W

Java 8 API by Example: Strings, Numbers, Math and Files

March 25, 2015

Plenty of tutorials and articles cover the most important changes in Java 8 like lambda expressions and functional streams. But furthermore many existing classes have been enhanced in the JDK 8 API with useful features and methods.

This article covers some of those smaller changes in the Java 8 API - each described with easily understood code samples. Let's take a deeper look into Strings, Numbers, Math and Files.

Slicing Strings

Two new methods are available on the String class: `join` and `chars`. The first method joins any number of strings into a single string with the given delimiter:

```
String.join(":", "foobar", "foo", "bar");
// => foobar:foo:bar
```

The second method `chars` creates a stream for all characters of the string, so you can use stream operations upon those characters:

```
"foobar:foo:bar"
    .chars()
    .distinct()
    .mapToObj(c -> String.valueOf((char)c))
    .sorted()
    .collect(Collectors.joining());
// => :abfor
```

Not only strings but also regex patterns now benefit from streams. Instead of splitting strings into streams for each character we can split strings for any pattern and create a stream to work upon as shown in this example:

```
Pattern.compile(":")
    .splitAsStream("foobar:foo:bar")
    .filter(s -> s.contains("bar"))
    .sorted()
    .collect(Collectors.joining(":"));
// => bar:foobar
```

Additionally regex patterns can be converted into predicates. Those predicates can for example be used to filter a stream of strings:

```
Pattern pattern = Pattern.compile(".*@gmail\\.com");
Stream.of("bob@gmail.com", "alice@hotmail.com")
    .filter(pattern.asPredicate())
    .count();
// => 1
```

The above pattern accepts any string which ends with `@gmail.com` and is then used as a Java 8 `Predicate` to filter a stream of email addresses.

Crunching Numbers

Java 8 adds additional support for working with unsigned numbers. Numbers in Java had always been signed. Let's look at `Integer` for example:

An `int` represents a maximum of 2^{32} binary digits. Numbers in Java are per default signed, so the last binary digit represents the sign (0 = positive, 1 = negative). Thus the maximum positive signed `int` is 2^{31} - 1 starting with the decimal zero.

You can access this value via `Integer.MAX_VALUE`:

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

Java 8 adds support for parsing unsigned ints. Let's see how this works:

```
long maxUnsignedInt = (1l << 32) - 1;
String string = String.valueOf(maxUnsignedInt);
int unsignedInt = Integer.parseUnsignedInt(string, 10);
String string2 = Integer.toUnsignedString(unsignedInt, 10);
```

As you can see it's now possible to parse the maximum possible unsigned number 2^{32} - 1 into an integer. And you can also convert this number back into a string representing the unsigned number.

This wasn't possible before with `parseInt` as this example demonstrates:

```
try {
    Integer.parseInt(string, 10);
}
catch (NumberFormatException e) {
    System.err.println("could not parse signed int of " + maxUnsignedInt);
}
```

The number is not parseable as a signed int because it exceeds the maximum of 2^{31} - 1.

Do the Math

The utility class `Math` has been enhanced by a couple of new methods for handling number overflows. What does that mean? We've already seen that all number types have a maximum value. So what happens when the result of an arithmetic operation doesn't fit into its size?

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

As you can see a so called **integer overflow** happens which is normally not the desired behavior.

Java 8 adds support for strict math to handle this problem. `Math` has been extended by a couple of methods who all ends with `exact`, e.g. `addExact`. Those methods handle overflows properly by throwing an `ArithmeticException` when the result of the operation doesn't fit into the number type:

```
try {
    Math.addExact(Integer.MAX_VALUE, 1);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

The same exception might be thrown when trying to convert longs to int via `toIntExact`:

```
try {
    Math.toIntExact(Long.MAX_VALUE);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

Working with Files

The utility class `Files` was first introduced in Java 7 as part of Java NIO. The JDK 8 API adds a couple of additional methods which enables us to use functional streams with files. Let's deep-dive into a couple of code samples.

