Java 12 Features

**Date 04/04/2023**

# Overview of Java 12 (Mar 19, 2019)

1. Switch Expression
2. String API Enhancements
3. Teeing Collector
4. Compact Number Formatting
5. Pattern Matching For InstanceOf

# Switch Expression

Java 12 introduces expressions to Switch statement and released it as a preview feature. Following are the changes introduced in case of new switch with expressions,

1. No fall through.
2. No break statement required to prevent fall through.
3. A single case can have multiple constant labels.
4. Default case is compulsory now.

New switch statements are not only 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.

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

## Example

public class \_01\_Switch\_Expression\_Demo1 {

public static void main(String[] args) {

*DayOfWeek* dayOfWeek = LocalDate.*now*().getDayOfWeek();

System.***out***.println(dayOfWeek.toString());

System.***out***.println(*getDayDetail*(dayOfWeek.toString()));

}

public static String getDayDetail(String day) {

String today = switch (day) {

case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" -> "Week Day";

case "SATURDAY", "SUNDAY" -> "Weekend";

default -> "Invalid Day";

};

return today;

}

}

# String API Enhancements

After we got a few new String methods in Java 11 and the Files.readString() and writeString() methods, the JDK developers extended both classes again for Java 12.

## String.indent()

To indent a string, we used to write a small helper method that put the desired number of spaces in front of the String. If it should work over multiple lines, the method became correspondingly complex. Java 12 has such a method built-in: String.indent().

n > 0 - insert space at the begining of each line.

n < 0 - remove space at the begining of each line.

n < 0 and n < available spaces - remove all leading space of each line.

n = 0 - no change.

public class \_01\_String\_Indent\_Demo1 {

public static void main(String[] args) {

var contents = "This is java12 \narticle about features \nand enhancements";

System.***out***.println("\*\*\* Contents without Indent \*\*\*" + "\n" + contents);

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println("\*\*\* Contents with +ve Indent \*\*\*" + "\n" + contents.indent(5));

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println("\*\*\* Contents with 0 Indent \*\*\*" + "\n" + contents.indent(0));

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.***out***.println("\*\*\* Contents with -ve Indent \*\*\*" + "\n" + contents.indent(-2));

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

## String.transform()

The new String.transform() method applies an arbitrary function to a String and returns the function's return value. The advantage of String.transform() is that the function to be applied can be determined dynamically at runtime.

public class \_02\_String\_Transform\_Demo1 {

public static void main(String[] args) {

var upperCase = "java".transform(String::toUpperCase);

System.***out***.println(upperCase);

var number = "12345".transform(Integer::*valueOf*);

System.***out***.println(number);

BigDecimal decimal = "1234567891011121314151617181920".transform(BigDecimal::new);

System.***out***.println(decimal);

var text = "Optimistic";

var transformedText = text.transform(value -> new StringBuilder(value).reverse().toString());

System.***out***.println(transformedText);

}

}

## Files.mismatch()

You can use the Files.mismatch() method to compare the contents of two files.

The method returns -1 if both files are the same. Otherwise, it returns the position of the first byte at which both files differ. If one of the files ends before a difference is detected, the length of that file is returned.

public class \_03\_File\_String\_Mismatch\_Demo1 {

public static void main(String[] args) throws IOException {

*verifyIdenticalFiles*();

*verifyDifferentFiles*();

}

private static void verifyIdenticalFiles() throws IOException {

Path path1 = Files.*createTempFile*("file1", ".txt");

Path path2 = Files.*createTempFile*("file2", ".txt");

Files.*writeString*(path1, "Java 12 Features");

Files.*writeString*(path2, "Java 12 Features");

System.***out***.println(Files.*mismatch*(path1, path2));

}

private static void verifyDifferentFiles() throws IOException {

Path path1 = Files.*createTempFile*("file1", ".txt");

Path path2 = Files.*createTempFile*("file2", ".txt");

Files.*writeString*(path1, "Java 12 Article");

Files.*writeString*(path2, "Java 12 Features");

System.***out***.println(Files.*mismatch*(path1, path2));

}

}

## String constants

Since Java 12, String class implements two additional interfaces java.lang.constant.Constable and java.lang.constant.ConstantDesc.

String class also introduces two additional low-level methods describeConstable() and resolveConstantDesc(MethodHandles.Lookup).

They are low-level APIs meant for libraries and tools providing bytecode parsing and generation functionality.

Just to note, a Constable type is one whose values are constants that can be represented in the constant pool of a Java class file as described in JVMS 4.4, and whose instances can describe themselves nominally as a ConstantDesc.

resolveConstantDesc() is similar to describeConstable() with the difference being that this method returns an instance of ConstantDesc instead.

