**What is an Enum?**

[Enums](https://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html)are the preferred way to define fixed values you want to use in your code. They are a special type of Java class and contain a group of unchangeable variables.

For example:

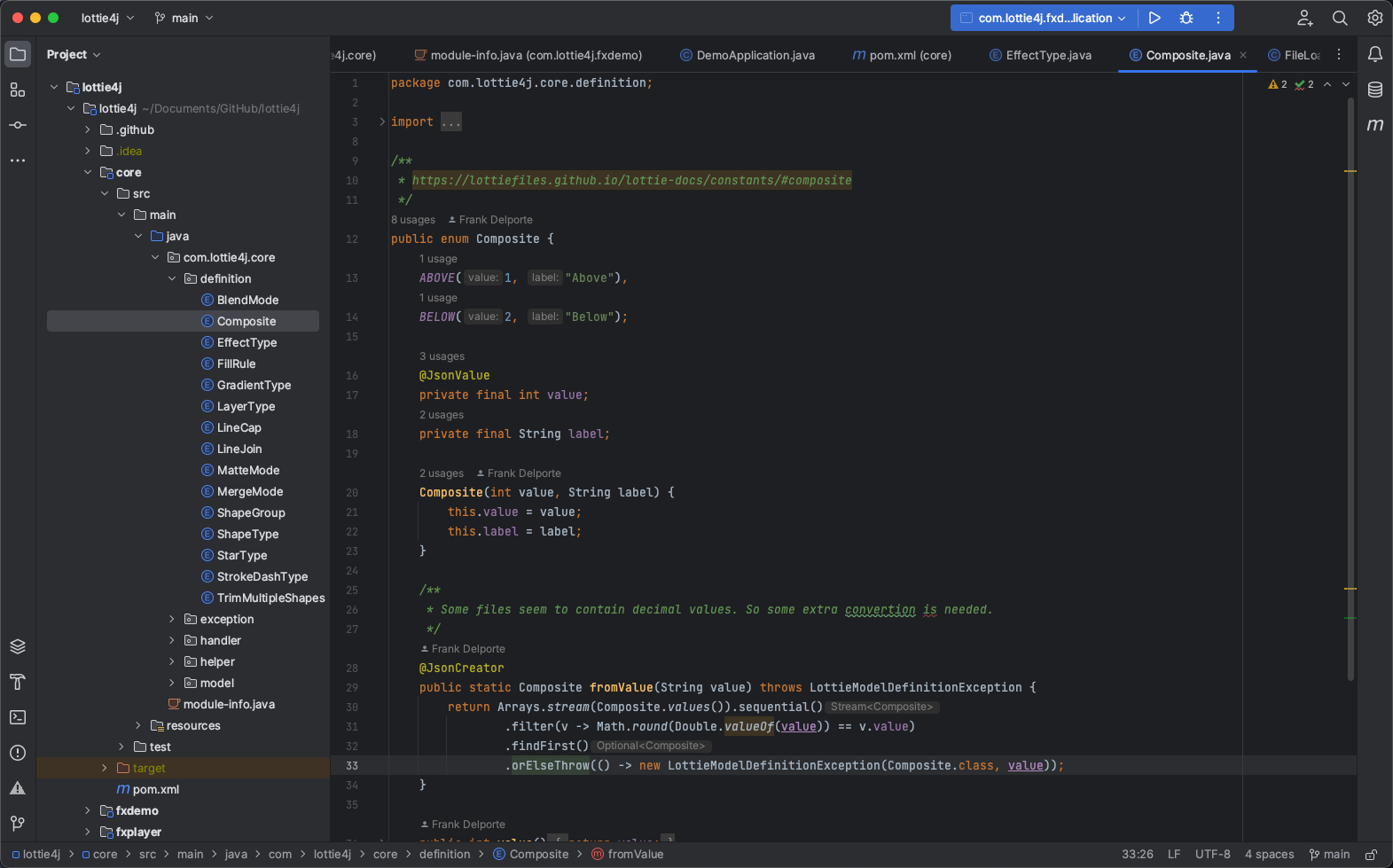
