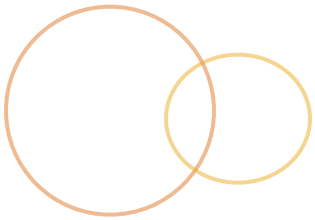
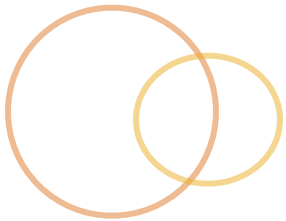
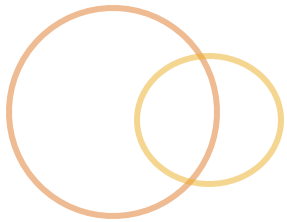
Java NIO





This is not your Father's Matrix…

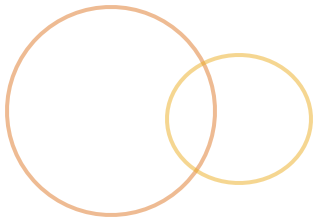
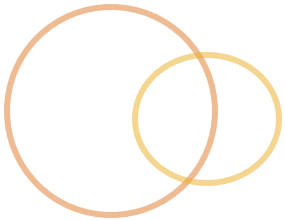
Presentation Topics



In this presentation, we will cover:

- Introduction to Java I/O
- Introduction to Java NIO
- Advanced NIO

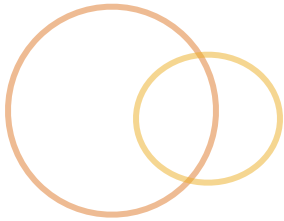
Objectives

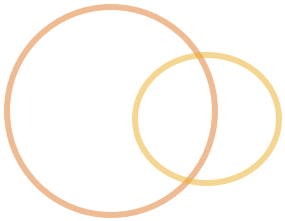

When we are done, you should be able to:

- Describe the differences between Java I/O and Java NIO
- Identify the key components of NIO
- Write a basic file reader using NIO
- Write a basic network handler using NIO

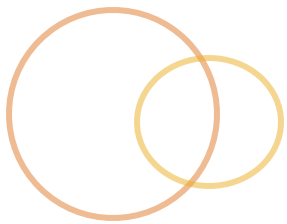
Introduction to I/O



A Quick Review



Intro to I/O

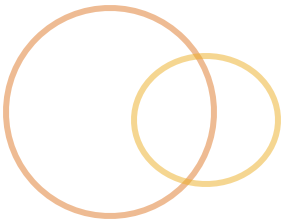


- Stands for input / output
- Input / output interface between application and operating system
- Typically seen as stream of data
- Streams filled and emptied with bytes

Java I/O

- Been around since JDK 1.0
- Found in `java.io`
- Two fundamental stream types
  - Binary
    - `InputStream`
    - `OutputStream`
    - Single byte focus
  - Character
    - `Reader`
    - `Writer`
    - Single character focus

Java I/O [cont.]



- Hides I/O details
  - Implemented in layered approach
  - Abstracts OS
- Supports stream chaining
  - Form of Decorator pattern
  - Convert stream into “higher-level” I/O construct
- Integrated with networking capabilities

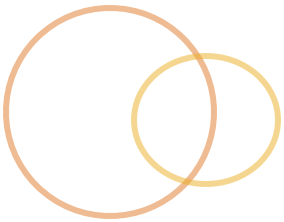
Java I/O Example

```
 1  package examples.io;
 2
 3  import java.io.*;
 4
 5  public class CopyFileIO {
 6
 7    public static void main(String[] args) {
 8      File original = new File("/tmp/pic.jpg");
 9      File copy = new File("/tmp/pic_copy.jpg");
10      int fileLength = (int) original.length();
11      InputStream originalStream = null;
12      OutputStream copyStream = null;
13
14      try {
15        originalStream = new FileInputStream(original);
16        copyStream = new FileOutputStream(copy);
17        byte [] contents = new byte[fileLength];
18        originalStream.read(contents);
19        copyStream.write(contents);
20      } catch(IOException ioe) {
21        ioe.printStackTrace();
22      } finally {
23        try{
24          originalStream.close();
25        } catch(IOException ioe) {}
26        try{
27          copyStream.close();
28        } catch(IOException ioe) {}
29      }
30    }
31  }
```

