# Java NIO

# Java NIO

Java started initially by offering the File class, in the java.io package to access file systems. This object represents a file/directory and did allow you to perform some operations such as checking if a file/directory exists, get properties and delete it.

It had, though, some shortcomings. To name a few:

- The File class lacked some important functionality, such as a copy method.
- It also defined many methods that returned boolean. As one can imagine, in case of an error, false was returned, rather than throwing an exception. The developer had, indeed, no way of knowing why it failed.
- Did not provide good handling on support of symbolic links.
- A limited set of file attributes was provided.

**To overcome these problems, java.nio package was introduced in java 4. The key features were:**

- Channels and Selectors: A channel is an abstraction on lower-level file system features, e.g. memory-mapped files.
- Buffers: Buffering for all primitive classes
- Charset: Charset (java.nio.charset), encoders, and decoders to map bytes and Unicode symbols

# Creating a NIO Path

## Creating NIO Path

A Path object is a hierarchical representation of the path on a system to the file or directory.
The java.nio.file.Path interface is the primary entry point for working with the NIO 2 API.
The easiest way to create a Path Object is to use the java.nio.files.Paths factory class

```
Path p2 = Paths.get("/home/project");
```

## Writing files with the NIO API

```
Path path = Paths.get("src/main/resources/question.txt");
String question = "To be or not to be?";
Files.write(path, question.getBytes());
```

The **Files.newBufferedWriter(Path,Charset)** writes to the file at the specified Path location, using the user defined Charset for character encoding.

```
Path path = Paths.get("src/main/resources/shakespeare.txt");
try(BufferedWriter writer = Files.newBufferedWriter(path, Charset.forName("UTF-8"))){
writer.write("To be, or not to be. That is the question.");
}catch(IOException ex){
ex.printStackTrace();
}
```

# NIO API to copy a file

Using NIO API to copy a file with an OutputStream

```
Path oldFile = Paths.get("src/main/resources/", "oldFile.txt");
Path newFile = Paths.get("src/main/resources/", "newFile.txt");
  try (OutputStream os = new FileOutputStream(newFile.toFile())) {
    Files.copy(oldFile, os);
   } catch (IOException ex) {
   ex.printStackTrace();
   }
```

# Java NIO

Java **Non-blocking Input Output** (NIO) consist of the following core components:

- **Channels**
- **Buffers**
- **Selectors**

- **Pipe** and **FileLock** are utility classes that are used in conjunction with the above three core components