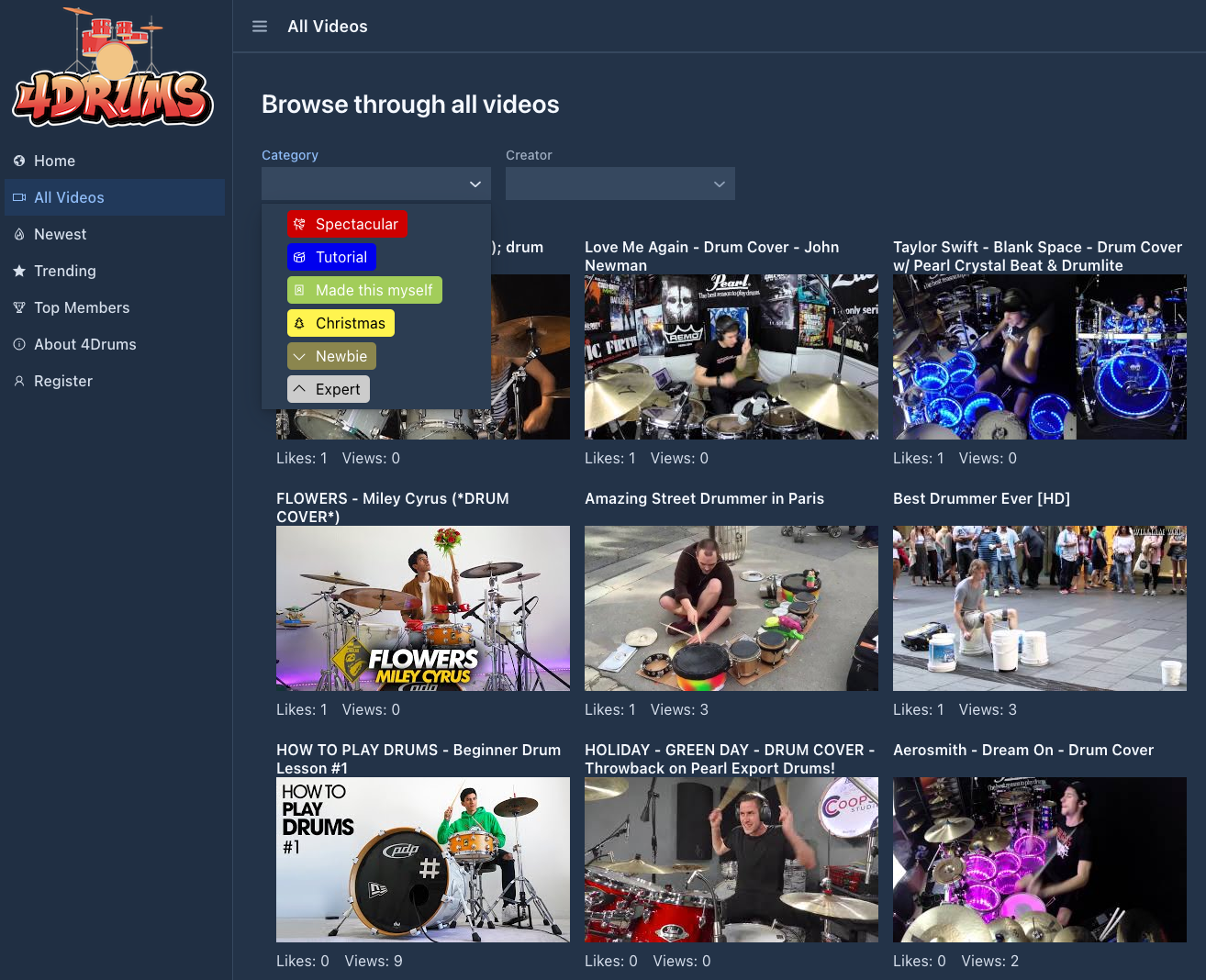
enum Level {  
 INFO,  
 WARNING,  
 ERROR  
}

**Projects Using Enums**

A few of my “pet projects” make extensive use of Java enums, and while working on these, I learned that it’s not clear to everyone that these are really powerful and can contain much more than just a list of fixed values.

