Enum

A type-safe way to define a fixed set of constants in Java.

Beginner 35 min

Last updated: November 1, 2026
Read online: https://www.learnforgex.com/en/topics/enum

Enum

In Java, an **enum** (short for enumeration) is a special reference type that allows you to define a fixed set of constants in a type-safe manner. It was added to the language in Java 5 and has since become the standard solution for cases where "this variable can only take one of these few values."

What is an Enum?

If the values a variable can take are known in advance and belong to a limited set (for example, days of the week, order statuses, suit of a card), you should model this with an enum instead of string or int constants. In Java, every enum is a class that implicitly extends the `java.lang.Enum` class — meaning enum constants are actually **objects**, but their number is fixed:

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

In this single line, Java creates seven `Day` objects (`MONDAY`, `TUESDAY`, ...) behind the scenes and stores them as `static final` fields. This is why it is completely safe to compare enum constants using `==` — `equals()` is not needed.

Enum vs String

Using an enum provides compile-time type safety compared to string constants. Compare these two approaches:

```
// With String – the compiler doesn't protect you
void setStatus(String status) { ... }
setStatus("APPROVEDD"); // typo, but the code compiles and the problem arises at runtime

// With Enum – the compiler protects you
void setStatus(OrderStatus status) { ... }
setStatus(OrderStatus.APPROVEDD); // compile-time error – no such constant exists
```

With a string, a typo is only noticed at runtime (perhaps even in production); with an enum, the same error is caught at compile time. Additionally, IDEs can offer auto-completion and a list of "all possible values" when using enums — this is not possible with strings.

Basic Enum Usage

In its simplest form, an enum is defined like this:

```
enum Gender {
    MALE,
    FEMALE
}
```

> Enum constants are conventionally written in uppercase (`MALE` , `FEMALE`) — just like `static final` constants, because they actually are.

Constructor

Since enum constants are objects, they can also have constructors — each constant can pass its own parameters during creation:

```
enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6);

    private final double massKg;
    private final double radiusM;

    // The constructor must always be private (or package-private) — enum
    // constants are only ever "constructed" on the definition line above,
    // by the enum itself.
    Planet(double massKg, double radiusM) {
        this.massKg = massKg;
        this.radiusM = radiusM;
    }

    public double getMassKg() {
        return massKg;
    }

    public double getRadiusM() {
        return radiusM;
    }

    public double surfaceGravity() {
        final double gravitationalConstant = 6.67300E-11;
        return gravitationalConstant * massKg / (radiusM * radiusM);
    }
}
```

Here, each planet constant (`MERCURY`, `VENUS`, `EARTH`) is "constructed" with its own mass and radius values on the line where the enum is defined. This constructor call runs once when the class is first loaded (at class loading time) — there is a single instance of each object, just like a Singleton.

> Enum constructors can never be `public` or `protected` — Java enforces this at compile time (only `private` or package-private/default access is allowed). This is because enum constants can only be created by the enum itself on the definition line; it is not possible to call `new Planet(...)` from the outside.

Fields

Values passed to the constructor are usually stored in `private final` fields — this means each enum constant has its own unique, immutable data. In the `Planet` example above, `massKg` and `radiusM` were defined exactly this way; they were exposed to the outside only through getter methods:

```
class PlanetUsageExample {
    public static void main(String[] args) {
        Planet earth = Planet.EARTH;

        System.out.println("Mass: " + earth.getMassKg() + " kg");
        System.out.printf("Surface gravity: %.2f m/s^2%n", earth.surfaceGravity());
    }
}
```

> Making enum fields `final` is not a requirement but a strong practice: it is logically meaningless for a constant (e.g., `EARTH`) to "change" to a different mass at runtime — `final` guarantees this to the compiler.

Methods

Enums can contain instance methods just like ordinary classes. `surfaceGravity()` in the `Planet` example was already an illustration of this. We can also add a method that makes decisions based on the constant values without needing a constructor:

```
enum DayWithMethod {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
    }
}
```

```
for (DayWithMethod day : DayWithMethod.values()) {
    System.out.println(day + " is it the weekend? " + day.isWeekend());
}
```

values()

`values()` is a static method that the compiler automatically generates for every enum; it returns all constants as an array in the order they were defined — we already used it in the loop above:

```
DayWithMethod[] days = DayWithMethod.values();
System.out.println(days.length); // 7
```

> `values()` copies a new array every time it is called — calling it repeatedly inside a loop (e.g., `for (int i = 0; i < DayWithMethod.values().length; i++)`) leads to unnecessary performance loss. It is better to get the array once and store it in a variable.

