Java 14 Features

**Date 15/04/2023**

# Overview of Java 14 (Mar 17, 2020)

1. JEP - 361 Switch Expression Standardized
2. JEP - 368 Text Blocks Enhancement
3. JEP - 358 Helpful NullPointerException
4. JEP - 305 Pattern Matching for instanceof()
5. JEP - 359 Records
6. JEP - 343 JPackage Tool

# JEP - 361 Switch Expression Standardized

Switch Expressions is the second language enhancement from Project Amber to reach production status (the first was "var" in Java 10). Switch expressions allow a much more concise notation than before using arrow notation.

(Switch Expressions were first introduced as a preview feature in Java 12. In the second preview in Java 13, the keyword break, used initially to return values, was replaced by yield. Due to positive feedback, Switch Expressions were released as a final feature in Java 14 by JDK Enhancement Proposal 361 without further changes.)

With Java 14, switch expression now is a standard feature.

1. Each case block can return a value using yield statement.
2. In case of enum, default case can be skipped. In other cases, default case is required.

# JEP - 368 Text Blocks Enhancement

Java 13 introduces text blocks to handle multiline strings like JSON/XML/HTML etc as is a preview feature. With Java 14, we've second preview of Text Blocks. Now Text Block have following enhancements,

* **\** - indicates an end of the line in case new line is to be started.
* **\s** - indicates a single space.

public class \_01\_Text\_Block\_Enhance\_Demo1 {

public static void main(String[] args) {

var text1 = """

Does you know \

java 14 \

has new features

""";

var text2 = """

line1

line2\s

line3

""";

var text3 = "line1\nline2 \nline3\n";

System.***out***.println(text1);

System.***out***.println(text2);

System.***out***.println(text3);

System.***out***.println(text2.equals(text3));

}

}

# JEP - 358 Helpful NullPointerException

Java 14 introduces NullPointerException with helpful information. Null Pointer Exceptions are a nightmare for any developer. Previously, until Java 13, it was tricky to debug the infamous NPEs. Developers had to fall onto other debugging tools or manually figure the variable/method that was null since the stack trace would only show the line number. Java 14 introduced a new JVM feature which gives better insights with a more descriptive stack.

public class \_01\_Helpful\_NullPointerException\_Demo1 {

*@SuppressWarnings*("null")

public static void main(String[] args) {

int []arr = null;

arr[0] = 1;

System.***out***.println(arr[0]);

}

}

# JEP - 305 Pattern Matching for instanceof (Preview)

The second level preview in Java 14 is "Pattern Matching for instanceof". This feature, defined in JDK Enhancement Proposal 305, eliminates the annoying need to cast to the target type after an instanceof check.

The easiest way to illustrate this is with an example.

The following code gets the Object obj. If it is a String and longer than five characters, it should be converted to uppercase and printed. If it is an Integer, it should be squared and printed.

To do this, we need to perform a cast in lines 3 and 8,

Object obj = getObject();

if (obj instanceof String) {

String s = (String) obj;

if (s.length() > 5) {

System.out.println(s.toUpperCase());

}

} else if (obj instanceof Integer) {

Integer i = (Integer) obj;

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

}

Many of us have become so accustomed to this style that we no longer even question it. But there is a better way!

Starting with Java 14, we can omit the casts and write the code as follows instead. After an instanceof statement, we can now specify a variable name. If obj is of the specified type, it is bound to the new variable name; this new variable is then of the specified target type and visible in the "then block". We can even go one step further and combine the if statements of lines 1 and 2,

if (obj instanceof String s && s.length() > 5) {

System.out.println(s.toUpperCase());

} else if (obj instanceof Integer i) {

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

}

public class \_01\_InstanceOf\_Demo1 {

public static void main(String[] args) {

Object value = "Java14 features";

if (value instanceof String str && str.length() > 5) {

System.***out***.println(str.toUpperCase());

System.***out***.println(str.length());

} else {

System.***out***.println("The condition doesn't met!!");

}

}

}

# JEP - 359 Records

The first new preview feature in Java 14 is Records, defined in JDK Enhancement Proposal 359.

Records were introduced to reduce repetitive boilerplate code in data model POJOs. They simplify day to day development, improve efficiency and greatly minimize the risk of human error.

Records improves developer productivity by providing a compact syntax for declaring classes which act as transparent carriers for immutable data. New class called as record (AKA data class), it is final class, not abstract, and all of its fields are final as well.

As we can see, we are making use of a new keyword, record, here. This simple declaration will automatically add a constructor, getters, equals, hashCode and toString methods for us.

A record offers a compact syntax for a class with only final fields. These are set in the constructor and can be read via access methods.

Here is a simple example,

**record Point(int x, int y) {}**

This one-liner creates a class Point with,

final instance fields x and y,

a constructor setting both fields,

and access methods x() and y() to read the fields.

Point can be used as follows,

**Point p = new Point(3, 5);**

**int x = p.x();**

**int y = p.y();**

The equals(), hashCode() and toString() methods are generated automatically for records.