Listing files

The method `Files.list` streams all paths for a given directory, so we can use stream operations like `filter` and `sorted` upon the contents of the file system.

```
try (Stream<Path> stream = Files.list(Paths.get(""))) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> !path.startsWith("."))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("List: " + joined);
}
```

The above example lists all files for the current working directory, then maps each path to it's string representation. The result is then filtered, sorted and finally joined into a string. If you're not yet familiar with functional streams you should read my Java 8 Stream Tutorial.

You might have noticed that the creation of the stream is wrapped into a try/with statement. Streams implement `AutoCloseable` and in this case we really have to close the stream explicitly since it's backed by IO operations.

> *The returned stream encapsulates a DirectoryStream. If timely disposal of file system resources is required, the try-with-resources construct should be used to ensure that the stream's close method is invoked after the stream operations are completed.*

Finding files

The next example demonstrates how to find files in a directory or it's sub-directories.

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, attr) ->
        String.valueOf(path).endsWith(".js"))) {
    String joined = stream
        .sorted()
        .map(String::valueOf)
        .collect(Collectors.joining("; "));
    System.out.println("Found: " + joined);
}
```

The method `find` accepts three arguments: The directory path `start` is the initial starting point and `maxDepth` defines the maximum folder depth to be searched. The third argument is a matching predicate and defines the search logic. In the above example we search for all JavaScript files (filename ends with .js).

We can achieve the same behavior by utilizing the method `Files.walk`. Instead of passing a search predicate this method just walks over any file.

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.walk(start, maxDepth)) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> path.endsWith(".js"))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("walk(): " + joined);
}
```

In this example we use the stream operation `filter` to achieve the same behavior as in the previous example.

Reading and writing files

Reading text files into memory and writing strings into a text file in Java 8 is finally a simple task. No messing around with readers and writers. The method `Files.readAllLines` reads all lines of a given file into a list of strings. You can simply modify this list and write the lines into another file via `Files.write`:

```
List<String> lines = Files.readAllLines(Paths.get("res/nashorn1.js"));
lines.add("print('foobar');");
Files.write(Paths.get("res/nashorn1-modified.js"), lines);
```

Please keep in mind that those methods are not very memory-efficient because the whole file will be read into memory. The larger the file the more heap-size will be used.

As an memory-efficient alternative you could use the method `Files.lines`. Instead of reading all lines into memory at once, this method reads and streams each line one by one via functional streams.

```
try (Stream<String> stream = Files.lines(Paths.get("res/nashorn1.js"))) {
    stream
        .filter(line -> line.contains("print"))
        .map(String::trim)
        .forEach(System.out::println);
}
```

If you need more fine-grained control you can instead construct a new buffered reader:

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    System.out.println(reader.readLine());
}
```

Or in case you want to write to a file simply construct a buffered writer instead:

```
Path path = Paths.get("res/output.js");
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("print('Hello World');");
}
```

Buffered readers also have access to functional streams. The method `lines` construct a functional stream upon all lines denoted by the buffered reader:

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    long countPrints = reader
        .lines()
        .filter(line -> line.contains("print"))
```

```
            .count();
        System.out.println(countPrints);
    }
```

So as you can see Java 8 provides three simple ways to read the lines of a text file, making text file handling quite convenient.

Unfortunately you have to close functional file streams explicitly with try/with statements which makes the code samples still kinda cluttered. I would have expected that functional streams auto-close when calling a terminal operation like `count` or `collect` since you cannot call terminal operations twice on the same stream anyway.

I hope you've enjoyed this article. All code samples are hosted on GitHub along with plenty of other code snippets from all the Java 8 articles of my blog. If this post was kinda useful to you feel free to star the repo and follow me on Twitter.

Keep on coding!

Follow @winterbe   Follow @winterbe_   Tweet

Benjamin is Software Engineer, Full Stack Developer at Pondus, an excited runner and table foosball player. Get in touch on Twitter, Google+ and GitHub.

Read More

Recent | All Posts | Java | Tutorials

Integrating React.js into Existing jQuery Web Applications

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap

Java 8 Concurrency Tutorial: Synchronization and Locks