Let’s take a look at a few of these use cases.



[](https://javarevisited.blogspot.com/2012/09/difference-between-enummap-and-hashmap-in-java-vs.html)[](http://www.java67.com/2012/10/java-enum-concepts-and-fundamentals.html)

**Raspberry Pi Boards in an Enum Database**

For the [Pi4J project](https://pi4j.com/), a library was needed that contains some information about the various Raspberry Pi boards, and which protocols they support on their GPIO (General Purpose Input/Output) pins. As this information should be available inside the code, on boards running without an internet connection, an API was not the right solution.

Therefore the decision was made to generate a Java library that contains all this information by using a list of enums: [github.com/Pi4J/pi4j-board-info](https://github.com/Pi4J/pi4j-board-info).

On top of this library, a website and API are created at [api.pi4j.com](https://api.pi4j.com/web/). As a result, the database can now be used in offline and online modes.

**Parsing of LottieFiles**

In another project [Lottie4J](https://lottie4j.com/) I’m trying to parse animations in JSON format into Java objects to create a JavaFX player. To make the LottieFiles-format easier to understand within this Java project, enums are used for the [various definitions as you can see in the GitHub project](https://github.com/lottie4j/lottie4j/tree/main/core/src/main/java/com/lottie4j/core/definition).

**Video Categories for**[**4drums.media**](https://4drums.media/)

This is a project inspired by my 12y son, who wants to collect and share drum videos, and hopes he can build a community around this. It’s fun to be able to use my “professional knowledge” to build such a project quickly, thanks to Spring and Vaadin, but it’s a work in progress as my son created a TODO list with ideas with 100 remaining bullet points…

Enums are used in this project to define the available video and tutorial categories which include a label, icon, text and background color.

public enum VideoCategory {  
 SPECTACULAR("la-dragon", "category.video.spectacular", "#CD0000", "#FFFFFF"),  
 TUTORIAL("la-drum", "category.video.tutorial", " #0000EE", "#FFFFFF"),  
 I\_MADE\_THIS("la-portrait", "category.video.selfmade", " #A2CD5A", "#000000"),  
 ...;  
  
 private final String icon;  
 private final String translationId;  
 private final String colorBg;  
 private final String colorText;  
  
 VideoCategory(String icon, String translationId, String colorBg, String colorText) {  
 this.icon = icon;  
 this.translationId = translationId;  
 this.colorBg = colorBg;  
 this.colorText = colorText;  
 }  
  
 public String getIcon() {  
 return icon;  
 }  
 ...  
}

**Examples of Enum Usage**

Let’s look at some examples, step-by-step. The full code is available on [GitHub as EnumExtended.java](https://github.com/foojayio/getting_started_with_java/blob/main/EnumExtended.java) and can be executed with [JBang!](https://www.jbang.dev/) with jbang EnumExtended.java.

**Basic Functionality**

A basic example of the use of an Enum is the following:

enum Level {  
 INFO,  
 WARNING,  
 ERROR  
}  
  
System.out.println("Error level: " + Level.WARNING);

Output:

Error level: WARNING

[Enums](https://www.java67.com/2013/07/15-java-enum-interview-questions-amswers-for-experienced-programmers.html)were introduced to replace the use of **int constants** as seen in other languages, which would look like this in Java:

class MyProgram {  
 public final static int LEVEL\_INFO = 1;  
 public final static int LEVEL\_WARNING = 2;  
 public final static int LEVEL\_ERROR = 3;  
 ...  
}

