Java NIO

Stream Oriented vs. Buffer Oriented

Java IO being stream oriented means that you read one or more bytes at a time, from a stream. What you do with the read bytes is up to you. They are not cached anywhere. Furthermore, you cannot move forth and back in the data in a stream. If you need to move forth and back in the data read from a stream, you will need to cache it in a buffer first.

Java NIO's buffer oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move forth and back in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all the data you need in order to fully process it. And, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

Blocking vs. Non-blocking IO

Java IO's various streams are blocking. That means, that when a thread invokes a read() or write(), that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime.

Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.

The same is true for non-blocking writing. A thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the meantime.

What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

java.nio.file.Path

A `Path` instance represents a *path* in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

In many ways the `java.nio.file.Path` interface is similar to the `java.io.File` class, but there are some minor differences.

You can create a `Path` instance using a static method in the `Paths` class named `Paths.get()`.

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathExample {
    public static void main(String[] args) {
        Path path = Paths.get("c:\\data\\myfile.txt");
    }
}
```

Creating an absolute path is done by calling the `Paths.get()` factory method with the absolute file as parameter. Here is an example of creating a `Path` instance representing an absolute path:

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

The `Path` class can also be used to work with relative paths. You create a relative path using the `Paths.get(basePath, relativePath)` method.

```
Path path = Paths.get("d:\\data", "projects");
// Creates a Path instance pointing to `d:\data\projects`
```

```
Path path = Paths.get("d:\\data", "projects\\a-project\\myfile.txt"
// Creates a Path instance pointing to `d:\data\projects\a-project\
```

Path.relativize()

The Java Path method `relativize()` can create a new Path which represents the second Path relative to the first Path.

For instance, with the path `/data` and `/data/subdata/subsubdata/myfile.txt"`, the second path can be expressed as `/subdata/subsubdata/myfile.txt` relative to the first path.

```
Path basePath = Paths.get("/data");
Path path = Paths.get("/data/subdata/subsubdata/myfile.txt");

Path basePathToPath = basePath.relativize(path);
Path pathToBasePath = path.relativize(basePath);

System.out.println(basePathToPath);
System.out.println(pathToBasePath);

// Output:
// subdata/subsubdata/myfile.txt
// ../../..
```

Path.normalize()

The `normalize()` method of the `Path` interface can normalize a path. Normalizing means that it removes all the `.` and `..` codes in the middle of the path string, and resolves what path the path string refers to.

```
String originalPath =
"d:\\data\\projects\\a-project\\..\\another-project";

Path path1 = Paths.get(originalPath);
System.out.println("path1 = " + path1);

Path path2 = path1.normalize();
System.out.println("path2 = " + path2);

// Output:
// path1 = d:\data\projects\a-project\..\another-project
// path2 = d:\data\projects\another-project
```

java.nio.file.Files

The `Files` class provides several methods for manipulating files in the file system. It contains many methods, so check the JavaDoc too, if you need a method that is not described here.

The `java.nio.file.Files` class works with `java.nio.file.Path` instances, so you need to understand the `Path` class before you can work with the `Files` class.

Files.exists()

Since `Path` instances may point to paths that do not exist, you can use the `Files.exists()` method to determine if they do.

```
Path path = Paths.get("data/logging.properties");

boolean pathExists = Files.exists(path, new LinkOption[]{ LinkOptio
```

The second parameter is an array of options that influence how the `Files.exists()` determines if the path exists or not. In this example above the array contains the `LinkOption.NOFOLLOW_LINKS` which means that the `Files.exists()` method should not follow symbolic links in the file system to determine if the path exists.

Files.is*

The `Files` class provides a set of methods for verifying files properties.

- isRegularFile(Path path)
- isDirectory(Path path)
- isReadable(Path path)
- isWritable(Path path)
- isExecutable(Path path)

```
System.out.println(Files.isDirectory(Paths.get(System.getProperty("
System.out.println(Files.isRegularFile(Paths.get(System.getProperty
```

Files.create*

The `Files.createFile()` method creates a new file from a `Path` instance.

```
Path path = Paths.get("data/subdir/output.log");

try {
    Path newFile = Files.createFile(path);
} catch (FileAlreadyExistsException e) {
    // the file already exists.
} catch (IOException e) {
    // something else went wrong
}
```

The `Files.createDirectory()` method creates a new directory from a `Path` instance.

```
Path path = Paths.get("data/subdir");

try {
    Path newDir = Files.createDirectory(path);
} catch (FileAlreadyExistsException e) {
    // the directory already exists.
} catch (IOException e) {
    // something else went wrong
}
```

If creating the file or the directory succeeds, a `Path` instance is returned which points to the newly created path.

Files.copy()

The `Files.copy()` method copies a file from one path to another.

