Java Enums

Jakob Jenkov
Last update: 2018-06-04

A *Java Enum* is a special Java type used to define collections of constants. More precisely, a Java type is a special kind of Java class. An enum can contain constants, methods etc. Java enums were added in Java 5.

This Java enum tutorial explains how to create and use a Java enum. If you prefer video, I have a version of this tutorial here:

Enum Example

Here is a simple Java enum example:

```
public enum Level {
    HIGH,
    MEDIUM,
    LOW
}
```

Notice the `enum` keyword which is used in place of `class` or `interface`. The Java `enum` keyword signals the Java compiler that this type definition is an enum.

You can refer to the constants in the enum above like this:

```
Level level = Level.HIGH;
```

Notice how the `level` variable is of the type `Level` which is the Java enum type defined in the example above. The `level` variable can take one of the `Level` enum constants as value (`HIGH`, `MEDIUM` or `LOW`). In case `level` is set to `HIGH`.

Enums in if Statements

```
Level level = ...  //assign some Level constant to it

if( level == Level.HIGH) {

} else if( level == Level.MEDIUM) {

} else if( level == Level.LOW) {

}
```

This code compares the `level` variable against each of the possible enum constants in the `Level`

If one of the enum values occur more often than the others, checking for that value in the first `if`-statement will result in better performance, as less comparison on average are executed. This is r difference though, unless the comparisons are executed a lot.

Enums in switch Statements

If your Java enum types contain a lot constants and you need to check a variable against the valu shown in the previous section, using a Java `switch` statement might be a good idea.

You can use enums in switch statements like this:

```
Level level = ...  //assign some Level constant to it

switch (level) {
    case HIGH   : ...; break;
    case MEDIUM : ...; break;
    case LOW    : ...; break;
}
```

Replace the `...` with the code to execute if the `level` variable matches the given Level constant v The code could be a simple Java operation, a method call etc.

Enum Iteration

You can obtain an array of all the possible values of a Java enum type by calling its static `values(` method. All enum types get a static `values()` method automatically by the Java compiler. Here is example of iterating all values of an enum:

```
for (Level level : Level.values()) {
    System.out.println(level);
}
```

Running this Java code would print out all the enum values. Here is the output:

```
HIGH
MEDIUM
LOW
```

Notice how the names of the constants themselves are printed out. This is one area where Java e are different than `static final` constants.

Enum toString()

An enum class automatically gets a `toString()` method in the class when compiled. The `toString` method returns a string value of the name of the given enum instance. Here is an example:

```
String levelText = Level.HIGH.toString();
```

The value of the `levelText` variable after execution of the above statement will be the text `HIGH` .

Enum Printing

If you print an enum, like this:

```
System.out.println(Level.HIGH);
```

Then the `toString()` method will get called behind the scenes, so the value that will be printed ou textual name of the enum instance. In other words, in the example above the text `HIGH` would hav printed.

Enum valueOf()

An enum class automatically gets a static `valueOf()` method in the class when compiled. The `valu` method can be used to obtain an instance of the enum class for a given String value. Here is an e

```
Level level = Level.valueOf("HIGH");
```

Enum Fields

You can add fields to a Java enum. Thus, each constant enum value gets these fields. The field v must be supplied to the constructor of the enum when defining the constants. Here is an example

```
public enum Level {
    HIGH  (3),  //calls constructor with value 3
    MEDIUM(2),  //calls constructor with value 2
    LOW   (1)   //calls constructor with value 1
    ; // semicolon needed when fields / methods follow


    private final int levelCode;

    private Level(int levelCode) {
        this.levelCode = levelCode;
    }
}
```

Notice how the Java enum in the example above has a constructor which takes an `int`. The enum constructor sets the `int` field. When the constant enum values are defined, an `int` value is passed enum constructor.

The `enum` constructor must be either `private` or package scope (default). You cannot use `public` o `protected` constructors for a Java `enum`.

Enum Methods

You can add methods to a Java enum too. Here is an example:

```
public enum Level {
    HIGH  (3),  //calls constructor with value 3
    MEDIUM(2),  //calls constructor with value 2
    LOW   (1)   //calls constructor with value 1
    ; // semicolon needed when fields / methods follow


    private final int levelCode;

    Level(int levelCode) {
        this.levelCode = levelCode;
    }

    public int getLevelCode() {
        return this.levelCode;
    }

}
```

You call a Java enum method via a reference to one of the constant values. Here is Java enum m call example:

```
Level level = Level.HIGH;

System.out.println(level.getLevelCode());
```

This code would print out the value `3` which is the value of the `levelCode` field for the enum consta

You are not restricted to simple getter and setter methods. You can also create methods that mak calculations based on the field values of the enum constant. If your fields are not declared `final` y even modify the values of the fields (although that may not be so good an idea, considering that tl enums are supposed to be constants).

Enum Abstract Methods

It is possible for a Java enum class to have abstract methods too. If an enum class has an abstra method, then each instance of the enum class must implement it. Here is a Java enum abstract m example:

```
public enum Level {
    HIGH{
        @Override
        public String asLowerCase() {
            return HIGH.toString().toLowerCase();
        }
    },
    MEDIUM{
        @Override
        public String asLowerCase() {
            return MEDIUM.toString().toLowerCase();
        }
    },
    LOW{
        @Override
        public String asLowerCase() {
            return LOW.toString().toLowerCase();
```

```
}
```

Notice the abstract method declaration at the bottom of the enum class. Notice also how each enu
instance (each constant) defines its own implementation of this abstract method. Using an abstrac
method is useful when you need a different implementation of a method for each instance of a Jav

EnumSet

Java contains a special **Java Set** implementation called `EnumSet` which can hold enums more effic
than the standard Java Set implementations. Here is how you create an instance of an `EnumSet` :

```
EnumSet<Level> enumSet = EnumSet.of(Level.HIGH, Level.MEDIUM);
```

Once created, you can use the `EnumSet` just like any other Set.

EnumMap

Java also contains a special **Java Map** implementation which can use Java enum instances as ke
is a Java `EnumMap` example:

```
EnumMap<Level, String> enumMap = new EnumMap<Level, String>(Level.class);
enumMap.put(Level.HIGH  , "High level");
enumMap.put(Level.MEDIUM, "Medium level");
enumMap.put(Level.LOW   , "Low level");

String levelValue = enumMap.get(Level.HIGH);
```

Enum Miscellaneous Details

Java enums extend the `java.lang.Enum` class implicitly, so your enum types cannot extend anothe

If a Java enum contains fields and methods, the definition of fields and methods must always com
the list of constants in the enum. Additionally, the list of enum constants must be terminated by a
semicolon;

Next: Java Annotations

Jakob Jenkov