But an enum can do a lot more! Let’s take a look at some examples…

**Using Enums in Switch**

By using enums, switch - case code can be written in a very clean way.

public int getLevelValue(Level level) {  
 switch (level) {  
 case INFO:   
 return 1;  
 case WARNING:   
 return 2;  
 case ERROR:   
 return 3;  
 }  
 return 0;  
}

But as we will see further, this code can still be improved a lot, by extending the Level enum…

**Use Enum Instead of Boolean**

Enums are also a good use case to replace boolean checks. Let’s take an example with customers that can become “inactive” in a system. Typically you would do this with a boolean.

class Customer() {  
 String name;  
 boolean active;  
 ...  
}

But a bit later, your use-case changes and customers could also be suspended because they stopped paying the bills. Would you then add and extra boolean suspended? This could lead to a lot of changes in your program which could be avoided by using an enum for the customer state, like this:

enum CustomerState() {  
 ACTIVE,  
 INACTIVE,  
 SUSPENDED  
}  
  
class Customer() {  
 String name;  
 CustomerState state;  
 ...  
}

**Extra Data in an Enum Value**

For me, the true power of an enum, is the fact that it can actually hold a lot of information and provide methods to use these values.

In the following example, the Level enum we used before, is extended with a severity value, label description and a color code.

enum Level {  
 INFO(1, "Informative message", 0x00aa00),  
 WARNING(2, "Warning message", 0xFFA500),  
 ERROR(3, "Error message", 0xA30000);  
  
 private final int severity;  
 private final String label;  
 private final int color;  
  
 private Level(int severity, String label, int color) {  
 this.severity = severity;  
 this.label = label;  
 this.color = color;  
 }  
  
 public int getSeverity() {  
 return severity;  
 }  
  
 public String getLabel() {  
 return label;  
 }  
  
 public int getColor() {  
 return color;  
 }  
}

We can now get the data of all the Level values with the following code:

for (Level level : Level.values()) {  
 System.out.println("Level " + level.name()  
 + "\n\tSeverity: " + level.getSeverity()  
 + "\n\tLabel: " + level.getLabel()  
 + "\n\tColor: " + level.getColor());  
}

Which will produce this output:

Level INFO  
 Severity: 1  
 Label: Informative message  
 Color: 43520  
Level WARNING  
 Severity: 2  
 Label: Warning message  
 Color: 16753920  
Level ERROR  
 Severity: 3  
 Label: Error message  
 Color: 10682368

So this means level.getSeverity() fully replaces the getLevelValue(Level level) method with switch-case we've seen before.

**JSON parsing with Enums**

In many cases where you want to generate JSON output or parse JSON to Java objects, enums can be involved.

**JSON Example Data**

Let’s take for example this JSON file that contains a few log messages. The level is stored with the severity value from the Level enum.

[  
 {   
 "level": 1,  
 "timestamp": 1675867184342,  
 "message": "Program started"  
 },  
 {  
 "level": 2,  
 "timestamp": 1675867185921,  
 "message": "File X not found"  
 },  
 {  
 "level": 3,  
 "timestamp": 1675867186357,  
 "message": "Error at line Y"  
 }  
]

**Record to Load the JSON data**

We want to read these messages into Java objects, by using the Jackson library and a record:

record LogMessage(Level level, Long timestamp, String message) {  
 @JsonIgnore  
 public ZonedDateTime getZonedDateTime() {  
 return ZonedDateTime.ofInstant(Instant.ofEpochMilli(timestamp()), ZoneId.of("UTC"));  
 }  
}

The additional method getZonedDateTime will convert the Long timestamp value to be used in our code, but because of the @JsonIgnore attribute, this value will not be converted back to JSON as you will see later.