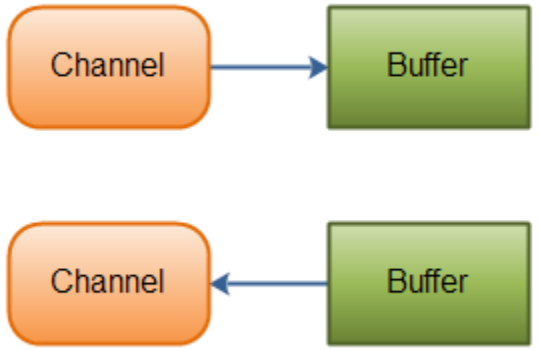
# Channel

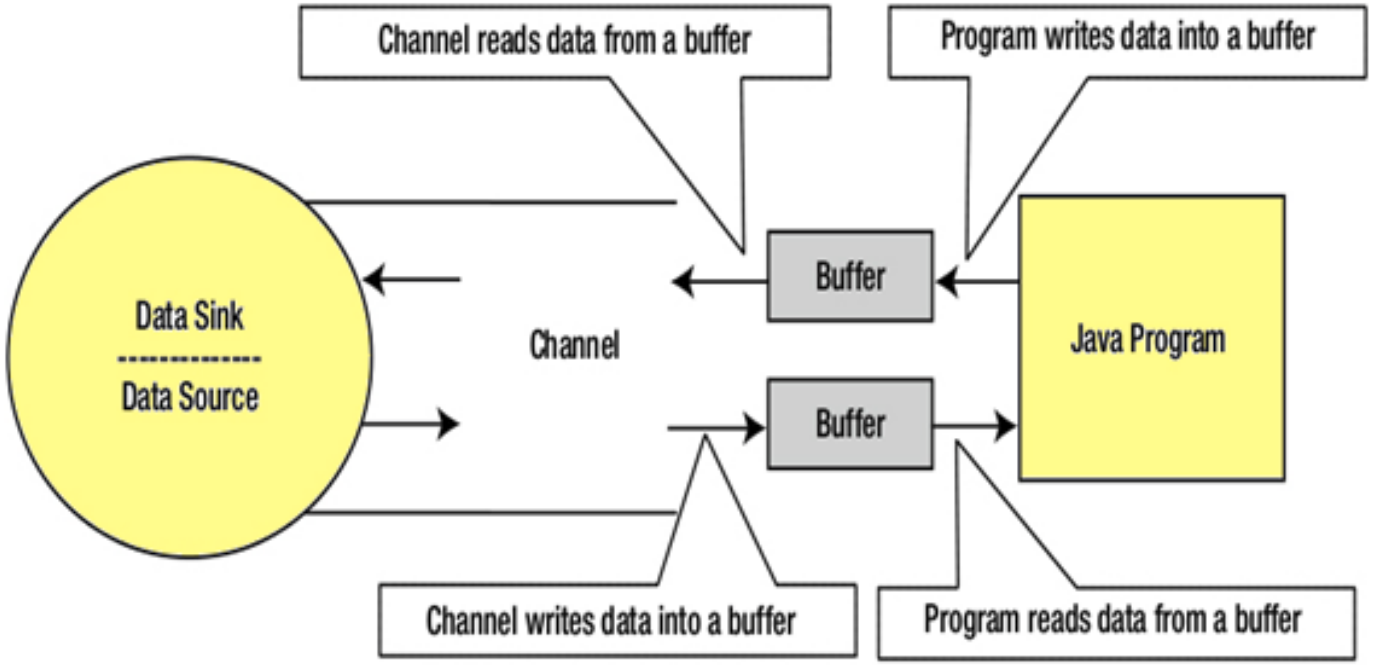
Channels partner with buffers to achieve high-performance I/O. Channels are gateways through which I/O services are accessed. Channels use byte buffers as the endpoints for sending and receiving data.

Java NIO Channels are similar to streams with a few differences:

- You can both read and write to a Channels. Streams are typically one-way (read or write).
- Channels can be read and written asynchronously.
- Channels always read to, or write from, a Buffer.

**Java NIO: Channels read data into Buffers, and Buffers write data into Channels**



# Channel Implementations

## Channel Implementations

Here are the most important Channel implementations in Java NIO:

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

The FileChannel reads data from and to files.

The DatagramChannel can read and write data over the network via UDP.

The SocketChannel can read and write data over the network via TCP.

The ServerSocketChannel allows you to listen for incoming TCP connections, like a web server does. For each incoming connection a SocketChannel is created.

# Buffers

A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

## Basic Buffer Usage

Using a Buffer to read and write data typically follows this little 4-step process:

