A Guide to Java Enums

Last modified: March 1, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/)

1. Overview

In this article we will see what Java enums are, what problems they solve and how some of the design patterns they can be used in practice.

The *enum* keyword was introduced in Java 5. It denotes a special type of class that always extends the *java.lang.Enum* class. For the official documentation on their usage have a look at the documentation (http://docs.oracle.com/javase/6/docs/api/java/lang/Enum.html).

Constants defined this way make the code more readable, allow compile time checking, document upfront the list of accepted values and avoid unexpected behavior due to invalid values being passed in.

Here's a quick and simple example of an enum that defines the status of an order for a pizza; the order status can be *ORDERED*, *READY* or *DELIVERED*:

```java
public enum PizzaStatus {
    ORDERED,
    READY,
    DELIVERED;
}
```

Additionally, they come with many useful methods, which you would otherwise have to write yourself if you were using traditional public static final constants.

1.1. An Enum with a Custom API

OK, so now that we have a basic understanding of what enums are and how you can use them, let's take our previous example to the next level by defining some extra API methods on the enum

```java
public class Pizza {
    private PizzaStatus status;
    public enum PizzaStatus {
        ORDERED,
        READY,
        DELIVERED;
    }

    public boolean isDeliverable() {
        if (getStatus() == PizzaStatus.READY) {
            return true;
        }
        return false;
    }

    // Methods that set and get the status variable.
}
```

1.2. Comparing Enum Types using "==" Operator

Since enum types ensure that only one instance of the constants exist in the JVM, we can safely use "==" operator to compare two variables as seen in the example above; moreover the "==" operator provides compile-time and run-time safety.

Let's first have a look **at run-time safety** in the following snippet where the "==" operator is used to compare statuses and a *NullPointerException* will not be thrown if either value is *null*. Conversely the an *NullPointerException* would be thrown if the equals method were used:

```java
if(testPz.getStatus().equals(Pizza.PizzaStatus.DELIVERED));
if(testPz.getStatus() == Pizza.PizzaStatus.DELIVERED);
```

As for **compile time safety**, let's have a look at another example where an enum of a different type is compared using the *equals* method is determined to be true – because the values of the enum and the *getStatus* method coincidentally are the same, but logically the comparison should be false. This issue is avoided by using the "==" operator.

The compiler will flag the comparison as an incompatibility error:

```java
if(testPz.getStatus().equals(TestColor.GREEN));
if(testPz.getStatus() == TestColor.GREEN);
```

Best Value

1.3. Using Enum Types in Switch Statements

Enum types can be used in a *switch* statements also:

```java
public int getDeliveryTimeInDays() {
    switch (status) {
        case ORDERED: return 5;
        case READY: return 2;
        case DELIVERED: return 0;
    }
    return 0;
}
```

1.4. Fields, Methods and Constructors in Enums

You can define constructors, methods and fields inside enum types that make it very powerful.

Let's extend the example above and implement the transition from one stage of a pizza to another and see how we can get rid of the *if* statement and *switch* statement used before:

```java
public class Pizza {

    private PizzaStatus status;
    public enum PizzaStatus {
        ORDERED (5){
            @Override
            public boolean isOrdered() {
                return true;
            }
        },
        READY (2){
            @Override
            public boolean isReady() {
                return true;
            }
        },
        DELIVERED (0){
            @Override
            public boolean isDelivered() {
                return true;
            }
        };

        private int timeToDelivery;

        public boolean isOrdered() {return false;}

        public boolean isReady() {return false;}

        public boolean isDelivered(){return false;}

        public int getTimeToDelivery() {
            return timeToDelivery;
        }

        PizzaStatus (int timeToDelivery) {
            this.timeToDelivery = timeToDelivery;
        }
    }

    public boolean isDeliverable() {
        return this.status.isReady();
    }

    public void printTimeToDeliver() {
        System.out.println("Time to delivery is " +
          this.getStatus().getTimeToDelivery());
    }

    // Methods that set and get the status variable.
}
```

The test snippet below demonstrate how this works:

```java
@Test
public void givenPizaOrder_whenReady_thenDeliverable() {
    Pizza testPz = new Pizza();
    testPz.setStatus(Pizza.PizzaStatus.READY);
    assertTrue(testPz.isDeliverable());
}
```

2. *EnumSet* and *EnumMap*

2.1. *EnumSet*

The *EnumSet* is a specialized *Set* implementation meant to be used with *Enum* types.

It is a very efficient and compact representation of a particular *Set* of *Enum* constants when compared to a *HashSet*, owning to the internal *Bit Vector Representation* that is used. And it provides a type safe alternative to traditional *int*-based "bit flags", allowing us to write concise code that is more readable and maintainable.

