# Modern File IO

## Java Developer

StayAhead Training; May, 2022

# Streams

## Modern File IO

- A stream is an object that represents data flowing into/out of an application
- Think of a stream as connecting the app with some ext. source of data
- A stream can either read or write but not both at the same time
- The java.io and java.nio packages contain stream classes
- The Files class that we're familiar with uses streams internally
- Since Java 8 there exists in the standard library classes that deal with data streams (java.util.stream) which should not be confused with IO streams

# The Different Types of Stream

## Modern File IO

- A byte stream is one that reads/writes bytes
- A byte stream is used to read/write binary files, e.g. images/audio/video etc.
- A character stream is one that reads/writes characters (two bytes)
- A character stream is used to read/write text files
- A buffered stream is one that reads/writes into a segment of memory (the buffer) to minimise disk access
- For example, if writing using a buffered stream, each write operation writes to the buffer and when the buffer is full it is flushed to disk

# The Different Types of Stream (a subset)

## Modern File IO

| | Input | Output |
|---|---|---|
| Bytes | `FileInputStream` | `FileOutputStream` |
| Characters | `FileReader` | `FileWriter` |
| Bytes (w/buffering) | `BufferedInputStream` | `BufferedOutputStream` |
| Characters (w/buffering) | `BufferedReader` | `BufferedWriter` |

# The Different Types of Stream

## Modern File IO

- Both FileOutputStream and FileWriter have write methods that accept an int
- Any int can be converted to a byte and any char can be converted to an int
- FileOutputStream will **truncate** chars from two bytes to one, e.g.

```
'Ʊ' as an int is 433
433 as binary is 0000000110110001 (truncated)

var out = new FileOutputStream("myfile.bin");
out.write('Ʊ'); // DON'T USE byte streams to write chars
```

# Decoration*

## Modern File IO

- Classes in the java.io package exploit the decorator pattern (beyond the scope of this course) but this makes instantiation painful, e.g.

```
var bufferedReader = new BufferedReader(
    new InputStreamReader(
        new FileInputStream("myfile.txt")
    )
);
```

- BufferedReader decorates (wraps/builds on top of) InputStreamReader which decorates FileInputStream

# The Path Class

## Modern File IO

- java.nio.Path is an interface whose instances represents file paths
- It has a static method, of, that is used to create instances, e.g.

```
var path = Path.of("myfile.txt");
```

- Avoid platform issues by creating a Path without separators, e.g.

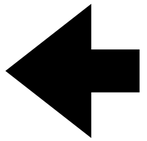
```
var path = Path.of("projects", "songs");
// path is projects/songs on a UNIX OS
```

- Path has many instance methods too

# The Files Class*

## Modern File IO

- java.nio.Files is a class containing static methods only that operate on files
- Excepting methods to read from and write to files it has methods incl.:
  - copy
  - createDirectory
  - createFile
  - delete
  - exists
  - isDirectory
  - isReadable
  - isWritable
  - move



All of these methods accept a Path or Paths as arguments

# Reading All Bytes/Lines From a File

## Modern File IO

- Reading bytes, e.g.

```
// the return value is a byte[]
var bytes = Files.readAllBytes(Path.of("myimage.png"));
```

- Reading lines, e.g.

```
// the return value is a List<String>
var lines = Files.readAllLines(Path.of("mynotes.txt"));
```

# Writing All Bytes/Lines To a File

## Modern File IO

- Writing bytes, e.g.

```
// bytes is a byte[]
Files.write(Path.of("mybytes.bin"), bytes);
```

- Writing lines, e.g.

```
// lines is a List of Strings
// newline characters are added automatically
Files.write(Path.of("bylines"), lines);
```

# Reading From a File Using a BufferedStream

## Modern File IO

- Use a buffered stream when the file is large and reading the entire contents at once may overwhelm memory
- Reading (characters) using a buffered stream, e.g.

```
var br = File.newBufferedReader(Path.of("myfile.txt));
var line = br.readLine();
while (line != null) {
  // process the line
  line = br.readLine();
}
br.close();
```

# Writing To a File Using a BufferedStream

## Modern File IO

- Use a buffered stream when you've to perform many write operations, e.g. you've to write every element existent in a large collection
- Writing (characters) using a buffered stream, e.g.

```
var lines = new String[] {"Salut", "Ca va?", "Au revoir"};
var bw = File.newBufferedWriter(Path.of("myfile.txt));
for (var line : lines) {
  bw.write(line + "\n"); // note the adding of the newline
}
bw.close();
```

# Parsing a File Using a Scanner

## Modern File IO

- Using the techniques describes thus far to read a text file yields String input
- But if the lines of the file include numbers then some parsing is required
- We've used a Scanner already to read from stdin
- stdin (System.in) is an input stream (in a static field of the System class)
- So we can build a Scanner that reads from a file, e.g.

```
var scanner = new Scanner(
    new FileInputStream("my file.txt"));
```

# Parsing a File Using a Scanner

## Modern File IO

- Assuming a CSV file with lines like this: `London,8.982,true`
- Reading (and parsing) using a Scanner, e.g.

```
scanner.useDelimiter(",|\\n"); // (regex) , or newline

var city = scanner.next();

var populationInMillions = scanner.nextDouble();

var isCapital = scanner.nextBoolean();
```