Modern Java
Master All NEW Features in Java by Coding it

Dilip Sundarraj

About Me

- Dilip
- Building Software’s since 2008
- Teaching in **UDEMY** Since 2016

Course Objectives

- This course covers all the new features in the Modern Java since **Java 9**.
  - Local Variable Type Inference (LVTI), Record Types, Enhanced Switch, TextBlocks, Sealed Classes, Pattern Matching, JPMS and more.
- This course will be continuously updated with all the new features.
- All the concepts will be explored by actually coding it.

Targeted Audience

- Experienced Java Developers.
- Java Developers who is interested in exploring the latest features in Java.
- Java Developers who likes to stay up to date.
- Hands-On Oriented course.

Source Code

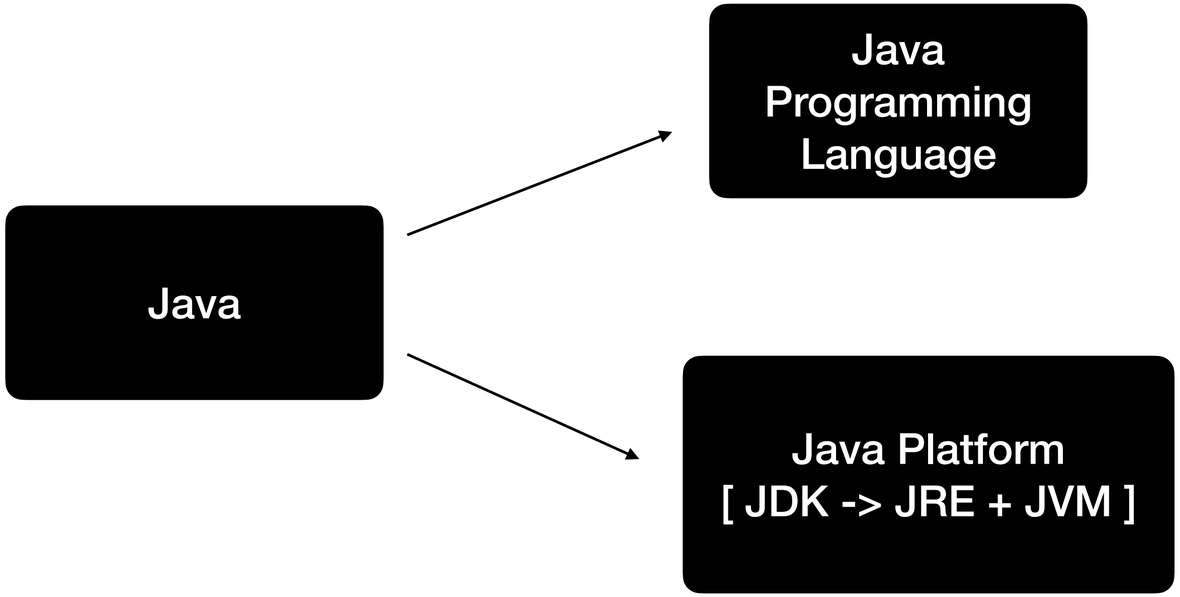
Thank You!

Prerequisites

- Java 20 or above.
- Prior Java Experience is a must
  - Functional programming concepts such as Lambdas, Streams API.
- Experience working with Gradle.
- Experience Writing JUnit tests.
- **Intellij** or any other IDE.

Introduction to the Modern Java

What does “Java” mean?



Evolution of Java = Modern Java

- Functional Programming (Java 8)
  - Lambdas, Streams
  - CompletableFuture
- Release of Java Modules (Java 9)
  - Java Platform Module System
  - Java Runtime Libraries are also modularized.
- Six Month Release Cycle(Java 10 onwards)
  - Release new features every 6 months

Fantastic Features - Java 9 & Beyond

- Java Platform Module System (JPMS)
- Local Variable Type Inference (LVTI)
- Record Types
- Enhanced Switch
- TextBlocks
- Sealed Classes
- Pattern Matching
- Virtual Threads

Java Release Model

- Java is an open source language and its managed under the OpenJDK project.
  - Java 7 was the first version that was relesed under the open source license.
- OpenJDK project is manitained by Oracle, Redhat and the community.
- There are different OpenJDK providers:
  - Orcale, Eclipse Adoptium(Temurin) , Amazon(Corretto), Azul Systems (Zulu), IBM, Microsoft, Red Hat, and SAP
- All of these vendors have this concept of LTS (Long Term Support) releases.
  - Primary applications run on these LTS releases even though new Java version is released every 6 months