Records can have static fields and methods but no instance fields. They can implement interfaces but cannot inherit from other classes. They are implicitly final, so you can't inherit from them either.

public class \_01\_Records\_Demo1 {

record AuthenticateUser(String username, String password) {

}

public static void main(String[] args) {

AuthenticateUser authenticateUser1 = new AuthenticateUser("admin", "password");

System.***out***.println(authenticateUser1.username());

System.***out***.println(authenticateUser1.password());

AuthenticateUser authenticateUser2 = new AuthenticateUser("admin", "password");

System.***out***.println(authenticateUser2.username());

System.***out***.println(authenticateUser2.password());

System.***out***.println(authenticateUser1.equals(authenticateUser2));

System.***out***.println(authenticateUser1.hashCode());

System.***out***.println(authenticateUser1.toString());

}

}

A record is a data class that stores pure data. The idea behind introducing records is to quickly create simple and concise classes devoid of boilerplate code.

Normally a POJO class in Java would require you to implement equals(), hashCode() , the getters and setters methods. While some IDEs support auto-generation of such classes, the code is still verbose.

**record Author(){}**

**//or**

**record Author (String name, String topic) {}**

The Java compiler will generate a constructor, private final fields, accessors, equals/hashCode and toString methods automatically. The auto-generated getter methods of the above class are name() and topic().

To look into the generated code, use javap Author after you’ve compiled the program using javac. The following illustration shows the generated class for record Author (String name, String topic) {}:

Furthermore, we can add additional fields, methods, and constructor to the record in the following way:

record Author (int id, String name, String topic) {

static int followers;

public static String followerCount() {

return "Followers are "+ followers;

}

public String description(){

return "Author "+ name + " writes on "+ topic;

}

public Author{

if (id < 0) {

throw new IllegalArgumentException( "id must be greater than 0.");

}

}

}

The additional constructor defined inside the record is called a Compact constructor. It doesn’t consist of any parameters and is just an extension of the canonical constructor.

A compact constructor wouldn’t be generated as a separate constructor by the compiler. Instead, it is used for validation cases and would be invoked at the start of the main constructor.

Few important things to note about Records,

* A record can neither extend a class nor it can be extended by another class. It’s a final class.
* Records cannot be abstract
* Records cannot extend any other class and cannot define instance fields inside the body. Instance fields must be defined in the state description only
* Declared fields are private and final
* The body of a record allows static fields and methods

public class \_02\_Records\_Demo2 {

record Author(int bookId, String authorName, String bookName) {

static int *followers* = 0;

public static String followers() {

return "Followers count " + *followers*;

}

public String details() {

return "Author " + authorName + " has published a book called " + bookName;

}

public Author {

if (bookId < 0) {

throw new IllegalArgumentException("id must be greater than 0.");

}

}

}

*@SuppressWarnings*("static-access")

public static void main(String[] args) {

Author author = new Author(101, "John", "Time Traveller");

System.***out***.println(author.bookId());

System.***out***.println(author.authorName());

System.***out***.println(author.bookName());

System.***out***.println(author.*followers*());

System.***out***.println(author.details());

System.***out***.println(author.hashCode());

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

}

}

public class \_03\_Records\_Demo3 {

record Author(String authorName, List<String> bookName) {

}

public static void main(String[] args) {

var booksName = new ArrayList<String>(Arrays.*asList*("Inner Engineering"));

var author = new Author("John", booksName);

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

booksName.add("Time Traveller");

System.***out***.println(author.bookName());

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

}

}

interface Detail {

String getDetail();

}

public class \_04\_Records\_Demo4 {

record Author(String authorName, String bookName) implements Detail {

*@Override*

public String getDetail() {

return "Author " + authorName + " has written the book " + bookName;

}

}

public static void main(String[] args) {

var author = new Author("John", "Inner Engineering");

System.***out***.println(author.getDetail());

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

}

}

# JEP - 343 JPackage Tool

The jpackage tool is being developed based on JDK Enhancement Proposal 343. You can use this tool to create a platform-specific installer for a Java application, which in turn installs the application and the JRE required for it.

Platform-specific means that the installer feels familiar to users of a particular platform. On Windows, for example, this is an .msi or .exe file that is launched by double-clicking it. For macOS, a .pkg or .dmg file. And for Linux, a .deb or .rpm file.

The functionality is based on the javapackager tool, which was included since JDK 8, but removed in Java 11 along with JavaFX.

jpackage will be ready for production in Java 16. In the corresponding article of this series, I will show how to use the tool.

Java 14 introduces a new packaging tool, jpackage based on javapackager. javapackager was introduced in Java 8 and was part of JavaFX kit. As JavaFX is split from Java from 11 version, this packaging tool is no more available in standard offering.

This new tool is developed to provide native installer for an operating system. For example, an msi/exe for windows, pkg/dmg for MacOS, deb/rpm for Linux and so on. Without this tool, developer generally share a jar file which a user has to run within own JVM.

Developer can use jlink to compress the required JDK modules to minimum modules and use the jpackage to create a lightweight image.

Traditionally, to deliver Java code, an application developer would simply send out a JAR file that the user was supposed to run inside their own JVM.

However, users rather expected an installer that they'd double click to install the package on their native platforms, such as Windows or macOS.

This JEP aims to do precisely that. Developers can use jlink to condense the JDK down to the minimum required modules, and then use this packaging tool to create a lightweight image that can be installed as an exe on Windows or a dmg on a macOS.