Java 12

1.. String Class New Methods

Java 12 comes with two new methods in the String class. The first one – **indent** adjusts the indentation of each line based on the integer parameter. If the parameter is greater than zero, new spaces will be inserted at the beginning of each line. On the other hand, if the parameter is less than zero, it removes spaces from the begging of each line. If a given line does not contain sufficient white space, then all leading white space characters are removed. Now, let's take a look at a basic example. Firstly, we'll indent the text with four spaces, and then we'll remove the whole indentation:

```
String text = "Hello Baeldung!\nThis is Java 12 article.";

text = text.indent(4);
System.out.println(text);

text = text.indent(-10);
System.out.println(text);
```

The output looks like the following:

```
    Hello Baeldung!
    This is Java 12 article.

Hello Baeldung!
This is Java 12 article.
```

Note that even if we passed value -10, which exceeds our indent count, only the spaces were affected. Other characters are left intact. The second new method is **transform**. It accepts a single argument function as a parameter that will be applied to the string. As an example, let's use the transform method to revert the string.

```
@Test
public void givenString_thenRevertValue() {
    String text = "Baeldung";
    String transformed = text.transform(value ->
      new StringBuilder(value).reverse().toString()
    );

    assertEquals("gnudleaB", transformed);
}
```

2.. File New Method

Java 12 introduced a new mismatch method in the **nio.file.Files** utility class. The method is used to compare two files and find the position of the first mismatched byte in their contents. The return value will be in the inclusive range of 0L up to the byte size of the smaller file or -1L if the files are identical. Now let's take a look at two examples. In the first one, we'll create two identical files and try to find a mismatch. The return value should be -1L.

```
public static long mismatch(Path path, Path path2) throws IOException
```

```
@Test
public void givenIdenticalFiles_thenShouldNotFindMismatch() {
    Path filePath1 = Files.createTempFile("file1", ".txt");
    Path filePath2 = Files.createTempFile("file2", ".txt");
    Files.writeString(filePath1, "Java 12 Article");
    Files.writeString(filePath2, "Java 12 Article");

    long mismatch = Files.mismatch(filePath1, filePath2);
    assertEquals(-1, mismatch);
}
```

In the second example, we'll create two files with "Java 12 Article" and "Java 12 Tutorial" contents. The mismatch method should return 8L as it's the first different byte.

```
@Test
public void givenDifferentFiles_thenShouldFindMismatch() {
    Path filePath3 = Files.createTempFile("file3", ".txt");
    Path filePath4 = Files.createTempFile("file4", ".txt");
    Files.writeString(filePath3, "Java 12 Article");
    Files.writeString(filePath4, "Java 12 Tutorial");

    long mismatch = Files.mismatch(filePath3, filePath4);
    assertEquals(8, mismatch);
}
```

3.. Teeing Collector

A new teeing collector was introduced in Java 12 as an addition to the Collectors class.

```
Collector<T, ?, R> teeing(Collector<? super T, ?, R1> downstream1,
  Collector<? super T, ?, R2> downstream2, BiFunction<? super R1, ? super R2, R> merger)
```

It is a composite of two downstream collectors. Every element is processed by both downstream collectors. Then their results are passed to the merge function and transformed into the final result. The example usage of teeing collector is counting an average from a set of numbers. The first collector parameter will sum up the values, and the second one will give us the count of all numbers. The merge function will take these results and count the average.

```
@Test
public void givenSetOfNumbers_thenCalculateAverage() {
    double mean = Stream.of(1, 2, 3, 4, 5)
      .collect(Collectors.teeing(Collectors.summingDouble(i -> i),
        Collectors.counting(), (sum, count) -> sum / count));
    assertEquals(3.0, mean);
}
```

4.. Compact Number Formatting

Java 12 comes with a new number formatter – the **CompactNumberFormat**. It's designed to represent a number in a shorter form, based on the patterns provided by a given locale.

We can get its instance via the **getCompactNumberInstance** method in NumberFormat class.

```
public static NumberFormat getCompactNumberInstance(Locale locale, NumberFormat.Style formatStyle)
```

As mentioned before, the locale parameter is responsible for providing proper format patterns. The format style can be either SHORT or LONG. For a better understanding of the format styles, let's consider number 1000 in the US locale. The SHORT style would format it as "10K", and the LONG one would do it as "10 thousand".