valueOf()

`valueOf(String)` returns the constant that exactly matches the given name — if no match is found, it throws an `IllegalArgumentException`:

```
DayWithMethod day = DayWithMethod.valueOf("MONDAY"); // MONDAY
DayWithMethod error = DayWithMethod.valueOf("monday"); // IllegalArgumentException! Case-sensitive
```

Always validate free text coming from the user before passing it to `valueOf()` or wrap the call in a `try/catch`.

name()

`name()` returns the name of the constant **exactly** as it is written in the source code — even if `toString()` is overridden, `name()` always gives the original name:

```
System.out.println(DayWithMethod.MONDAY.name()); // "MONDAY"
```

> For text to be displayed to the user, use a separate `displayName` field (passed through the constructor) or a `toString()` override instead of `name()` — `name()` is fixed and cannot be localized (it's not i18n friendly).

ordinal()

`ordinal()` returns the position (starting from 0) of the constant in the order it was defined:

```
System.out.println(DayWithMethod.MONDAY.ordinal()); // 0
System.out.println(DayWithMethod.SUNDAY.ordinal());  // 6
```

It is worth repeating the warning we gave at the very beginning of this section here: never store the `ordinal()` value as persistent data (database, file, API contract) — adding a new constant to the enum definition or changing the order silently breaks the meaning of existing data.

Usage with switch

Enums naturally work very well with `switch` statements. With the modern switch syntax introduced in Java 21:

```
String message = switch (DayWithMethod.SATURDAY) {
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Work day";
    case SATURDAY, SUNDAY -> "Weekend";
};
```

> In `case` labels, we write the name of the enum constant alone (`MONDAY`), not `DayWithMethod.MONDAY` — Java automatically completes this by inferring from the switched type. Also, in a switch expression that covers all constants (like the one above), a `default` branch is not needed; the compiler checks that all possible constants are covered and gives an error if any are missing — a great way to catch forgotten switch blocks at compile time when a new constant is added.

Interface Implementation

An enum cannot extend another class (it already implicitly extends `Enum`), but it can implement as many interfaces as you want. This is a nice way to give enum constants a common "contract":

```
interface Describable {
    String describe();
}

enum TrafficLight implements Describable {
    RED, YELLOW, GREEN;

    @Override
    public String describe() {
        return "Traffic light: " + name();
    }
}
```

Here, every `TrafficLight` constant shares the same `describe()` implementation. In the next section, we will see how each constant can write its **own** implementation.

Abstract Method and Constant-Specific Body

One of the most powerful features of enums is that each constant can write its **own** implementation of a method — this is called a "constant-specific method body":

```
enum Operation {
    PLUS("+") {
        @Override
        public double apply(double a, double b) {
            return a + b;
        }
    },
    MINUS("-") {
        @Override
        public double apply(double a, double b) {
            return a - b;
        }
    },
    TIMES("*") {
        @Override
        public double apply(double a, double b) {
            return a * b;
        }
    };

    private final String symbol;

    Operation(String symbol) {
        this.symbol = symbol;
    }

    public abstract double apply(double a, double b);

    @Override
    public String toString() {
        return symbol;
    }
}
```

> Writing a separate body for each constant creates a hidden anonymous subclass for each constant behind the scenes — it's a powerful but "heavy" feature. Use it only if there are a few constants and truly differing behavior; if there are dozens of constants and complex logic, separate strategy classes might be more readable (see the next section).

EnumSet

`EnumSet` is a very efficient `Set` implementation based on bit vectors, designed specifically for enum constants — it is almost always faster and consumes less memory than using `HashSet<MyEnum>`:

```
import java.util.EnumSet;

class EnumSetExample {
    public static void main(String[] args) {
        EnumSet<DayWithMethod> weekend = EnumSet.of(DayWithMethod.SATURDAY, DayWithMethod.
SUNDAY);
        EnumSet<DayWithMethod> weekdays = EnumSet.complementOf(weekend);

        System.out.println("Weekend: " + weekend);
        System.out.println("Weekdays: " + weekdays);
    }
}
```

Expected Output

```
Weekend: [SATURDAY, SUNDAY]
Weekdays: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
```

EnumMap

`EnumMap<K,V>` is a `Map` implementation where the keys are enums; internally, instead of a `HashMap`, it uses an array indexed by the constants' `ordinal()` values — this makes it significantly faster than `HashMap<MyEnum, V>`:

```
import java.util.EnumMap;
import java.util.Map;

class EnumMapExample {
    public static void main(String[] args) {
        EnumMap<DayWithMethod, String> plan = new EnumMap<>(DayWithMethod.class);
        plan.put(DayWithMethod.MONDAY, "Study Spring Boot");
        plan.put(DayWithMethod.SATURDAY, "Rest");

        for (Map.Entry<DayWithMethod, String> entry : plan.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Expected Output

