دیتا تایپ boolean به منظور تعریف شرایطی مورد استفاده قرار می‌گیرد که در آن بیش از دو حالت نداشته باشیم و مسلماً اگر در شرایطی قرار بگیریم که بیش از دو حالت پیش روی ما باشد، به طور قطع از این دیتا تایپ نخواهیم توانست استفاده کنیم که طراحان زبان برنامه‌نویسی جاوا برای چنین شرایطی enum را در این زبان شیئ‌گرا گنجانده‌اند.

در واقع، کیورد enum از واژۀ Enumeration به معنای «شمارش» گرفته شده است که در زبان برنامه‌نویسی جاوا به منظور تعریف شرایطی با **بیش از دو حالت** مورد استفاده قرار می‌گیرد که به عنوان چند مثال از دنیای واقعی می‌توان چراغ‌های راهنمایی را نام برد که دارای سه حالت سبز، قرمز و زرد هستند یا روزهای هفته که شنبه، یکشنبه، دوشنبه، سه شنبه، چهارشنبه، پنج شنبه و جمعه را شامل می‌شوند. به علاوه،‌ یک کشور می‌تواند توسعه یافته، در حال توسعه و یا عقب افتاده باشد!

به طور کلی، می‌توان گفت که ساختار enum یک نوع کلاس در زبان جاوا می‌باشد که از چند مقدار کانستنت تشکیل شده است و مثال‌های فوق‌الذکر مواردی از دنیای واقعی هستند که اگر بخواهیم برنامه‌ای در رابطه با آن‌ها بنویسیم، ناگزیر می‌باید از ساختار enum استفاده کنیم.

--------------------------------------------------------------------------------------------

enum یک struct نیست بلکه همون enumeration تو اکثر زبان های برنامه نویسی هستش. enum ها مثل مجموعه ها می مونن و برای تعریف یک سری ثابت به هم وابسته هستن.  
مثل مجموعه اسامی گل ها (شامل 10 گل) که ثابت هستن و به هم ربط دارن رو میشه یک enum براشون تعریف کرد. یا مثلن مجموعه حالاتی که یک پردازش به خودش میگیره.  
در کل توی جاوا چون مبنا بر این بوده که همه چی شی گرا باشه و جز کلاس هیچ ساختار دیگه ای نباشه، enum رو شبیه کلاس ها پیاده سازی کردن.  
قبل از اومدن enum تو جاوا تو نسخه 5 از اینترفیس ها برای این منظور استفاده میشد. چون لازم نیست شما شی تعریف کنی از یک سری مقدار ثابت مشخص!

Java enumis a kind of a compiler magic. In byte code, any enum is represented as a class that extends the abstract class java.lang.Enum and has several static members. Therefore, enum cannot extend any other class orenum: there is no multiple inheritance.

Class cannot extend enum, as well. This limitation is enforced by the compiler.

Here is a simple enum:

1

enum Color {red, green, blue}

This class tries to extend it:

1

class SubColor extends Color {}

This is the result of an attempt to compile class SubColor:

1

$ javac SubColor.java

2

SubColor.java:1: error: cannot inherit from final Color

3

class SubColor extends Color {}

4

^

5

SubColor.java:1: error: enum types are not extensible

6

class SubColor extends Color {}

7

^

8

2 errors

Enum cannot either extend or be extended. So, how is it possible to extend its functionality? The key word is "functionality." Enumcan implement methods. For example, enumColor may declare abstract method draw() and each member can override it:

1

enum Color {

2

red { @Override public void draw() { } },

3

green { @Override public void draw() { } },

4

blue { @Override public void draw() { } },

5

;

6

public abstract void draw();

7

}

Popular usage of this technique is explained [here](https://dzone.com/articles/enum-tricks-featured-enum-instead-of-switch). Unfortunately, it is not always possible to implement method in enumitself because:

1. theenummay belong to a third-party library or another team in the company
2. theenumis probably overloaded with other data and functions, so it becomes unreadable
3. the enumbelongs to a module that does not have dependencies required for implementation of method  draw().

This article suggests the following solutions for this problem.

**Mirror Enum**

We cannot modify enumColor? No problem! Let's create enumDrawableColor that has exactly the same elements as Color. This new enum will implement our method draw():

1

enum DrawableColor {

2

red { @Override public void draw() { } },

3

green { @Override public void draw() { } },

4

blue { @Override public void draw() { } },

5

;

6

public abstract void draw();

7

}

This enum is a kind of reflection of source enum Color, i.e.Color is its mirror.

But how doe we use the newenum? All our code uses Color, not DrawableColor. The simplest way to implement this transition is using built-in enum methods name() and valueOf() as following:

1

Color color = ...

2

DrawableColor.valueOf(color.name()).draw();

Since name() method is final and cannot be overridden, and valueOf()is generated by a compiler. These methods are always a good fit for each other, so no functional problems are expected here. Performance of such transition is good also: methodname()does not create a new String but returns a pre-initialized one (see source code of java.lang.Enum). Method valueOf()is implemented using Map, so its complexity is O(1).