Now let's take a look at an example that'll take the numbers of likes under this article and compact it with two different styles:

```
@Test
public void givenNumber_thenCompactValues() {
    NumberFormat likesShort =
      NumberFormat.getCompactNumberInstance(new Locale("en", "US"), NumberFormat.Style.SHORT);
    likesShort.setMaximumFractionDigits(2);
    assertEquals("2.59K", likesShort.format(2592));

    NumberFormat likesLong =
      NumberFormat.getCompactNumberInstance(new Locale("en", "US"), NumberFormat.Style.LONG);
    likesLong.setMaximumFractionDigits(2);
    assertEquals("2.59 thousand", likesLong.format(2592));
}
```

5.. Preview Changes

Some of the new features are available only as a preview. To enable them, we need to switch proper settings in the IDE or explicitly tell the compiler to use preview features.

```
javac -Xlint:preview --enable-preview -source 12 src/main/java/File.java
```

5.1 Switch Expressions (Preview)

The most popular feature introduced in Java 12 is the Switch Expressions. As a demonstration, let's compare the old and new switch statements. We'll use them to distinguish working days from weekend days based on the DayOfWeek enum from the LocalDate instance. Firstly, let's look and at the old syntax.

```
DayOfWeek dayOfWeek = LocalDate.now().getDayOfWeek();
String typeOfDay = "";
switch (dayOfWeek) {
    case MONDAY:
    case TUESDAY:
    case WEDNESDAY:
    case THURSDAY:
    case FRIDAY:
        typeOfDay = "Working Day";
        break;
    case SATURDAY:
    case SUNDAY:
        typeOfDay = "Day Off";
}
```

And now, let's see the same logic witch switch expressions:

```
typeOfDay = switch (dayOfWeek) {
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Working Day";
    case SATURDAY, SUNDAY -> "Day Off";
};
```

New switch statements are not only more compact and readable but they also remove the need for break statements. The code execution will not fall through after the first match. Another notable difference is that we can assign a switch statement directly to the variable. It was not possible previously. It’s also possible to execute code in switch expressions without returning any value.

```
switch (dayOfWeek) {
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> System.out.println("Working Day");
    case SATURDAY, SUNDAY -> System.out.println("Day Off");
}
```

More complex logic should be wrapped with curly braces.

```
case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> {
    // more logic
    System.out.println("Working Day")
}
```

Note that we can choose between the old and new syntax. Java 12 switch expressions are only an extension, not a replacement.

5.2 Pattern Matching for instanceof (Preview)

Another preview feature introduced in Java 12 is pattern matching for instanceof. In previous Java versions, when using, for example, if statements together with instanceof, we would have to explicitly typecast the object to access its features.

```
Object obj = "Hello World!";
if (obj instanceof String) {
    String s = (String) obj;
    int length = s.length();
}
```

With Java 12, we can declare the new typecasted variable directly in the statement:

```
if (obj instanceof String s) {
    int length = s.length();
}
```

The compiler will automatically inject the typecasted *String s* variable for us.

6.. JVM Changes

Java 12 comes with several JVM enhancements. In this section, we’ll have a quick look at a few most important ones.

6.1 Shenandoah: A Low-Pause-Time Garbage Collector

Shenandoah is an experimental garbage collection (GC) algorithm, for now not included in the default Java 12 builds. It reduces the GC pause times by doing evacuation work simultaneously with the running Java threads. This means that with Shenandoah, pause times are not dependent on the heap's size and should be consistent. Garbage collecting a 200 GB heap or a 2 GB heap should have a similar low pause behavior. Shenandoah will become part of mainline JDK builds since version 15.

6.2 Microbenchmark Suite

Java 12 introduces a suite of around 100 microbenchmark tests to the JDK source code. These tests will allow for continuous performance testing on a JVM and will become useful for every developer wishing to work on the JVM itself or create a new microbenchmark.

6.3 Default CDS Archives

The Class Data Sharing (CDS) feature helps reduce the startup time and memory footprint between multiple Java Virtual Machines. It uses a built-time generated default class list that contains the selected core library classes. The change that came with Java 12 is that the CDS archive is enabled by default. To run programs with CDS turned off we need to set the **Xshare** flag to off.

```
java -Xshare:off HelloWorld.java
```

Note, that this could delay the startup time of the program.