```
MONDAY -> Study Spring Boot
SATURDAY -> Rest
```

> `EnumMap` always iterates through constants in their `ordinal()` order (i.e., the order they were defined) — this provides a predictable iteration order unlike `HashMap`, which is useful in cases like listing days of the week in order.

Singleton Pattern

As recommended by Joshua Bloch in the book *Effective Java*, a single-element enum is the safest way to write a Singleton in Java:

```
enum ConfigurationManager {
    INSTANCE;

    private String environment = "production";

    public String getEnvironment() {
        return environment;
    }

    public void setEnvironment(String environment) {
        this.environment = environment;
    }
}
```

Usage: `ConfigurationManager.INSTANCE.getEnvironment()`.

> What makes this pattern so safe is that the JVM does not allow a second instance of enum constants to be created even through serialization and reflection — in the classic "private constructor + static getInstance()" Singleton, these guarantees must be provided manually (and in an error-prone way).

Strategy Pattern

The `Operation` example from before was actually an implementation of the Strategy Pattern: each constant carries a different "strategy" of the same interface (the `apply` method here). Let's reinforce this with a more business-oriented example:

```
enum ShippingStrategy {
    STANDARD {
        @Override
        public double calculate(double weightKg) {
            return weightKg * 2.5;
        }
    },
    EXPRESS {
        @Override
        public double calculate(double weightKg) {
            return weightKg * 5.0 + 10;
        }
    },
    SAME_DAY {
        @Override
        public double calculate(double weightKg) {
            return weightKg * 8.0 + 25;
        }
    };

    public abstract double calculate(double weightKg);
}
```

In the classic Strategy Pattern, you would need to write a separate class for each strategy and a `Factory` to link them together; with an enum, both the strategies and the mapping of "which strategy corresponds to which key" are combined in a single structure without extra code.

Real-World Examples

One of the most common places enums appear in production code is in state machines. Let's model the valid state transitions of an order with an enum:

```
import java.util.EnumSet;
import java.util.Set;

enum OrderStatus {
    PENDING,
    PAID,
    SHIPPED,
    DELIVERED,
    CANCELLED;

    private static final Set<OrderStatus> TERMINAL = EnumSet.of(DELIVERED, CANCELLED);

    public boolean isTerminal() {
        return TERMINAL.contains(this);
    }

    public boolean canTransitionTo(OrderStatus next) {
        return switch (this) {
            case PENDING -> next == PAID || next == CANCELLED;
            case PAID -> next == SHIPPED || next == CANCELLED;
            case SHIPPED -> next == DELIVERED;
            case DELIVERED, CANCELLED -> false;
        };
    }
}
```

This kind of structure places the "which state can transition to which state" rule within the enum itself, rather than in scattered `if/else` blocks — the rule lives in one place, and the compiler guarantees that all possible states (`switch` exhaustiveness) are handled.

Other common real-world uses: HTTP status code categories, user roles/permissions, payment methods, log levels (`DEBUG`, `INFO`, `WARN`, `ERROR`) — all are different applications of the same pattern.

Interview Questions

What advantages does Enum provide over `int` constants? Type safety (compile-time checking), readability, exhaustiveness checking with `switch`, and ready-to-use API like `values()` / `valueOf()`.

Can an enum extend another class? No — every enum implicitly extends `java.lang.Enum` and since Java supports single inheritance, it cannot extend another class. However, it can implement as many interfaces as it wants.

Do `==` and `equals()` give the same result for enum constants? Yes. Since each enum constant exists as a single instance in the JVM, reference comparison with `==` produces the same result as `equals()`. Still, `equals()` is preferred by convention.

Why shouldn't we use `ordinal()` for storing data? Because the order of constants (and thus their ordinal values) can change in the source code; in this case, a previously saved ordinal value may now point to the wrong constant. Store `name()` (or a custom code field) instead.

Are enums `Serializable`? Yes, `java.lang.Enum` already implements `Serializable` and the JVM handles enum serialization specifically (the constant is resolved via `name()`) — so you don't need to write a custom `readObject`/`writeObject`, and it's even recommended not to.

Can an enum be `clone()`ed? No. `Enum.clone()` throws `CloneNotSupportedException` — because it must be guaranteed that each constant has only one instance in the JVM; cloning would break this guarantee.