The *EnumSet* is an abstract class that has two implementations called *RegularEnumSet* and *JumboEnumSet*, one of which is chosen depending on the number of constants in the enum at the time of instantiation.

Therefore it is always a good idea to use this set when ever we want to work with a collection of enum constants in most of the scenarios (like subsetting, adding, removing, and for bulk operations like *containsAll* and *removeAll*) and use *Enum.values()* if you just want to iterate over all possible constants.

In the code snippet below, you can see how *EnumSet* is used to create a subset of constants and its usage:

```java
public class Pizza {

    private static EnumSet<PizzaStatus> undeliveredPizzaStatuses =
      EnumSet.of(PizzaStatus.ORDERED, PizzaStatus.READY);

    private PizzaStatus status;

    public enum PizzaStatus {
        ...
    }

    public boolean isDeliverable() {
        return this.status.isReady();
    }

    public void printTimeToDeliver() {
        System.out.println("Time to delivery is " +
          this.getStatus().getTimeToDelivery() + " days");
    }

    public static List<Pizza> getAllUndeliveredPizzas(List<Pizza> input) {
        return input.stream().filter(
          (s) -> undeliveredPizzaStatuses.contains(s.getStatus()))
            .collect(Collectors.toList());
    }

    public void deliver() {
        if (isDeliverable()) {
            PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy()
              .deliver(this);
            this.setStatus(PizzaStatus.DELIVERED);
        }
    }

    // Methods that set and get the status variable.
}
```

Executing the following test demonstrated the power of the *EnumSet* implementation of the *Set* interface:

```java
@Test
public void givenPizaOrders_whenRetrievingUnDeliveredPzs_thenCorrectlyRetrieved() {
    List<Pizza> pzList = new ArrayList<>();
    Pizza pz1 = new Pizza();
    pz1.setStatus(Pizza.PizzaStatus.DELIVERED);

    Pizza pz2 = new Pizza();
    pz2.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz3 = new Pizza();
    pz3.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz4 = new Pizza();
    pz4.setStatus(Pizza.PizzaStatus.READY);

    pzList.add(pz1);
    pzList.add(pz2);
    pzList.add(pz3);
    pzList.add(pz4);

    List<Pizza> undeliveredPzs = Pizza.getAllUndeliveredPizzas(pzList);
    assertTrue(undeliveredPzs.size() == 3);
}
```

2.2. EnumMap

EnumMap is a specialized *Map* implementation meant to be used with enum constants as keys. It is an efficient and compact implementation compared to its counterpart *HashMap* and is internally represented as an array:

```java
EnumMap<Pizza.PizzaStatus, Pizza> map;
```

Let's have a quick look at a real example that shows how it can be used in practice:

```java
public static EnumMap<PizzaStatus, List<Pizza>>
  groupPizzaByStatus(List<Pizza> pizzaList) {
    EnumMap<PizzaStatus, List<Pizza>> pzByStatus =
      new EnumMap<PizzaStatus, List<Pizza>>(PizzaStatus.class);

    for (Pizza pz : pizzaList) {
        PizzaStatus status = pz.getStatus();
        if (pzByStatus.containsKey(status)) {
            pzByStatus.get(status).add(pz);
        } else {
            List<Pizza> newPzList = new ArrayList<Pizza>();
            newPzList.add(pz);
            pzByStatus.put(status, newPzList);
        }
    }
    return pzByStatus;
}
```

Executing the following test demonstrated the power of the *EnumMap* implementation of the *Map* interface:

```java
@Test
public void givenPizaOrders_whenGroupByStatusCalled_thenCorrectlyGrouped() {
    List<Pizza> pzList = new ArrayList<>();
    Pizza pz1 = new Pizza();
    pz1.setStatus(Pizza.PizzaStatus.DELIVERED);

    Pizza pz2 = new Pizza();
    pz2.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz3 = new Pizza();
    pz3.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz4 = new Pizza();
    pz4.setStatus(Pizza.PizzaStatus.READY);

    pzList.add(pz1);
    pzList.add(pz2);
    pzList.add(pz3);
    pzList.add(pz4);

    EnumMap<Pizza.PizzaStatus,List<Pizza>> map = Pizza.groupPizzaByStatus(pzList);
    assertTrue(map.get(Pizza.PizzaStatus.DELIVERED).size() == 1);
    assertTrue(map.get(Pizza.PizzaStatus.ORDERED).size() == 2);
    assertTrue(map.get(Pizza.PizzaStatus.READY).size() == 1);
}
```

3. Implement Design Patterns using Enums

3.1. Singleton Pattern

Normally, implementing a class using the Singleton pattern is quite non-trivial. Enums provide an easy and quick way of implementing singletons. In addition to that, since the enum class implements the *Serializable* interface under the hood, the class is guaranteed to be a singleton by the JVM, which unlike the conventional implementation where we have to ensure that no new instances are created during deserialization.