Fantastic Features - Java 9 & Beyond

- Java Platform Module System (JPMS)
- Local Variable Type Inference (LVTI)
- Record Types
- Enhanced Switch
- TextBlocks
- Sealed Classes
- Pattern Matching

Local Variable Type Inference(LVTI) using “var”

- Historically Java was always labelled as a verbose language.
- LVTI feature is specifically introduced to address the verbosity concern.
- **var** is a reserved type name `var var = "Java";` ✓

Before LVTI

```
List<String> names1 = List.of("adam", "dilip");
```

After LVTI

```
var names = List.of("adam", "dilip");
```

Variable name

Reserved type name

Local Variable Type Inference(LVTI) using “var”

Before LVTI

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

After LVTI

```
var usersLists = new HashMap<Integer, Map<String,String>>();
```

```
var map : Map<String, List<String>>
        = Map.ofEntries(Map.entry( k: "a", List.of("adam", "alex")));
```

Inlay hints

Limitations of using “var”

- “null” value cannot be assigned to a “var” as the type cannot be inferred.
  - var x = null ✗
- Changing the type is not allowed.

```
var s = "Hello, World”;

s=3 // Changing the type to integer is not allowed
```

- “var” cannot be used as a class property
- “var” cannot be used as function argument.

Collection Factory Methods

- These factory methods are created to ease the creation of collection.

```
var list = List.of(1,2, 3);

var set = Set.of("a", "b", "c");

var sampleMap = Map.of(1, "One", 2 , "2");

var sampleMap1 = Map.ofEntries(entry(1, "One"), entry(2, "two"));
```

- All these collections are immutable ones.
- These factory methods were introduced in **Java 9**.

TextBlocks - Enhanced the power of String

- This feature got released in Java 15.

- Java's **String** was always considered very primitive compared to other programming languages.

```
var home = "Dilip\"s Home '";
```

```
var multiLine = "This is a\n" +
        "    multiline string\n" +
        "with newlines inside";
```

"TextBlocks" are primarily introduced to make Strings better.

TextBlocks

- Begin and end with the triplequotes.
- A new line is must for a textblock.
- Indentation is based on the the closing triplequotes.
- This code much cleaner.

```
var multiLine = """
    This is a
    multiline string
    with newlines inside
    """;
```

Text Blocks - Real Time Examples

- Sql:

```
var sql = """
        SELECT * FROM employee
        WHERE first_name = 'Dilip'
        AND last_name = 'Sundarraj'
        """;
```

- JSON:

```
var json =  """
        {
          "order_id": 123456,
          "status": "DELIVERED",
          "final_charge": 999.99,
          "order_line_items": [{
            "item_name": "iphone 14",
            "quantity": 1
          }]
        }
        """
        ;
```

Enhanced Switch

- Enhanced Switch got released as part of Java 14.
- Enhanced Switch is an “expression”.
  - The switch statement returns a value.

Enhanced Switch

- Function that returns the number of days based on the Month and Year.

Old "switch"

```
public static int getDays(Month month, int year) {
1   int noOfDays = 0;
    switch (month) {
        case APRIL:
        case JUNE:
2       case SEPTEMBER:
        case NOVEMBER:
            noOfDays = 30;
             break;
4       case FEBRUARY:
            noOfDays = Year.isLeap(year) ? 29 : 28;
             break;
5       default:
             noOfDays = 31;
    }
3   return noOfDays;

}
```

Enhanced "switch"

```
public static int getDaysV2(Month month, int year) {
    1 return switch (month) {
        2 case SEPTEMBER, APRIL, JUNE, NOVEMBER -> 30;
          case FEBRUARY -> Year.isLeap(year) ? 29 : 28;
          default -> 31;
      };
  }
```

1. Switch is an expression, so **return** is placed on the **switch** itself.
2. Multiple case lablels are allowed.
3. **break** is replaced by **arrow** and the **value**.

Records

- Records are a new type of class with a **record** keyword instead of the class keyword.
- Record classes are immutable data holders.
  - They are intended to just hold data.

```
public record Product(String name, BigDecimal cost, String type) { }
```

- This is available from Java 17.
- Record classes are **final**, no inheritance is supported.
- Record classes have autogenerated equals( ), hashcode( ) and toString( ) functions.

