

An enum, also referred to as an enumeration or enumerated data type, allows you to define a distinct type comprising an organized list of constants in a specified order. Enums are introduced in Java 5.



Let's take a scenario and try to build with and without enums

---

Let's imagine you are trying to build a program that gives an estimated completion time based on the priority of a task. Below is the input and output expected from the program,

- 1) LOW -> 7
- 2) MEDIUM -> 5
- 3) HIGH -> 2
- 4) URGENT -> 1

# With out enums

A typical way to represent the four types of priorities before Java 5 was to declare four final constants in a class, say Priority,


```
public class Priority {  
  
    public static final int LOW = 0;  
    public static final int MEDIUM = 1;  
    public static final int HIGH = 2;  
    public static final int URGENT = 3;  
  
}
```

Next we can write a utility class named TaskUtil that has a method to determine the estimated completion time based on the priority of a task. The code for the TaskUtil class may look like in the image,


```
public class TaskUtil {  
  
    public static int getEstimatedCompletionTime(int priority) {  
        return switch (priority) {  
            case Priority.LOW -> 7;  
            case Priority.MEDIUM -> 5;  
            case Priority.HIGH -> 3;  
            case Priority.URGENT -> 1;  
            default -> throw new IllegalStateException("Unexpected value: " + priority);  
        };  
    }  
  
}
```


# Problems with constants approach

- ① As priority is represented as an integer constant, any integer value can be passed to the `'getEstimatedCompletionTime()'` method, not just 0, 1, 2, and 3 which are the valid values for the priority type. To address this, it may be necessary to include a check within the method to ensure that only valid priority values can be passed, preventing potential exceptions. However, this approach doesn't offer a permanent solution, as any addition of new priority types would require updating the code that validates these values.
- ② If the value for a priority constant is changed, the code using it must be recompiled to reflect the modifications. For instance, when compiling the `'TaskUtil'` class, `'Priority.LOW'` is replaced with 0, `'Priority.MEDIUM'` with 1, and so forth. If the value for the constant `'LOW'` in the `'Priority'` class is changed to 7, recompiling the `'TaskUtil'` class is essential to ensure it acknowledges the updated value; otherwise, the `'TaskUtil'` class will continue using the old value, 0.
- ③ When saving the value of the priority on disk, its corresponding integer value, such as 0, 1, 2, etc., will be saved, not the string values `'LOW'`, `'MEDIUM'`, `'HIGH'`, etc. A separate map must be maintained to facilitate the conversion from an integer value to its corresponding string representation for all priority types.
- ④ When printing the priority value of a task, it will print an integer, such as 0, 1, 2, etc. An integer value for a priority may not convey meaningful information to end users.
- ⑤ Priority types of tasks have a specific order. For example, a LOW priority task is given less importance than a MEDIUM priority task. Since priority is being represented by arbitrary numbers, code must be written using hard-coded values to maintain the order of the constants defined in the `'Priority'` class. If another priority type, such as `'VERY_HIGH'`, is added, which has less priority than `'URGENT'` and more priority than `'HIGH'`, the code handling the ordering of priority types must be modified.
- ⑥ There is no automatic way (except by hard coding) that allows listing all priority types. The developer needs to manually maintain and update the code to reflect any changes or additions to priority types.


 As per the online dictionary, the term "**enumerate**" is defined as "**to specify one after another**". The enum type precisely facilitates this, allowing you to specify constants in a particular order. The constants within an enum type are instances of that specific enum type. To define an enum type, the keyword used is "**enum**", and its most basic syntax is as follows:

```
[access-modifier] enum <enum-type-name> {  
    // List of comma separated names of enum constants  
}
```

 The access modifier for an enum aligns with that of a class, which can be public, private, protected, or at the package level. The enum type name is a valid Java identifier, and the body of the enum type is enclosed within braces immediately following its name. Within the enum type body, you can have a series of comma-separated constants.

 Below declares a public enum type called **PriorityEnum** with four enum constants: LOW, MEDIUM, HIGH, and URGENT

```
public enum PriorityEnum {  
    LOW, MEDIUM, HIGH, URGENT;  
}
```

 A customary practice is to designate enum constants in uppercase. The inclusion of a semicolon after the final enum constant is discretionary, particularly when there is no code following the enumeration of constants.

# Using Enum Types in switch Statements

```
public enum PriorityEnum {  
    LOW, MEDIUM, HIGH, URGENT  
}
```

Enum types can be employed in switch statements. When the switch expression is of an enum type, all case labels can consist of unqualified enum constants belonging to the same enum type. The switch statement automatically deduces the enum type name from the expression's type. Including a default label is also an option.

