selectedKeys = selector.selectedKeys();
17         Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
18
19         while (keyIterator.hasNext()) {
20             SelectionKey key = keyIterator.next();
21             SelectableChannel channel = key.channel();
22             if (key.isReadable()) {
23                 // TODO - read from channel
24             } else if (key.isWritable()) {
25                 // TODO - write to channel
26             }
27             keyIterator.remove();
28         }
29     }
30
31 }
```

New NIO - Java 7 File Abstractions

- Provides convenience methods for interacting with file systems and files.
 - `FileSystem` - interaction with the underlying file system
 - `Path` - a location to directories and files within a `FileSystem`
 - `Files` - convenience class (like `Collections`) to assist in doing common tasks related to `File` objects (copy, move, etc)
- Provided platform-agnostic notions of determining if files were hidden, symbolic, etc.
 - Prior to Java 7, this wasn't possible in Java in a platform agnostic manner


```
1 public void explore() throws IOException {
2
3     FileSystem defaultFileSystem = java.nio.file.FileSystems.getDefault();
4     // prior to Java 7, would have to know system separator ('/' v '\\') when concatenating file paths
5     Path path = defaultFileSystem.getPath("Users", "blangel", "projects");
6     Path samePath = Paths.get("Users", "blangel", "project");
7
8     System.out.printf("Is %s the same file as %s ? %s", path.getFileName(), samePath.getFileName(),
9         Files.isSameFile(path, samePath));
10    System.out.printf("Is %s a directory ? %s", path.getFileName(), Files.isDirectory(path));
11    System.out.printf("Is %s a symbolic-link ? %s", path.getFileName(), Files.isSymbolicLink(path));
12
13    ByteArrayOutputStream output = new ByteArrayOutputStream();
14    // copy to an output stream
15    Files.copy(path, output);
16
17    // create a file
18    Path file = Files.createFile(Paths.get("Users", "blangel", "foo.txt"));
19    Path tempDirectory = Paths.get("tmp");
20
21    // move a file to another location
22    Files.move(file, tempDirectory);
23
24    // read all bytes
25    byte[] bytes = Files.readAllBytes(file);
26
27    // read all lines
28    List<String> lines = Files.readAllLines(file, Charset.forName("UTF-8"));
29
30 }
```

Read Chapter 6.3 & Java 8 Tutorials

- Core Java - Chapter 6 section 3
 - [Default / Static Interface Methods](#)
 - [Lambda Expressions](#)
 - [Method References](#)
- 

Homework 11

<https://github.com/NYU-CS9053/Spring-2017/homework/week11>