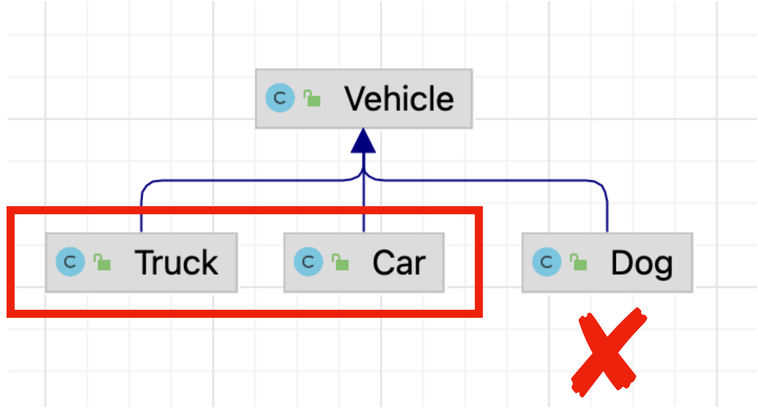
Records - Benefits

- Domain classes in Java can be represented in simpler form.
  - Now its part of the language itself.
- Avoids the need to write boiler plate code for domain classes.
  - This involves constructors, getter(), setter(), hashCode(), equals() and toString().
- Avoids the needs to rely on other libraries such as Lombok.
  - This requires us to have the knowledge of the different annotations.

Sealed Classes/Interfaces

- This concept was generally available from Java 17.
- Allow inheritance by permission.

- Java is very open by default.
  - Any class can extend other class as long its accessible.
- **Sealed classes/interfaces comes into play to prevent this kind of behavior.**



Sealed Classes/Interfaces

```
public sealed class Vehicle permits Car, Truck {}
```

```
public final class Car extends Vehicle { }
```

```
public final class Truck extends Vehicle{ }
```

```
public class Dog extends Vehicle{ }
```

Subclasses of sealed classes

- final

```
public final class Car extends Vehicle { }
```

  - This ensures no other class can extend the Car class.

- sealed

```
public sealed class Car extends Vehicle permits FlyingCar { }
```

  - This ensures that inheritance is allowed but controlled for classes thats defined after the permits keyword.

- non-sealed

```
public non-sealed class Car extends Vehicle { }
```

  - In this case, any class can extend the subclass **Car**. This basically disables the controlled inheritance behavior.

Why Pattern Matching?

- Code is verbose.
- Step 1 and Step 2 can be combined into one step.

```
public String pattern(Object o) {
    if (o instanceof Integer) {
        var i = (Integer) o;
        return "Integer:" + i;
    }
    if (o instanceof String) {
        var i = (String) o;
        return "String of length:" + i.length();
    }
    return "Not a String or Integer";

}
```

1 Check the type

2 cast and create a variable

3 Act on the variable

Pattern Matching to the rescue to write concise and elegant code.

What is Pattern Matching ?

- **Checks** the type, **Casts** the type and creates a **binding variable** if its a match.
- Act on the variable.
- Pattern Matching using **instanceOf** is available from Java 16.
- This particular type of pattern is also **Type Patterns**.
- Other Patterns:
  - Record Patterns
  - Guarded Patterns

```
public String patternMatchUsingInstanceOf(Object o) {
     // i is the binding variable.
    if (o instanceof Integer i) {  1

        return "Integer:" + I;  2

    }
    if (o instanceof String i) {
        return "String of length:" + i.length();
    }
    return "Not a String or Integer";

}
```

Pattern Matching - Different Approaches

Using “instanceOf”

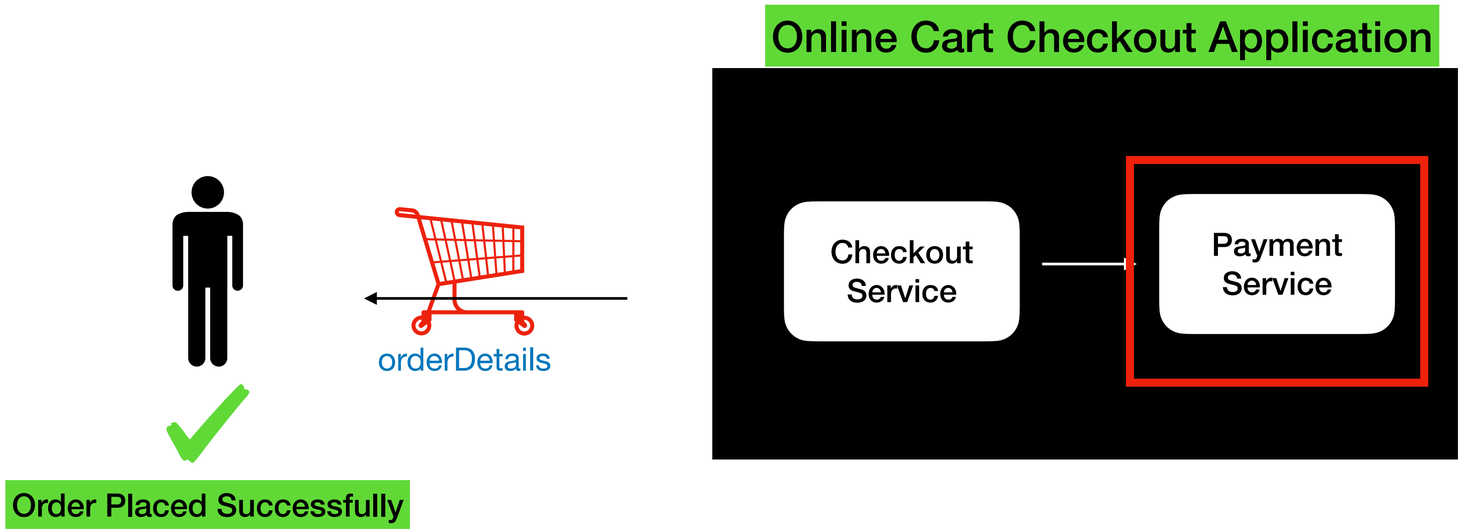
```
public String patternMatchUsingInstanceOf(Object o) {
     // i is the binding variable.
    if (o instanceof Integer i) {
       return "Integer:" + I;
    }
    if (o instanceof String i) {
        return "String of length:" + i.length();
    }
    return "Not a String or Integer";

}
```