**Reading the JSON to Java objects**

By using the Jackson library, we can easily convert this JSON data to an array of LogMessage, if we add a few Jackson attributes to the enum, and by using the ObjectMapper:

enum Level {  
 ...;  
 @JsonValue  
 private final int severity;  
 ...  
 @JsonCreator  
 public static Level fromValue(Integer severity)   
 throws IllegalArgumentException {  
 return Arrays.stream(Level.values()).sequential()  
 .filter(v -> v.getSeverity() == severity)  
 .findFirst()  
 .orElseThrow(() ->   
 new IllegalArgumentException("The given severity does not exist:" + severity));  
 }  
}  
  
ObjectMapper objectMapper = new ObjectMapper();  
LogMessage[] logMessages = objectMapper  
 .readValue(json, LogMessage[].class);  
  
for (LogMessage logMessage : logMessages) {  
 System.out.println("Log message at "   
 + logMessage.getZonedDateTime()  
 + "\n\tSeverity: "   
 + logMessage.level().getLabel()  
 + "\n\tMessage: "   
 + logMessage.message());  
}

This will generate the following output. As you can see, the numeric level value from the JSON has been converted to a Level-enum-value:

Log message at 2023-02-08T14:39:44.342Z[UTC]  
 Severity: Informative message  
 Message: Program started  
Log message at 2023-02-08T14:39:45.921Z[UTC]  
 Severity: Warning message  
 Message: File X not found  
Log message at 2023-02-08T14:39:46.357Z[UTC]  
 Severity: Error message  
 Message: Error at line Y

**Converting Java Object to JSON data**

Because in the previous changes, we already added the @JsonValue attribute to the severity variable of the enum, the ObjectMapper will use the correct value to convert the level back to JSON. Let's use the PrettyPrinter to generate formatted JSON from our logMessages array:

System.out.println(objectMapper  
 .writerWithDefaultPrettyPrinter()  
 .writeValueAsString(logMessages));

This will create the following output, which contains the same content as our source JSON data.

[ {  
 "level" : 1,  
 "timestamp" : 1675867184342,  
 "message" : "Program started"  
}, {  
 "level" : 2,  
 "timestamp" : 1675867185921,  
 "message" : "File X not found"  
}, {  
 "level" : 3,  
 "timestamp" : 1675867186357,  
 "message" : "Error at line Y"  
} ]

**Converting ZonedDateTime to JSON**

As mentioned before, because of the @JsonIgnore attribute on the getZonedDateTime method, this value is not included in the generated JSON. And this even needs an extra remark... java.time.ZonedDateTime is not supported by default by Jackson. The extra module com.fasterxml.jackson.datatype:jackson-datatype-jsr310 is needed.

So a few extra changes are needed to be able to export this value, by removing @JsonIgnore, adding @JsonFormat and registering the JavaTimeModule in the objectMapper:

record LogMessage(Level level, Long timestamp, String message) {  
 @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy hh:mm:ss")  
 public ZonedDateTime getZonedDateTime() {  
 return ZonedDateTime.ofInstant(Instant.ofEpochMilli(timestamp()), ZoneId.of("UTC"));  
 }  
}  
  
objectMapper.registerModule(new JavaTimeModule());  
System.out.println(objectMapper  
 .writerWithDefaultPrettyPrinter()  
 .writeValueAsString(logMessages));

This generates JSON with the ZonedDateTime in the defined format:

[ {  
 "level" : 1,  
 "timestamp" : 1675867184342,  
 "message" : "Program started",  
 "zonedDateTime" : "08-02-2023 02:39:44"  
}, {  
 "level" : 2,  
 "timestamp" : 1675867185921,  
 "message" : "File X not found",  
 "zonedDateTime" : "08-02-2023 02:39:45"  
}, {  
 "level" : 3,  
 "timestamp" : 1675867186357,  
 "message" : "Error at line Y",  
 "zonedDateTime" : "08-02-2023 02:39:46"  
} ]

**Conclusion**

Java enums can contain much more than just a list of definitions, but also data and extended functionality!

Please experiment with the given example code file and discover how you can use these as a base to improve your code…