Java 12

*Java 12 comes with four new methods in the String class.*

1. *Indent - 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 beginning of each line. If a given line does not contain sufficient white space, then all leading white space characters are removed****.*

**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.

1. *Transform- It accepts a single argument function as a parameter that will be applied to the string.*

*Ex –*

@Test **public** **void** **givenString\_thenRevertValue**() {

**String** text = "Baeldung";

**String** transformed = text.transform(value ->

**new** **StringBuilder**(value)

.reverse().toString());

assertEquals("gnudleaB", transformed);

}

String s = "Java,Python,Angular";

List result = s.transform(s1 -> {return Arrays.asList(s1.split(","));});

System.out.println(result);

**Output**

[Java, Python, Angular]

1. *Optional<String> describeConstable(): This method will return an Optional object containing a descriptor for the String instance.*

*String message = "Welcome!";*

*Optional<String> opOfMessage = message.describeConstable();*

*System.out.println(opOfMessage);*

*Output*

*Optional[Welcome!]*

1. *String resolveConstantDesc​(MethodHandles.Lookup lookup): This method will return a String object which is the descriptor for the invoking String instance.*

*String message = "Welcome!";*

*String constantDesc = message.resolveConstantDesc(MethodHandles.lookup());*

*System.out.println(constantDesc);*

*Output*

*Welcome!*

*File::mismatch Method*

*Java 12 introduced a new mismatch method in the nio.file.Files utility class:*

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

*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.*

@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);

}

@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);

}

*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);

}

*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)

*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”.*

@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));

}

*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:*

*Switch Expressions (Preview)-*

typeOfDay = **switch** (dayOfWeek) {

**case** MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Working Day"; **case** SATURDAY, SUNDAY -> "Day Off";

};

*New switch statements are more compact and readable. 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. More complex logic should be wrapped with curly braces.*

*Pattern Matching for instanceof (Preview)*

*Another preview feature introduced in Java 12 is*[*pattern matching for instanceof.*](https://www.baeldung.com/java-pattern-matching-instanceof)

*In previous Java versions, when using, for example, if statements together with [instanceof](https://www.baeldung.com/java-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.*

***Shenandoah: A Low-Pause-Time Garbage Collection***

*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.*

***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.*

***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:*