In the code snippet below, we see how we can implement singleton pattern:

```java
public enum PizzaDeliverySystemConfiguration {
    INSTANCE;
    PizzaDeliverySystemConfiguration() {
        // Initialization configuration which involves
        // overriding defaults like delivery strategy
    }

    private PizzaDeliveryStrategy deliveryStrategy = PizzaDeliveryStrategy.NORMAL;

    public static PizzaDeliverySystemConfiguration getInstance() {
        return INSTANCE;
    }

    public PizzaDeliveryStrategy getDeliveryStrategy() {
        return deliveryStrategy;
    }
}
```

3.2. Strategy Pattern

Conventionally the Strategy pattern is written by having an interface that is implemented by different classes. Adding a new strategy meant adding a new implementation class. With enums, this is achieved with less effort, adding a new implementation means defining just another instance with some implementation.

The code snippet below shows how to implement the Strategy pattern:

```java
public enum PizzaDeliveryStrategy {
    EXPRESS {
        @Override
        public void deliver(Pizza pz) {
            System.out.println("Pizza will be delivered in express mode");
        }
    },
    NORMAL {
        @Override
        public void deliver(Pizza pz) {
            System.out.println("Pizza will be delivered in normal mode");
        }
    };

    public abstract void deliver(Pizza pz);
}
```

Add the following method to the *Pizza* class:

```java
public void deliver() {
    if (isDeliverable()) {
        PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy()
          .deliver(this);
        this.setStatus(PizzaStatus.DELIVERED);
    }
}
```

```java
@Test
public void givenPizaOrder_whenDelivered_thenPizzaGetsDeliveredAndStatusChanges() {
    Pizza pz = new Pizza();
    pz.setStatus(Pizza.PizzaStatus.READY);
    pz.deliver();
    assertTrue(pz.getStatus() == Pizza.PizzaStatus.DELIVERED);
}
```

4. Java 8 and Enums

The *Pizza* class can be rewritten in Java 8, and you can see how the methods *getAllUndeliveredPizzas()* and *groupPizzaByStatus()* become so concise with the use of lambdas and the *Stream* APIs:

```java
public static List<Pizza> getAllUndeliveredPizzas(List<Pizza> input) {
    return input.stream().filter(
      (s) -> !deliveredPizzaStatuses.contains(s.getStatus()))
        .collect(Collectors.toList());
}
```

```java
public static EnumMap<PizzaStatus, List<Pizza>>
  groupPizzaByStatus(List<Pizza> pzList) {
    EnumMap<PizzaStatus, List<Pizza>> map = pzList.stream().collect(
      Collectors.groupingBy(Pizza::getStatus,
      () -> new EnumMap<>(PizzaStatus.class), Collectors.toList()));
    return map;
}
```

5. JSON Representation of Enum

Using Jackson libraries, it is possible to have a JSON representation of enum types as if they are POJOs. The code snippet below shows the Jackson annotations that can be used for the same:

```java
@JsonFormat(shape = JsonFormat.Shape.OBJECT)
public enum PizzaStatus {
    ORDERED (5){
        @Override
        public boolean isOrdered() {
            return true;
        }
    },
    READY (2){
        @Override
        public boolean isReady() {
            return true;
        }
    },
    DELIVERED (0){
        @Override
        public boolean isDelivered() {
            return true;
        }
    };

    private int timeToDelivery;

    public boolean isOrdered() {return false;}

    public boolean isReady() {return false;}

    public boolean isDelivered(){return false;}

    @JsonProperty("timeToDelivery")
    public int getTimeToDelivery() {
        return timeToDelivery;
    }

    private PizzaStatus (int timeToDelivery) {
        this.timeToDelivery = timeToDelivery;
    }
}
```

We can use the Pizza and PizzaStatus as follows:

```java
Pizza pz = new Pizza();
pz.setStatus(Pizza.PizzaStatus.READY);
System.out.println(Pizza.getJsonString(pz));
```

to generate the following JSON representation of the *Pizza*s status:

```json
{
  "status" : {
    "timeToDelivery" : 2,
    "ready" : true,
    "ordered" : false,
    "delivered" : false
  },
  "deliverable" : true
}
```

For more information on JSON serializing/deserializing (including customization) of enum types refer to the Jackson – Serialize Enums as JSON Objects (/jackson-serialize-enums).

6. Conclusion

In this article we explored the Java enum, from the language basics to more advanced and interesting real-world use cases.

Code snippets from this article can be found in the Core Java 8 main Github (https://github.com/eugenp/tutorials/tree/master/core-java/src/main/java/com/baeldung/enums) repository and tests can be found in the Core Java 8 test Github (https://github.com/eugenp/tutorials/tree/master/core-java/src/test/java/com/baeldung/enums) repository.

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT