Java 10 Features

**Date 17/03/2023**

# Overview of Java 10 (Mar 10, 2018)

Features of Java 10 as below,

1. JEP 286 - Local Variable Type Inference
2. Immutable Collections
3. JEP 307 - Parallel Full GC for G1
4. JEP 310 - Application CDS
5. JEP 317 - Experimental JIT compiler for Linux

Other Enhancements of Java 10,

1. JEP 316 - Heap Allocation on Alternative Memory Devices
2. JEP 314 - Additional Unicode Language-Tag Extensions
3. JEP 304 - Garbage Collector Interface
4. JEP 319 - Root Certificates
5. JEP 312 - Thread Local Handshakes
6. JEP 313 - Native Header Generation Tool Removal (Javah)
7. JEP 296 - Consolidate JDK Forest into Single Repository

# JEP 286 - Local Variable Type Inference

## Intro

It is one of the cool features in Java 10. But it has its limitations. It allows us to define a variable using var and without specifying the type of it. The Compiler infer / identify the type of the variable using the value provided.

## Limitations

This var variable declaration is restricted to local variables and can’t be used globally.

No type inference in case of member variable, method parameters, return values.

Local variable should be initialized at time of declaration otherwise compiler will not be infer and will throw error.

Local variable inference is available inside initialization block of loop statements.

No dynamic type changes. Once type of local variable is inferred it cannot be changed.

## Benefits

No runtime overhead. As compiler infers the type based on value provided, there is no performance loss.

Complex boilerplate code can be reduced using local variable type inference.

## Example

String name = “Java”;

var name = “Java”;

Map<String, Integer> map = new HashMap<>();

var map = new HashMap<String, Integer>();

# Collections Enhancements

## Immutable Collections

### Intro

In Java 10, we have enhancements to the Collections. We have option to make the collections as unmodifiable collections. The following methods were introduced as part of Java 10,

1. unmodifiableList ()
2. unmodifiableSet ()
3. unmodifiableMap ()
4. unmodifiableCollection ()

### Examples

public class \_01\_ImmutableCollections\_List\_Demo1 {

var list = new ArrayList<Integer>();

list.add(1);

list.add(2);

list.add(3);

var immutableList = Collections.unmodifiableList(list);

System.out.println(immutableList);

}

}

**Output**

[1, 2, 3]

If we are try to add an element via the immutableList, then we will get an **UnsupportedOperationException** whereas adding an element to the original collection is allowed.

immutableList. add(4); // **java.lang.UnsupportedOperationException**

list.add(4); // **Valid**

## Copy Collections

### Intro

In Java 10, we also have facility to create immutable copies of collections. For this purpose, we have static methods introduced in the Interfaces as below,

1. List.CopyOf()
2. Set.CopyOf()
3. Map.CopyOf()

If we create the collections copy and modifying the original collection, will not affect the copy.

### Examples

public class \_01\_ImmutableCollections\_List\_Demo1 {

var list = new ArrayList<Integer>();

list.add(1);

list.add(2);

list.add(3);

var immutableList = List.CopyOf(list);

System.out.println(immutableList);

}

}

**Output**

[1, 2, 3]

If we are try to add an element via the immutableList, then we will get an **UnsupportedOperationException** whereas adding an element to the original collection is allowed.

immutableList. add(4); // **java.lang.UnsupportedOperationException**

list.add(4); // **Valid**

## Collectors Unmodifiable

### Intro

The Stream API’s Collectors created using the Collectors.toUnmodifiableList(), Collectors.toUnmodifiableSet() and Collectors.toUnmodifiableMap() allows stream elements to get collect into immutable lists, sets and maps.

### Example

public class \_03\_ImmutableCollections\_List\_Demo3 {

public static void main(String[] args) {

var list = IntStream.rangeClosed(1, 3).boxed().collect(Collectors.toUnmodifiableList());

System.out.println(list);

}

}

Output

[1, 2, 3]

If we are try to add an element to the list, then we will get an **UnsupportedOperationException**.

list.add(4); //  **java.lang.UnsupportedOperationException**

# Optional Enhancement

The optional class introduced in Java 8, provides the get () method to retrieve the value wrapped by the Optional. Before calling get (), you should always check with isPresent() whether a value exists. If the optional is Empty get() would otherwise throw NoSuchElementException.

## OrElseThrow ()

As part of Java 10, a nice solution provided with the method OrElseThrow(). This method is an exact copy of the get() method only with the difference in name. It is clear from the method name that it will throw an exception. The method OrElseThrow() is now a preferred alternative for get() method.

## Example

public class **01\_Optional\_OrElseThrow\_Demo1** {

public static void main(String[] args) {

// With Non-Empty List

List<Integer> list1 = List.of(1, 2, 3, 4, 5);

Optional<Integer> firstEvenNum1 = Optional.of(list1.stream().filter(i -> i % 2 == 0).findFirst().orElseThrow());

System.out.println(firstEvenNum1);

// With Empty List

List<Integer> list2 = List.of();

Optional<Integer> firstEvenNum2 = Optional.of(list2.stream().filter(i -> i / 2 == 0).findFirst().orElseThrow());

System.out.println(firstEvenNum2);

}

}

# JEP 307 - Parallel Full GC for G1

With JDK 9, the Garbage-First (G1) garbage collector has replaced the parallel collector as the default GC.

While the parallel GC could perform a full garbage collection (i.e., cleaning up all regions of the heap) in parallel with the running application, this was not possible with G1 until now. G1 had to temporarily stop the application ("stop-the-world"), leading to noticeable latencies.

Since G1 was designed to avoid full collections as much as possible, this rarely posed a problem.

In Java 10, the full garbage collection of the G1 collector has now also been parallelized. The worst-case latencies (pause times) reach those of the parallel collector.

# JEP 310 - Application CDS (Class Data Sharing)

It is not known by many people around the world.

## Class Data Sharing (Existing Feature from Java 5)

When JVM starts it loads the classes in memory as a preliminary step. In case there are multiple jars having multiple classes, an evident lag appears for the first request. In serverless architecture, such a lag can delay the boot time which is a critical operation in such an architecture. When a JVM starts, it loads the JDK class library from the file system (up to JDK 8 from the jre/lib/rt.jar file; since JDK 9 from the jmod files in the jmods directory). In the process, the class files are extracted from the archives, converted into an architecture-specific binary form, and stored in the main memory of the JVM process.

If multiple JVMs are started on the same machine, this process repeats. Each JVM keeps its copy of the class library in memory.

Class-Data Sharing, introduced in JDK 5, allows a set of classes to be pre-processed into a shared archive file that can then be memory-mapped at runtime to reduce startup time. It can also reduce dynamic memory footprint when multiple JVMs share the same archive file.

Currently CDS only allows the bootstrap class loader to load archived classes. Application CDS allows the built-in system class loader, the built-in platform class loader, and custom class loaders to load archived classes.

## Class-data sharing ("CDS") has two goals

1. Reducing the startup time of the JVM.
2. Reducing the JVM's memory footprint.

Class-data sharing works as follows:

Using the command java -Xshare:dump, you initially create a file called classes.jsa (JSA stands for Java Shared Archive). This file contains the complete class library in a binary format for the current architecture.

When the JVM is started, the operating system "maps" this file into the JVM's memory using memory mapped I/O. Firstly, this is faster than loading the jar or jmod files. And secondly, the operating system loads the file into RAM only once, providing each JVM process with a read-only view of the same memory area.

## Application Class-Data Sharing

Application class-data sharing (also called "Application CDS" or "AppCDS") extends CDS by the possibility to store not only the JDK class library but also the classes of your application in a JSA file and to share them among the JVM processes.

# JEP 317 - Experimental JIT compiler for Linux

JIT compiler is written in C++ and is used to convert Java into Byte Code. Now Java 10 has option to enable an experimental Java based JIT compiler, Graal to be used instead of standard JIT compiler.

Since Java 9, the Graal Compiler (a Java compiler written in Java) has been supplied as an experimental Ahead-of-Time (AOT) compiler. This allows a Java program to be compiled into a native executable file (e.g., an exe file on Windows). Enabling it to be used as an experimental JIT compiler is one of the initiatives of Project Metropolis, and is the next step in investigating the feasibility of a Java-based JIT for the JDK.

In Java 10, JEP 317 created the possibility of using Graal also as a just-in-time (JIT) compiler - at least on the Linux/x64 platform. For this purpose, Graal uses the JVM Compiler Interface (JVMCI) introduced in JDK 9. Graal is already in the JDK, so enabling it as an experimental JIT will primarily be a testing and debugging effort.

You can activate Graal via the following option on the java command line:

**-XX: +UnlockExperimentalVMOptions -XX: +UseJVMCICompiler**

# References

<https://www.oracle.com/java/technologies/javase/10-relnote-issues.html>

<https://www.happycoders.eu/java/java-10-features/#Experimental_Java-Based_JIT_Compiler>

<https://www.tutorialspoint.com/java10/java10_jit_compiler.htm>

<https://www.baeldung.com/java-10-overview>

<https://www.digitalocean.com/community/tutorials/java-10-features>

<https://www.techadvisor.com/article/738813/java-10-features-whats-new-in-java-10.html>

<https://javarevisited.blogspot.com/2018/03/java-10-released-10-new-features-java.html#axzz7wBcgB1Ed>

<https://howtodoinjava.com/java10/java10-features/>