**Optional<String> describeConstable() method**

Returns Optional Object containing description of String instance.

**resolveConstantDesc​(MethodHandles.Lookup lookup) method**

Returns descriptor instance string of given string.

public class \_04\_String\_Constant\_Demo1 {

public static void main(String[] args) {

var text = "Program";

Optional<String> optionalText = text.describeConstable();

System.***out***.println(optionalText);

var constDesc = text.resolveConstantDesc(MethodHandles.*lookup*());

System.***out***.println(constDesc);

}

}

# Teeing Collector

A new teeing collector was introduced in Java 12 as an addition to the Collectors class. 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. It is the new collector utility introduced in the Streams API. This collector has three arguments - Two collectors and a Bi-function. All input values are passed to each collector and the result is available in the Bi-function. A teeing collector has been exposed as a static method Collectors::teeing. This collector forwards its input to two other collectors before merging their results with a function. teeing(Collector, Collector, BiFunction) accepts two collectors and a function to merge their results.

## Example

In the following example source code, we want to determine the difference from largest to smallest number from a stream of random numbers (we use Optional.orElseThrow() introduced in Java 10 to avoid a "code smell" blaming),

Stream<Integer> numbers = new Random().ints(100).boxed();

int min = numbers.collect(Collectors.minBy(Integer::compareTo)).orElseThrow();

int max = numbers.collect(Collectors.maxBy(Integer::compareTo)).orElseThrow();

long range = (long) max - min;

**The program compiles but aborts at runtime with an exception,**

Exception in thread "main" java.lang.IllegalStateException:

stream has already been operated upon or closed

at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)

at java.base/java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:578)

at eu.happycoders.sandbox.TeeingCollectorTest.main(TeeingCollectorTest.java:12)

We can overcome the error with the Teeing collector as below,

public class \_01\_Teeing\_Collector\_Demo1 {

public static void main(String[] args) {

Stream<Integer> numbers = new Random().ints(100).boxed();

long range = numbers.collect(Collectors.*teeing*(Collectors.*minBy*(Integer::compareTo),

Collectors.*maxBy*(Integer::compareTo), (min, max) -> (long) max.orElseThrow() - min.orElseThrow()));

System.***out***.println(range);

}

}

# 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. We can create a formatter for the so-called "compact number formatting". This is a form that is easy for humans to read, such as "2M" or "3 billion".

## Example

public class \_01\_CompactNumberFormat\_Demo1 {

public static void main(String[] args) {

NumberFormat nfShort = NumberFormat.*getCompactNumberInstance*(Locale.***US***, NumberFormat.*Style*.***SHORT***);

NumberFormat nfLong = NumberFormat.*getCompactNumberInstance*(Locale.***US***, NumberFormat.*Style*.***LONG***);

System.***out***.println("Short Form of 1000 ==> "+ nfShort.format(1000));

System.***out***.println("Long Form of 1000 ===> "+ nfLong.format(1000));

System.***out***.println("Short Form of 2000000 ==> "+ nfShort.format(2000000));

System.***out***.println("Long Form of 2000000 ===> "+ nfLong.format(2000000));

System.***out***.println("Short Form of 3456789000 ==> "+ nfShort.format(3456789000L));

System.***out***.println("Long Form of 3456789000 ===> "+ nfLong.format(3456789000L));

}

}

# JEP 305 - Pattern Matching for instanceof

The 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”;

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

Object obj = “World”;

if (Obj instanceof String s)

{

int length = s.length();

}

## Example

public class \_01\_InstanceOf\_Demo1 {

public static void main(String[] args) {

Object value = "Java12 features";

int strLength = 0;

if (value instanceof String str) {

strLength = str.length();

}

System.***out***.println(strLength);

}

}

# JEP 189 - Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)

Shenandoah is an experimental garbage collection (GC) algorithm, for now not included in the default Java 12 builds. An experimental Low-Pause-Time Garbage Collector, Shenandoah is introduced to reduce the GC pause time. It works in parallel with running java threads. This helps to reduce the dependency of GC over heap size and makes it consistent. Now Garbage collection pause time would be similar for 2 MB as well as for 2 GB heap space. Shenandoah is expected to be part of main JAVA release with Java 15.

# References

<https://www.oracle.com/java/technologies/javase/12-relnote-issues.html>

<https://www.happycoders.eu/java/java-12-features/#Complete_List_of_All_Changes_in_Java_12>

<https://www.tutorialspoint.com/java12/java12_garbage_collection_enhancements.htm>

<https://www.baeldung.com/java-12-new-features>

<https://www.digitalocean.com/community/tutorials/java-12-features>

<https://howtodoinjava.com/java12/new-features-enhancements/>