Using enum inside switch statement

```
public class TaskUtil {  
    public static int getEstimatedCompletionTime(PriorityEnum priority) {  
        int days = 0;  
        switch (priority) {  
            case LOW:  
                days = 7;  
                break;  
            case MEDIUM:  
                days = 5;  
                break;  
            case HIGH:  
                days = 2;  
                break;  
            case URGENT:  
                days = 1;  
                break;  
        }  
        return days;  
    }  
}
```

Using enum inside switch expression

```
public class TaskUtil {  
    public static int getEstimatedCompletionTime(PriorityEnum priority) {  
        return switch (priority) {  
            case LOW -> 7;  
            case MEDIUM -> 5;  
            case HIGH -> 2;  
            case URGENT -> 1;  
        };  
    }  
}
```



Other examples of enum and their constants,

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY  
}
```

```
public enum Size {  
    SMALL, MEDIUM, LARGE, EXTRA_LARGE  
}
```



We can declare a variable of an enum type the same way you declare a variable of a class type. We can assign null as well to an enum-type variable,

```
PriorityEnum priority;  
PriorityEnum priorityEnum = null;
```



We can't instantiate an enum type. Below code compilation fails,

```
PriorityEnum priority = new PriorityEnum();
```



The Java enum constants are static and final implicitly. Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum. Typically, the enum body predominantly comprises constants. You may have a question, if we can't create object of enum type, then what is the purpose of constructors, instance methods etc.

Constructors for an enum type become necessary due to the way enum instances are instantiated. The compiler generates code to create instances of the enum type, where all enum constants are treated as objects of the same type. The instances, named identical to the enum constants, are generated by the compiler, employing certain mechanisms to achieve this.





An enum type serves to specify two crucial aspects:

- 1) Enum constants, representing the exclusive and permissible values for that particular type.
- 2) The sequence or order assigned to those constants.



In the case of the `PriorityEnum` enum type, it defines four enum constants. Consequently, a variable of the `PriorityEnum` enum type can exclusively hold one of the four values—`LOW`, `MEDIUM`, `HIGH`, or `URGENT`—or a null value. Enum constants can be referenced using dot notation, with the enum type name serving as the qualifier. The following code snippet exemplifies the assignment of values to a variable of the `PriorityEnum` enum type:

```
PriorityEnum low = PriorityEnum.LOW;  
PriorityEnum medium = PriorityEnum.MEDIUM;  
PriorityEnum high = PriorityEnum.HIGH;  
PriorityEnum urgent = PriorityEnum.URGENT;
```



Every enum constant is associated with a name, and this name corresponds to the identifier provided for the constant in its declaration. For instance, in the `PriorityEnum` enum type, the name of the `LOW` constant is "LOW".



An enum type provides each of its constants with an ordinal, starting from zero and incrementing by one for each subsequent constant in the list. In the `PriorityEnum` enum type, the ordinal values are assigned as follows: 0 to `'LOW'`, 1 to `'MEDIUM'`, 2 to `'HIGH'`, and 3 to `'URGENT'`. Modifying the order or adding new constants in the enum type will result in corresponding changes to their ordinal values.



The '**name()**' and '**ordinal()**' methods enable the retrieval of the name and ordinal of an enum constant, respectively. Additionally, every enum type includes a static method called '**values()**' which returns an array containing its constants in the order they are declared within the type. The below code snippet exemplifies printing the name and ordinal of all enum constants declared in the 'PriorityEnum' enum type.

```
for (PriorityEnum priority : PriorityEnum.values()) {  
    String name = priority.name();  
    int ordinal = priority.ordinal();  
    System.out.println(name + "(" + ordinal + ")");  
}
```

output →  
LOW (0)  
MEDIUM (1)  
HIGH (2)  
URGENT (3)



Obtaining the reference of an enum constant based on its name or ordinal in the list is referred to as reverse lookup. To achieve this, you can utilize the **valueOf()** method, automatically added by the compiler to an enum type. The **valueOf()** method enables reverse lookup based on the name or ordinal of an enum constant.

```
PriorityEnum low = PriorityEnum.valueOf("LOW"); // A reverse lookup using name  
PriorityEnum high = PriorityEnum.values()[2]; // A reverse lookup using Ordinal
```





It is possible to include a **nested enum type** declaration within a class, interface, or another enum type. Nested enum types are inherently static, and you can also explicitly declare them as static in their declaration. Due to the static nature of enum types, whether declared explicitly or not, local enum types cannot be declared.