```
Path sourcePath = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");

try {
    Files.copy(sourcePath, destinationPath);
} catch (FileAlreadyExistsException e) {
    // destination file already exists
} catch (IOException e) {
    // something else went wrong
}
```

The example calls `Files.copy()`, passing two `Path` instances as parameters. This will result in the file referenced by the source path to be copied to the file referenced by the destination path.

Files.move()

The `Files.move()` method moves a file from one path to another.

```
Path sourcePath      = Paths.get("data/logging-copy.properties");
Path destinationPath = Paths.get("data/subdir/logging-moved.propert

try {
    Files.move(sourcePath, destinationPath,
        StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    //moving file failed.
}
```

First the source path and destination path are created. The source path points to the file to move, and the destination path points to where the file should be moved to. Then the `Files.move()` method is called. This results in the file being moved.

Notice the third parameter passed to `Files.move()` . This parameter tells the `Files.move()` method to overwrite any existing file at the destination path. This parameter is actually optional.

Files.delete()

The `Files.delete()` method can delete a file or directory.

```
Path path = Paths.get("data/subdir/logging-moved.properties");

try {
    Files.delete(path);
} catch (IOException e) {
    //deleting file failed
}
```

First the `Path` pointing to the file to delete is created. Second the `Files.delete()` method is called. If the `Files.delete()` fails to delete the file for some reason (e.g. the file or directory does not exist), an `IOException` is thrown.

Files.list()

The `Files.list()` method returns a `Stream<Path>` representing the entries in the directory. The listing is not recursive.

```
Path srcPath = Paths.get(src);
try (Stream<Path> stream = Files.list(srcPath)) {
    stream.forEach(System.out::println);
}
```

First the `Path` pointing to the directory to be listed is created. Second the `Files.list()` returns a stream of `Path` objects which is printed using a functional approach.

Files.readAllBytes()

Reads all the bytes from a file. The method ensures that the file is closed when all bytes have been read or an I/O error, or other runtime exception, is thrown.

Note that this method is intended for simple cases where it is convenient to read all bytes into a byte array. It is not intended for reading in large files.

```
Path src = Paths.get(filename);
byte[] buffer = Files.readAllBytes(src);
```

Files.readAllLines()

Read all lines from a file. Bytes from the file are decoded into characters using the UTF-8 charset.

Note that this method is intended for simple cases where it is convenient to read all lines at once. It is not intended for reading in large files.

```
Path src = Paths.get(filename);
List<String> lines = Files.readAllLines(src);
```

Files.lines()

Read all lines from a file as a Stream. Bytes from the file are decoded into characters using the UTF-8 charset. The returned stream contains a reference to an open file. The file is closed by closing the stream.

The file contents should not be modified during the execution of the terminal stream operation. Otherwise, the result of the terminal stream operation is undefined.

```
try (Stream<String> lines = Files.lines(Paths.get(filename))) {
    lines.forEach(System.out::println);
}
```

Files.write()

```
public static Path write(Path path, byte[] bytes, OpenOption... opt
```

Writes bytes to a file. The options parameter specifies how the file is created or opened. If no options are present, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing regular-file to a size of 0. All bytes in the byte array are written to the file. The method ensures that the file is closed when all bytes have been written (or an I/O error or other runtime exception is thrown).

```
public static Path write(Path path, Iterable<? extends CharSequence
```

Write lines of text to a file. Characters are encoded into bytes using the UTF-8 charset.

Dealing with large files

When dealing with large files, it is important to process them in chucks instead of fully loading them in memory before any processing.

Text files can be lazily read using the Files.lines() method returning a `Stream<String>` and lazily written using the Writer interface.

```
try (BufferedReader bufferedReader = Files.newBufferedReader(Paths.g
     BufferedWriter bufferedWriter = Files.newBufferedWriter(Paths.g
    bufferedReader.lines().forEach(line -> {
        try {
            bufferedWriter.write(line);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    });
}
```

Binary files, instead, can be processed in chucks using the FileChannel and ByteBuffer classes (which are part of the core classes of Java NIO).

```
try (FileChannel inChannel = FileChannel.open(Paths.get(src), Standa
     FileChannel outChannel = FileChannel.open(Paths.get(dst), Stand
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    while (inChannel.read(buffer) != -1) {
        buffer.flip();
        outChannel.write(buffer);
        buffer.clear();
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
```