Using “switch”

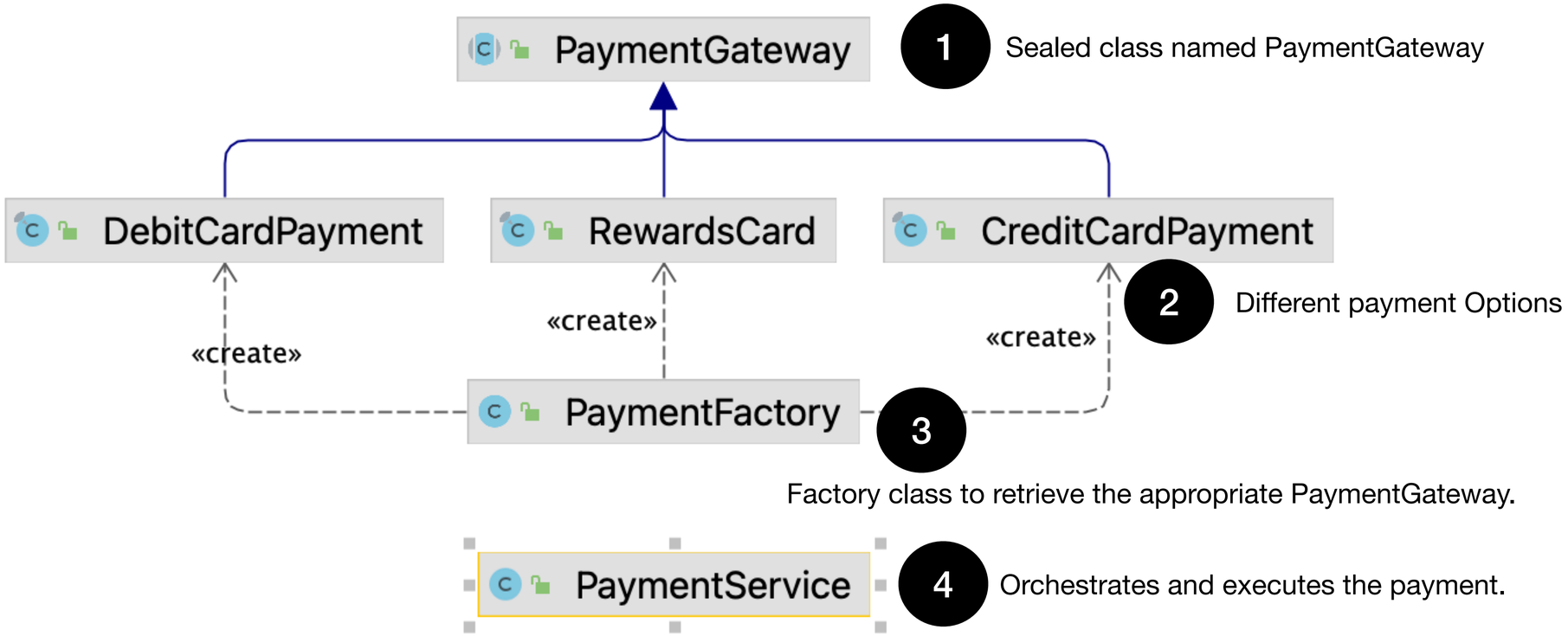
```
public String patternMatchingSwitch(Object o) {
    return switch (o) {
        case String s -> "String of length:" + s.length();
        case Integer i -> "Integer:" + i;
        case Number i -> "Integer:" + i;
        case null, default -> "Not a String or Integer";
    };
}
```

- Each case statement applies a pattern match.
- We can use lambdas for the case body.
- **case null** condition is a nice addition.