```
public class Car {  
    public enum Model {  
        SEDAN, SUV, HATCHBACK  
    }  
}
```

# Associating Data to Enum Constants

Typically, you declare an enum type primarily to define a set of enum constants, as demonstrated in the example of the `PriorityEnum` enum type. As an enum type is essentially a class type, you have the flexibility to declare almost anything within the body of an enum type that you would declare within a class body. Let's associate a data element, such as estimated completion days, with each of the enum constants in the `PriorityEnum` enum type.

This program prints the names, ordinals, and estimated completion days of the constants. Notably, the logic to compute estimated completion days is encapsulated within the enum type, combining the functionality of the `PriorityEnum` enum type and the `getEstimatedCompletionTime()` method in the `TaskUtil` class. This eliminates the need for a switch statement, as each enum constant independently holds information about its estimated completion days.

```
public enum AdvancedPriorityEnum {  
    LOW(7), MEDIUM(5), HIGH(2), URGENT(1);  
    private int estimatedCompletionDays;  
    private AdvancedPriorityEnum(int estimatedCompletionDays) {  
        this.estimatedCompletionDays = estimatedCompletionDays;  
    }  
    public int getEstimatedCompletionDays() {  
        return estimatedCompletionDays;  
    }  
}
```

```
public class AdvancedPriorityEnumTest {  
    public static void main(String[] args) {  
        for (AdvancedPriorityEnum p : AdvancedPriorityEnum.values()) {  
            String name = p.name();  
            int ordinal = p.ordinal();  
            int days = p.getEstimatedCompletionDays();  
            System.out.println("name=" + name + ", ordinal=" + ordinal  
                               + ", days=" + days);  
        }  
    }  
}
```

EnumSet is a specialized implementation of the Set interface in Java that is designed to work exclusively with enum types. It is part of the java.util package and provides a high-performance alternative to other general-purpose set implementations when dealing with enum constants. EnumSet is particularly efficient in terms of memory usage and performance.

Here are three key methods related to EnumSet: of(), allOf() and range().

## of() method:

The of() method is a static factory method used to create an EnumSet containing specified enum constants. It allows you to create an EnumSet with a variable number of enum constants passed as arguments. Here's an example:

```
// Creating an EnumSet with MONDAY, TUESDAY, and WEDNESDAY
EnumSet<Day> someDays = EnumSet.of(Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY);
```

## allOf() method:

The allOf() method is a static factory method used to create an EnumSet containing all of the elements in the specified enum type.. Here's an example:

```
// Creating an EnumSet with all days inside the Day Enum type
EnumSet<Day> allDays = EnumSet.allOf(Day.class);
```

## range() method:

The range() method is another static factory method that creates an EnumSet consisting of all the elements in the specified range. This is particularly useful when dealing with enum types that have a natural order, such as numeric or alphabetical order. Below is an example,

```
public enum Grades { A, B, C, D, E }  
  
// Creating an EnumSet with all grades from A to C (inclusive)  
EnumSet<Grades> passGrades = EnumSet.range(Grades.A, Grades.C);
```

## complementOf() method:

The complementOf() method in the EnumSet class is another static factory method that returns an EnumSet consisting of all the elements in the specified universe that are not present in the specified set. Below is an example,

```
// Creating the complement of primaryColors with respect to the universe of Colors  
EnumSet<Grades> failGrades = EnumSet.complementOf(passingGrades);
```



1

## Type safety

Enums provide type safety by allowing you to define a fixed set of named values. This means that you can avoid errors caused by passing invalid values to methods or using the wrong data type.

2

## Readability

Enums make code more readable by providing a clear and concise way to represent a fixed set of related constants. This makes code easier to understand and maintain.

3

## Robustness

Enums make code more robust by providing a centralized and consistent way to handle a set of related constants. This makes it easier to modify code in the future without introducing bugs.

4

## Code organization

Enums can help you organize your code by grouping related constants together. This makes it easier to find and modify code that uses those constants.

5

## Switch statements

Enums can be used in switch statements, making it easy to handle a set of related cases. This can improve the readability and maintainability of your code.

# Enums in Java

```
public enum Day {  
  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY  
  
}
```

This is a Java program that uses an enum called Day to print out whether a given day is a weekday or a weekend. Here is the sample output of the program,

```
Weekday  
Weekday  
Weekday  
Weekend  
Weekend
```

Output

```
public class EnumTest {  
  
    Day day;  
  
    public EnumTest(Day day) {  
        this.day = day;  
    }  
  
    public void print() {  
        switch (day) {  
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> System.out.println("Weekday");  
            case SATURDAY, SUNDAY -> System.out.println("Weekend");  
        }  
    }  
  
    public static void main(String[] args) {  
        EnumTest firstDay = new EnumTest(Day.MONDAY);  
        firstDay.print();  
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);  
        thirdDay.print();  
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);  
        fifthDay.print();  
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);  
        sixthDay.print();  
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);  
        seventhDay.print();  
    }  
}
```