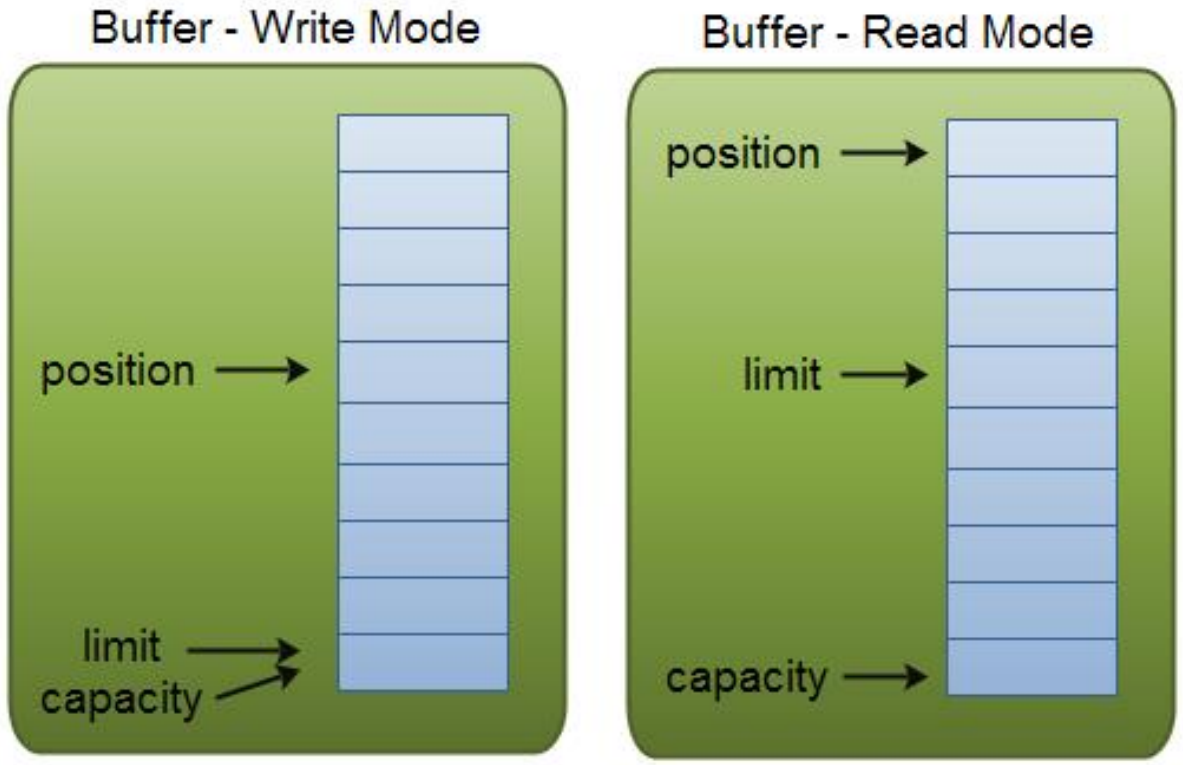
1. Write data into the Buffer
2. Call buffer.flip()
3. Read data out of the Buffer
4. Call buffer.clear() or buffer.compact()

# Buffer Capacity, Position and Limit

A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

A Buffer has three properties you need to be familiar with, in order to understand how a Buffer works. These are:

- capacity
- position
- Limit



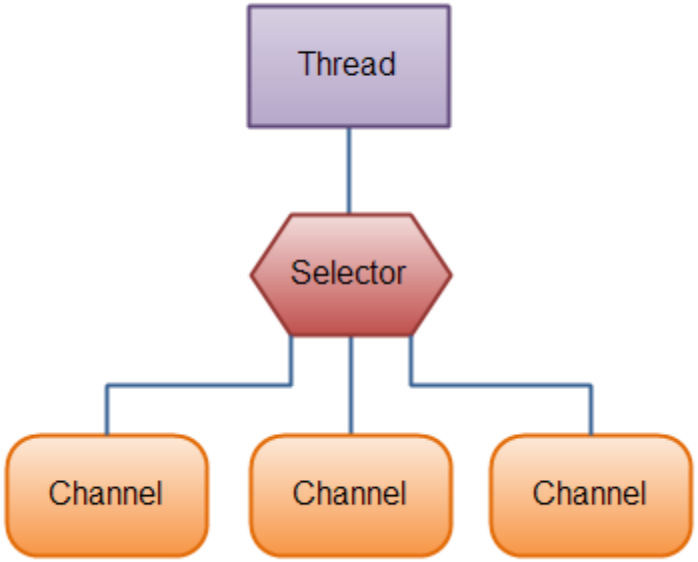
# Channel & Buffer Implementation

Following example uses a *FileChannel* to read some data into a Buffer:

```
RandomAccessFile aFile = new RandomAccessFile("f://data//sapient//nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while (bytesRead != -1) {
      System.out.println("Read " + bytesRead);
      buffer.flip();
      while( buffer.hasRemaining() ){
            System.out.print((char) buffer.get());
      }
      buffer.clear();
      bytesRead = inChannel.read(buffer);
}
aFile.close();
```

# Selectors

With a selector, we can use one thread instead of several to manage multiple channels. **Context-switching between threads is expensive for the operating system**, and additionally, **each thread takes up memory.**
Therefore, the fewer threads we use, the better. However, it's important to remember that **modern operating systems and CPU's keep getting better at multitasking**, so the overheads of multi-threading keep diminishing over time.




To use a Selector,  register the Channels with it. Then call it's *select()* method.