The code above contains obvious problem. If source enumColor is changed, the secondary enumDrawableColordoes not know this fact, so the trick with name()and valueOf() will fail at runtime. We do not want this to happen. But how to prevent possible failure? We have to letDrawableColor know that its mirror is Color and enforce this preferably at compile time or at least at unit test phase. Here, we suggest validation during unit tests execution. Enum can implement a static initializer that is executed when enum is mentioned in any code. This actually means that if static initializer validates that enumDrawableColor fits Color, it is enough to implement a test like the following to be sure that the code will be never broken in production environment:

1

@Test

2

public void drawableColorFitsMirror {

3

DrawableColor.values();

4

}

Static initializer just has to compare elements ofDrawableColorand Color and throw an exception if they do not match. This code is simple and can be written for each particular case. Fortunately, a simple open-source library named [enumus](https://github.com/alexradzin/enumus) already implements this functionality, so the task becomes trivial:

1

enum DrawableColor {

2

....

3

static {

4

Mirror.of(Color.class);

5

}

6

}

That's it. The test will fail if source enum and DrawableColor do not fit it any more. Utility class Mirror has other methods that gets two arguments: classes of two enums that have to fit. This version can be called from any place in code and not only from enum that has to be validated.

**EnumMap**

Do we really have to define another enum that just holds implementation of one method? In fact, we do not have to. Here is an alternative solution. Let's define interface Draweras following:

1

public interface Drawer {

2

void draw();

3

}

The next examples assume that all elements of enum Color are statically imported.

Now, let's create mapping between enum elements and implementation of interfaceDrawer:

1

Map<Color, Drawer> drawers = new EnumMap<>(Color.class) {{

2

put(red, new Drawer() { @Override public void draw(){}});

3

put(green, new Drawer() { @Override public void draw(){}})

4

put(blue, new Drawer() { @Override public void draw(){}})

5

}}

The usage is simple:

1

drawers.get(color).draw();

EnumMap is chosen here as a Map implementation for better performance. Map guaranties that each enum element appears there only once. However, it does not guarantee that there is entry for each enum element. But it is enough to check that size of the map is equal to number ofenum elements:

1

drawers.size() == Color.values().length

[Enumus](https://github.com/alexradzin/enumus) suggests convenient utility for this case also. The following code throws IllegalStateException with descriptive message if map does not fit Color:

1

EnumMapValidator.validateValues(Color.class, map, "Colors map");

It is important to call the validator from the code which is executed by unit test. In this case the map based solution is safe for future modifications of source enum.

**EnumMap and Java 8 Functional Interface**

In fact, we do not have to define special interface to extend enum functionality. We can use one of functional interfaces provided by JDK starting from version 8 (Function, BiFunction, Consumer, BiConsumer, Supplier, etc.) The choice depends on parameters that have to be sent to the function. For example, Supplier can be used instead of Drawable defined in the previous example:

1

Map<Color, Supplier<Void>> drawers = new EnumMap<Color, Supplier<Void>>(Color.class) {{

2

put(red, new Supplier<Void>() { @Override public Void get() {return null;}});

3

put(green, new Supplier<Void>() { @Override public Void get() {return null;}});

4

put(blue, new Supplier<Void>() { @Override public Void get() {return null;}});

5

}}

The previous code snippet can be simplified:

1

Map<Color, Supplier<Void>> drawers = new EnumMap<Color, Supplier<Void>>(Color.class) {{

2

put(red, () -> null);

3

put(green, () -> null);

4

put(blue, () -> null);

5

}};

6

​

Usage of this map is pretty similar to one from the previous example:

1

drawers.get(color).get();

This map can be validated exactly as the map that stores instances of Drawable.

enum Seasons

{

SUMMER, WINTER, AUTUMN, SPRING

}

class Main

{

public static void main(String[] args)

{

System.out.println(Seasons.SUMMER);

System.out.println(Seasons.WINTER);

System.out.println(Seasons.AUTUMN);

System.out.println(Seasons.SPRING);

}

}

Output:

SUMMER

WINTER

AUTUMN

SPRING

The above code displays the functionality of an enum data type. The use of enums can make any code more explicit and less error-prone.

Enums are widely used in menu-driven programs or when we know all possible values at compile time.

## [Java enum Inheritance](https://www.delftstack.com/howto/java/enum-inheritance-java/#java-enum-inheritance)

enum Seasons

{

SUMMER, WINTER, AUTUMN, SPRING

}

class Main

{

class Weather extends Seasons

{

public static void main(String[] args)

{

// statements

}

}

}

Output:

Main.java:8: error: cannot inherit from final Seasons

class Weather extends Seasons {

^

Main.java:8: error: enum types are not extensible

class Weather extends Seasons {

^

The above code produces an error because an enum class cannot be used to derive another functional class.

## [Use an enum to Implement an Interface](https://www.delftstack.com/howto/java/enum-inheritance-java/#use-an-enum-to-implement-an-interface)

interface Weather {

public void display();

}

enum Seasons implements Weather

{

SUMMER, WINTER, AUTUMN, SPRING;

public void display()

{

System.out.println("The season is " + this);

}

}

class Main

{

public static void main(String[] args)

{

Seasons.SUMMER.display();

}

}

Output:

The season is SUMMER

In the above code, we are using an enum class, Seasons, to implement the Weather interface. Since we can use an enum class to implement an interface, we have written the abstract method display() inside the enum class.

**Conclusion**

This article shows how powerful Java enumscan be if we put some logic inside. It also demonstrates two ways to expand the functionality ofenumsthat work despite the language limitations. The article introduces to user the open-source library named [Enumus](https://github.com/alexradzin/enumus) that provides several useful utilities that help to operate enums easier.

منابع:

<https://www.baeldung.com/java-extending-enums>

sokan academy.com

https://dzone.com/articles/enum-tricks-two-ways-to-extend-enum-functionality