Introduction to Java NIO

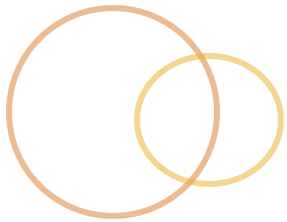
Channel the Stream into a Buffer

Overview of NIO



- Introduced with JDK 1.4
- Shift to block-oriented I/O
- Supports non-blocking I/O facilities
- `java.io` re-implemented using NIO

Motivations for NIO

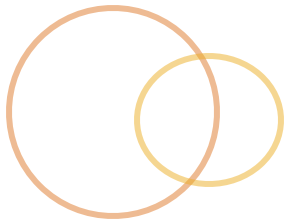


- Provide high-speed, high-throughput I/O
  - Avoid having to use native code
  - Available through Java platform
- Support asynchronous interactions

I/O and NIO Differences

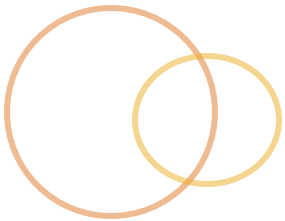
- I/O
  - Stream based; one `byte` at a time
  - Blocking
  - Easy to “build” up with chains
  - Slow
- NIO
  - Block based; produces / consumes block of data in one operation
  - Non-blocking
  - Not as elegant
  - Fast

NIO Components



- Central Themes of NIO
  - Buffers
  - Channels
  - Selectors and selection keys
  - Charsets
- Found in 3 primary packages
  - `java.nio`
  - `java.nio.channels`
  - `java.nio.charsets`

NIO Buffers



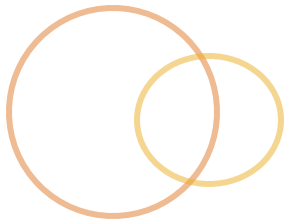
- Data container
  - Conceptually an array of byte
  - Provides structured access to data
  - Represents block
- Used by Channels for read/write operations
  - Reads fill a `Buffer`
  - Writes drain a `Buffer`
- Tracks read/write interactions

java.nio.Buffer

- Abstract class
  - Parent for all other `Buffers`
  - Supports only primitive data elements
- Basic buffer characteristics:
  - **Position** - index representing where should read / write
  - **Limit** - value representing first element that should not be read / written
  - **Capacity** - value representing number of elements buffer contains
  - **Mark** – positional memory

  0 <= *mark* <= *position* <= *limit* <= *capacity*

Buffer Management



- Marking - `mark() : Buffer`
  - Sets a mark in the buffers at current position
- Resetting - `reset() : Buffer`
  - Returns position to mark
- Clearing - `clear() : Buffer`
  - Clears buffer
  - Position set to 0; limit set to capacity; marks removed
  - Makes buffer ready for reads; Call before filling buffer
  - Does not “clear” the data

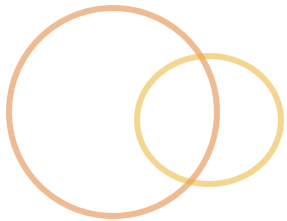
Buffer Management [cont.]

- Flipping - `flip() : Buffer`
  - Flips buffer
  - Limit set to current position; position set to 0; marks removed
  - Makes buffer ready for writes; Call before emptying buffer
- Rewinding - `rewind() : Buffer`
  - Rewinds buffer
  - Sets position to 0; removes marks
  - Makes buffer ready for re-reading info

Buffer Implementations

- Buffer implementation for every primitive . . .
  - `ByteBuffer, ShortBuffer, IntBuffer, LongBuffer`
  - `FloatBuffer, DoubleBuffer`
  - `CharBuffer`
  - . . . except `boolean`
- Creating Buffers
  - Allocate memory
  - Wrap existing collection

NIO Channels



- Connection to something that can do I/O
  - Files, Sockets, etc
  - Similar to streams in `java.io`
  - All NIO goes through channels
- Don't directly read / write to `Channel`
  - All data moved through `Channel` using `Buffer`
  - Bi-directional I/O
- Cannot create an instance; derive from I/O entity

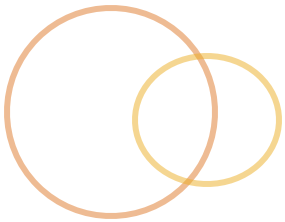
java.nio.Channel

- Super-interface for all NIO channels
  - Almost tag-like interface
  - Provides two basic operations: `close` and `isOpen`
- Channel realization found in **large** type hierarchy
  - `FileChannel`
  - `ByteChannel, InterruptibleChannel`
  - `ScatteringChannel, GatheringChannel`
  - etc.
- Reading / Writing not available on all channels
  - `ReadableByteChannel`
  - `WriteableByteChannel`

FileChannel

- `java.nio.channels.FileChannel`
  - Channel used to access (r/w) to files
  - Similar to `RandomAccessFile` in terms of functionality
- Must be derived from:
  - `FileInputStream`
    - `FileChannel fc = fis.getChannel()`
    - Only represents a readable channel
  - `FileOutputStream`
    - `FileChannel fc = fos.getChannel()`
    - Only represents a writable channel
  - Or, utility methods in Channels class

FileChannel [cont.]



Full featured file support

- File locking
  - `lock`
  - `tryLock`
  - `FileLock` class
- Optimized data transfer
  - `transferFrom`
  - `transferTo`
- Truncating - `truncate`

Basic NIO File I/O Example

- Copy contents of one file into another
- Steps involved
  - Create data holder for `Channel`
  - Create `Channel` for old `File` and new `File`
  - Read data from old `File` into `Buffer`
  - Writing data from `Buffer` into new `File`

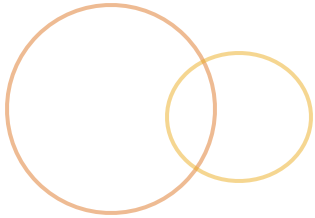
NIO File Copy Example

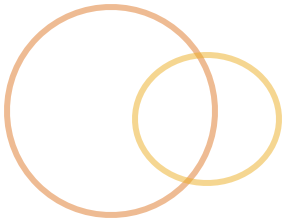
```
1  package examples.nio;
2
3  import ...
9
10 public class CopyFileNIO {
11
12   public static void main(String[] args) throws IOException {
13     File readFile = new File("/tmp/pic.jpg");
14     File writeFile = new File("/tmp/pic_copy.jpg");
15
16     ByteBuffer fileBuffer = ByteBuffer.allocate(256);
17
18     FileInputStream fis = new FileInputStream(readFile);
19     FileChannel origChannel = fis.getChannel();
20     FileOutputStream fos = new FileOutputStream(writeFile);
21     FileChannel copyChannel = fos.getChannel();
22
23     int bytesRead = origChannel.read(fileBuffer);
24     while(bytesRead != -1) {
25       fileBuffer.flip();
26       copyChannel.write(fileBuffer);
27       fileBuffer.clear();
28       bytesRead = origChannel.read(fileBuffer);
29     }
30
31     copyChannel.close();
32     origChannel.close();
33     fos.close();
34     fis.close();
35   }
36 }
```

NIO LAB: Write a File Copier

- Implement a file copier using a `FileChannel`, and a `ByteBuffer`. Refer to the example for guidance.

ByteBuffer





- Support features you expect of buffer
  - Allocating with specific size
  - Wrapping array as buffer
  - Slicing a buffer into two
  - Making it read-only

ByteBuffer Configurations

- Supports three types of configurations
  - Indirect - Fast
    - Read / writes stored in intermediary buffer before I/O operations
    - Default configuration
  - Direct - Faster
    - Performs native I/O operations directly on buffer
    - Created using `allocatedDirect`
    - May exist outside of GC heap

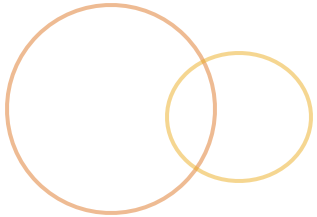
ByteBuffer Configurations [cont.]

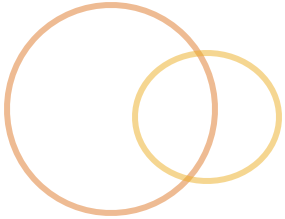
- Supports three types of configurations [cont.]
  - Memory Mapped - Fastest*
    - Map portion of file to physical memory
    - Created using `FileChannel.map`
      - Returns specific type of `ByteBuffer`
      - `MappedByteBuffer`
    - Mapping based on `FileChannel.MapMode`
      - `READ_ONLY`
      - `READ_WRITE`
      - `PRIVATE`
    - Changes dependent on OS
    - *Fastest when dealing with large files

LAB : Write a Mapped Byte Buffer

- Write a file copier that uses a direct byte buffer to perform copying.
- Do a time based test against the direct and indirect file copy mechanism.
- Write one more file copy using standard I/O.
- Re-run the time test again.
- Which one wins? By how much?

Advanced NIO

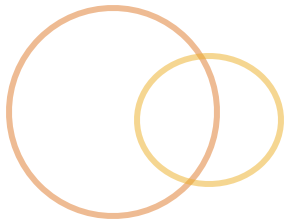




Asynchronous I/O

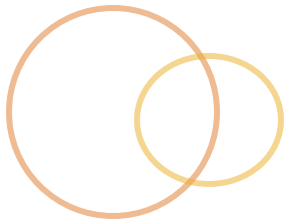
No more lines . . .

Asynchronous I/O



- Reading / writing data without blocking
  - No waiting for data on read or for access on write
  - Can use single thread to do ***ALL*** I/O operations
- Operations occur as result of notification
  - Event-like, notification-based system
  - Register “interest” in specific events
  - Associate “registrations” with `Channels`
- Relies on:
  - `SelectableChannels` - configured to support NIO
  - `Selector` - event medium
  - `SelectionKey` - registration identification
  - Handler - entity that processes “events”

SelectableChannel



- Channel that supports selection
- Safe for multi-threaded interactions
- Supports two modes:
  - Blocking
    - Every I/O operations blocks until completes
    - Default configuration
  - Non-blocking
    - No I/O blocks
    - Configured through `configureBlocking` method
    - Must be configured to non-blocking before registration with `Selector`

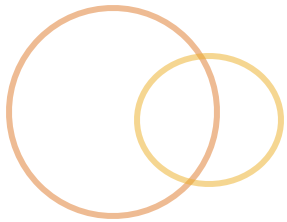
SelectableChannel [cont.]

- Selection functionality provided through registration
  - Registration relies on:
    - Selector - event facilitator
    - I/O op codes - event “ids”
  - Performed through:
    - `public SelectionKey register(Selector sel, int ops)`
    - `public SelectionKey register(Selector sel, int ops, Object att)`

SelectableChannel [cont.]

- Selection functionality provided through registration
  - Registration represented as `SelectionKey`
    - Valid until key is canceled
    - Or when channel closes
    - Or when `Selector` closes
- Common implementations:
  - `ServerSocketChannel`
  - `SocketChannel`

ServerSocketChannel



- Channel representation of a `ServerSocket`
  - All `ServerSocketChannel` objects have a associated `ServerSocket`
  - Not all `ServerSocket` have an associated `ServerSocketChannel`
- Creating a `ServerSocketChannel`

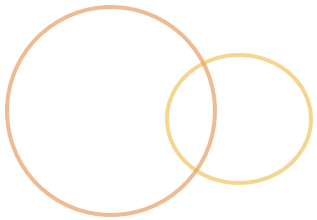
```
Selector selector = Selector.open();
ServerSocketChannel channel = ServerSocketChannel.open();
ServerSocket socket = channel.socket();
socket.bind(xxxx);
//configure non-blocking
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_ACCEPT);
```

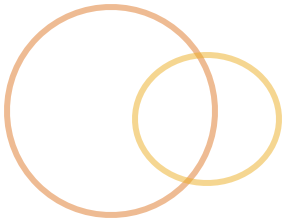
SocketChannel

- Channel representation of a `Socket`
  - All `SocketChannel` objects have a associated `Socket`
  - Not all `Socket` have an associated `SocketChannel`
- Creating a `SocketChannel`

```
Selector selector = Selector.open();
SocketChannel channel = SocketChannel.open();
Channel.connect(xxxx);
Socket socket = channel.socket();
//configure non-blocking
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_ACCEPT);
```