This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events.
Examples of events are incoming connection, data received etc.

# Selectors

**1. Creating a Selector**

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

**2. Registering Channels with the Selector**

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

The Channel must be in non-blocking mode to be used with a Selector.

This means that you cannot use FileChannel's with a Selector since FileChannel's cannot be switched into non-blocking mode. Socket channels will work fine though.

Notice the second parameter of the register() method. This is an "interest set", meaning what events you are interested in listening for in the Channel, via the Selector.

There are four different events you can listen for:
1.Connect  2. Accept  3. Read  4. Write

# Selectors

A channel that "fires an event" is also said to be "ready" for that event.
So, a channel that has connected successfully to another server is "connect ready".
A server socket channel which accepts an incoming connection is "accept" ready.
A channel that has data ready to be read is "read" ready.
A channel that is ready for you to write data to it, is "write" ready.

These four events are represented by the four SelectionKey constants:

1.SelectionKey.OP_CONNECT 2. SelectionKey.OP_ACCEPT
3.SelectionKey.OP_READ 3. SelectionKey.OP_WRITE

```
int ops = serverSocket.validOps();
```

The server socket channel's **validOps()** method returns an operation set identifying this channel's supported operations i.e., accepting new connections. The SelectionKey class has variables defining the operation sets.

Note: The OP_ACCEPT (socket-accept operation) is the only valid value for server socket channel.

# Selector Example: ServerSocketChannel

```
public class SelectorExample {

    public static void main (String [] args) throws IOException {
        // Get selector
        Selector selector = Selector.open();
        System.out.println("Selector open: " + selector.isOpen());
        // Get server socket channel and register with selector
        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        InetSocketAddress hostAddress = new InetSocketAddress("localhost", 5454);
        serverSocket.bind(hostAddress);
        serverSocket.configureBlocking(false);
        int ops = serverSocket.validOps();
        SelectionKey selectKey = serverSocket.register(selector, ops, null);

        for (;;) {
            System.out.println("Waiting for select...");
            int noOfKeys = selector.select();

            System.out.println("Number of selected keys: " + noOfKeys);

            Set selectedKeys = selector.selectedKeys();
            Iterator iterator = selectedKeys.iterator();
```

Contd…

# Selector Example: ServerSocketChannel

Contd…

```
while (iterator.hasNext()) {
    SelectionKey key = (SelectionKey) iterator.next();
    if (key.isAcceptable()) {
        // Accept the new client connection
        SocketChannel client = serverSocket.accept();
        client.configureBlocking(false);
        // Add the new connection to the selector
        client.register(selector, SelectionKey.OP_READ);
        System.out.println("Accepted new connection from client: " + client);
    }
    else if (key.isReadable()) {
        // Read the data from client
        SocketChannel client = (SocketChannel) ky.channel();
        ByteBuffer buffer = ByteBuffer.allocate(256);
        client.read(buffer);
        String output = new String(buffer.array()).trim();
        System.out.println("Message read from client: " + output);
        if (output.equals("Bye.")) {
            client.close();
            System.out.println("Client messages are complete; close.");
        }
    }
    iterator.remove();              }          }     }}
```

# Selector Example: SocketChannel

```
public class SocketClientExample {
    public static void main (String [] args)
            throws IOException, InterruptedException {
        InetSocketAddress hostAddress = new InetSocketAddress("localhost", 5454);
        SocketChannel client = SocketChannel.open(hostAddress);
        System.out.println("Client sending messages to server...");
        // Send messages to server
        String [] messages = new String [] {"Time goes fast.", "What now?", "Bye."};

        for (int i = 0; i < messages.length; i++) {
            byte [] message = new String(messages [i]).getBytes();
            ByteBuffer buffer = ByteBuffer.wrap(message);
            client.write(buffer);
            System.out.println(messages [i]);
            buffer.clear();
            Thread.sleep(3000);
        }
        client.close();
    }
}
```

From the output:
•Accepted new connection from client:
java.nio.channels.SocketChannel[connected local=/127.0.0.1:…]:
This message is displayed after the client program is started. This shows the client connection is accepted by the server.
•Message read from client: Time goes fast. This shows the earlier accepted client's first message is read.

# Thank You!