Using NIO in Java

Let me explain how to use NIO (New Input/Output) in Java with some practical examples. Java NIO is a non-blocking I/O API that provides a more efficient way to handle I/O operations compared to the traditional java.io package.

Here's a breakdown of the key components and a simple example:

Key NIO Components

1. **Buffers**: Containers for data (e.g., `ByteBuffer`).
2. **Channels**: Streams for reading/writing data (e.g., `FileChannel`, `SocketChannel`).
3. **Selectors**: For multiplexing multiple channels in non-blocking mode.

Basic Example: Reading a File with NIO

Here's how you can read a file using `FileChannel` and `ByteBuffer`:

```
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class NIOFileReader {
    public static void main(String[] args) {
        try {
            // Open a file channel
            FileChannel fileChannel = FileChannel.open(
                Paths.get("example.txt"),
                StandardOpenOption.READ
            );

            // Allocate a buffer
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            // Read data into buffer
            while (fileChannel.read(buffer) > 0) {
                // Flip buffer to prepare for reading
                buffer.flip();

                // Process the data
```

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

                // Clear buffer for next read
                buffer.clear();
            }

            // Close the channel
            fileChannel.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

How It Works

1. **FileChannel**: Opens a connection to the file.
2. **ByteBuffer**: Acts as a temporary storage for data. You allocate its size (e.g., 1024 bytes).
3. **Buffer Operations**:
   - `flip()`: Prepares the buffer to be read after writing data into it.
   - `clear()`: Resets the buffer for the next chunk of data.
4. **Reading**: The `read()` method fills the buffer, and you extract data byte by byte.

Writing to a File with NIO

Here's an example of writing data to a file:

```
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class NIOFileWriter {
    public static void main(String[] args) {
        try {
            // Open a file channel for writing
```

```
            FileChannel fileChannel = FileChannel.open(
                Paths.get("output.txt"),
                StandardOpenOption.CREATE,
                StandardOpenOption.WRITE
            );

            // Data to write
            String data = "Hello, NIO!";
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            // Put data into buffer
            buffer.put(data.getBytes());

            // Flip buffer to prepare for writing
            buffer.flip();

            // Write buffer to file
            fileChannel.write(buffer);

            // Close the channel
            fileChannel.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Non-Blocking Example with Selector

For network programming (e.g., a server), you can use `Selector` with `SocketChannel`:

```
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.net.InetSocketAddress;
import java.util.Iterator;
```

```java
public class NIOServer {
    public static void main(String[] args) throws Exception {
        // Open a selector
        Selector selector = Selector.open();

        // Open a server socket channel
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.bind(new InetSocketAddress(8080));
        serverChannel.configureBlocking(false);

        // Register channel with selector
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);

        System.out.println("Server started on port 8080");

        while (true) {
            selector.select(); // Block until events occur
            Iterator<SelectionKey> keys = selector.selectedKeys().iterator();

            while (keys.hasNext()) {
                SelectionKey key = keys.next();
                keys.remove();

                if (key.isAcceptable()) {
                    // Accept new connection
                    SocketChannel client = serverChannel.accept();
                    client.configureBlocking(false);
                    client.register(selector, SelectionKey.OP_READ);
                    System.out.println("Client connected");
                } else if (key.isReadable()) {
                    // Read data from client
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                    int bytesRead = client.read(buffer);
                    if (bytesRead == -1) {
                        client.close();
                    } else {
                        buffer.flip();
                        System.out.println("Received: " + new String(buffer.array(), 0, bytesRead));
                        buffer.clear();
```

```
                    }
                }
            }
        }
    }
}
```

Tips for Using NIO

- **Buffer Management**: Always `flip()` after writing to a buffer and before reading, and `clear()` when reusing it.
- **Non-Blocking**: Use `configureBlocking(false)` and `Selector` for scalable network apps.
- **Performance**: NIO shines in scenarios with many concurrent connections or large data transfers.

Let me know if you'd like a deeper dive into any specific part!