Selector



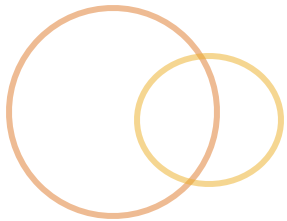


- Functions as an event medium facilitator
  - Not true event listener / event handler
  - More of a poll-based versus notification-based mechanism
  - Binds channel, I/O events, and handler
- Basic operation of `Selector`
  - Receives registration interest in I/O related events
  - `Channel` notifies `Selector` when “event” occurs
  - Poll `Selector` for “events”

Selector [cont.]

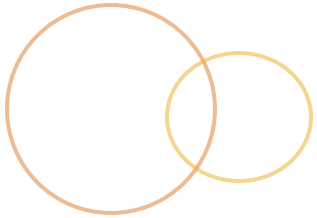
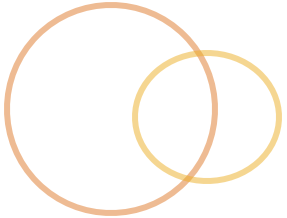
- `java.nio.channels.Selector`
  - Created using `open` call; functions as Factory method
  - Associated with `SelectableChannel`
  - Contains `SelectionKey` sets
  - Considered multi-threaded safe
- Two main functions:
  - Registration - represented by `SelectionKey`
  - Selection - getting subset of `SelectionKey` representing channels ready for I/O operations

SelectionKey



- Represents registration of `Selector` with `Channel`
  - New key is created for each registration
  - Key stays valid until canceled
- Used as “notification” mechanism when event occurs
  - “Event id” represented as op code
  - 4 predefined op-codes:
    - `OP_ACCEPT`
    - `OP_CONNECT`
    - `OP_READ`
    - `OP_WRITE`
- Can contain attachment accessible by “handler”

Handler


- No “handler” interface
- Three main functions:
  - Select ready channels
    - `select() : int` - synchronous
    - `select(long timeout) : int` - synchronous
    - `selectNow() : int` - asynchronous
  - Retrieves selected channels
    - `selectedKeys() : Set<SelectionKey>` - result not thread safe
  - Processes results
    - `channel() : SelectableChannel` - “ready” chanel
    - read / write
    - remove key from selection
- Typically written as loop

Asynchronous I/O Example

- Traditionally, needed one thread per connection to handle “many requests”
  - Can adopt NIO strategy
    - One thread for “accepts”
    - One thread for request processing
  - Three key components
    - Main - application launcher
    - ServerSocketThread - accepts connections
    - SocketThread - processes requests

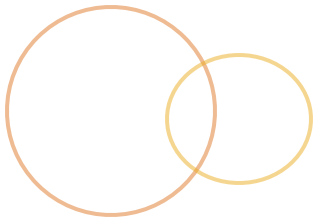
NIO Charsets

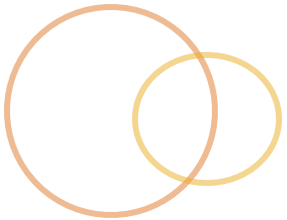
- “named mapping between sequences of 16-bit Unicode characters and sequences of 8-bit characters”
- NIO supports character set mappings on channels
  - Not typically used by developer
  - Defined in terms of:
    - `CharsetEncoder` - encodes sequence of characters
    - `CharsetDecoder` - decodes sequence of characters
- Found in `java.nio.charset`

Summary

- Java NIO expands I/O to a channel model
- `Channels` are like streams of data that support bi-directional I/O operations
- `Channels` rely on `Buffers` to read / write data
- `SelectableChannels` support asynchronous I/O

NIO Lab 2





- Description: Convert the sitemap utility that was created in the advanced threading chapter into a asynchronous NIO application. The resulting application will function as an asynchronous network client, similar to how a browser works when you surf the web.
- Duration: 1 hour