Online Cart Checkout Application



PaymentService Design



Simple Web Server

- Java18 released a **Simple Web Server.**
  - It’s part of the Java Distribution that’s installed in our machine.
  - This **webserver** servers files and folders from your machine.
- This can be primarily used for prototyping, testing and debugging.
- We can launch the webserver by running the **jwebserver** in the terminal.
  - It supports GET and HEAD requests only.
  - HTTPS is not supported.
  - Support HTTP/1.1.

HTTP 2 Client

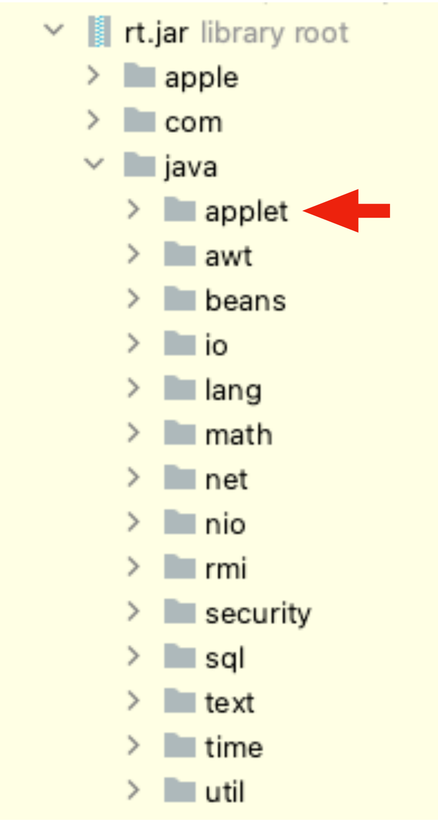
- New HTTP Client API got released in Java 11
  - It has the support for HTTP/2 and Websockets
- The Client has the support for build clients in both Synchronous and Asynchronous mode.

Java Platform Module System (JPMS) or Project Jigsaw

- This is a new concept that got introduced in **Java 9**.
- **JPMS** is introduced to package & deploy your applications in a better way.
- Why JPMS ?
  - Modularize the JDK.

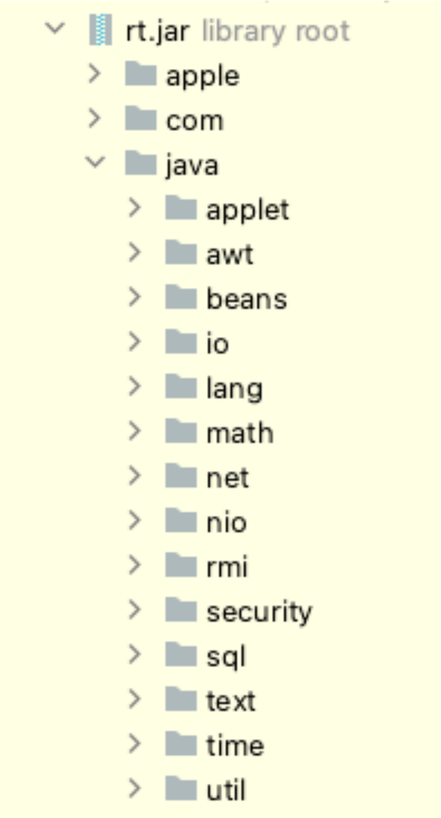
Why JPMS or Modules?

- 1. Modularize JDK
  - Until Java8 we used to have rt.jar file which is a giant monolith jar.
    - But in mostcases you dont need everything.
      - Eg., We dont need classes to be under the **applet** package for building a RestFul API.
  - By modularizing the JDK, the application can control on adding just the modules the app requires.



Modularized JDK

Until Java 8



From Java 9

JDK is structured as modules.

Why JPMS or Modules ?

- 2. Secure coding
  - By default , public is too open.
    - New restrictive controls are available to restrict access to certain internal classes.
      - Eg., We can control the packages in a library that can exposed to the client(library user).
  - In today’s words, any class can be accessed and modified using Reflection.
    - Using modules, we can restrict Reflection acces during runtime.

Benefits of JPMS

- Smaller jar files.
  - Reduce the process footprint.
  - Improve the application startuptime.
- Clean Separation of boundaries.
- Stricter access control.