Module 10 of 20

Enums & records

Enums as fixed sets of instances with fields and behaviour, EnumSet and EnumMap, and records as transparent immutable data with validation.

Two of Java's most useful class forms are special-purpose: an enum is a class with a fixed set of named instances, and a record is a class that is nothing but its components. Both replace error-prone hand-written code — integer constants and parallel arrays on one side, forty-line data classes on the other — with declarations the compiler checks.

This module covers enums from the plain constant list through fields, constructors, per-constant behaviour, interfaces and code lookups, then the EnumSet and EnumMap collections built for them; and records from what they generate through compact constructors, wither methods, nesting, use as map keys and in streams, and the cases where a class is still right.

By the end you model "one of a known set" with enums and "just this data" with records, and reach for EnumMap and compact constructors without thinking.

CONTENTS

1	Enums — a fixed set of named instances	13 min
2	Enums with fields, constructors and behaviour	14 min
3	EnumSet and EnumMap	10 min
4	Records — transparent immutable data	14 min
5	Records in practice — validation, wither methods and modelling	12 min
✓	Checkpoint — Enums & records	22 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Enums — a fixed set of named instances

Before enums, a "day of the week" was `int DAY_MONDAY = 1` and nothing stopped you passing `7` or `42`. A Java `enum` is a class with a **fixed set of instances** created by the JVM, each a named constant, type-safe and comparable. Every enum you write is a full class — the next lesson adds fields and methods — but even the plain form solves most "a value from a known set" problems. This lesson is the plain form and the machinery every enum gets for free.

Declaring

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

Day d = Day.FRIDAY;
Day bad = new Day();          // compile error: enum types may not be instanti
```

`Day` is a class that extends `java.lang.Enum<Day>` implicitly; the seven constants are its only instances, `public static final`, created once when the class initialises. Constant names are UPPER_SNAKE_CASE. An enum can be top-level or nested (implicitly static). It is implicitly `final` — unless constants have bodies (next lesson) — and cannot extend another class (it already extends `Enum`), though it may implement interfaces.

What every enum has

```
Day.values()                  // Day[] of all constants, in declaration order
Day.valueOf("FRIDAY")         // FRIDAY; throws IllegalArgumentException for '
d.name()                      // "FRIDAY" — the identifier, exactly
d.ordinal()                   // 4 — position in declaration order, zero-based
d.toString()                  // "FRIDAY" by default; overridable
d.compareTo(Day.MONDAY)       // positive — by ordinal; enums are Comparable i
d == Day.FRIDAY               // true — the idiomatic comparison
d.equals(Day.FRIDAY)          // also true; == is preferred (null-safe, compil
d.getDeclaringClass()         // Day.class
Enum.valueOf(Day.class, "MONDAY")   // the generic form
```

`==` is correct for enums because each constant is a single instance: no `equals` override to forget, no `NullPointerException` when the left side is null, and a type error if you compare against a different enum. Do not write `d.equals(…)` out of string habit.

Enums in `switch`

```
String kind = switch (d) {
    case SATURDAY, SUNDAY → "weekend";
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY → "weekday";
};                                  // exhaustive over the enum: no default neede
```

Case labels are the bare constant names (`SATURDAY` , not `Day.SATURDAY`). A switch **expression** covering every constant needs no `default` , and then *adding a constant breaks the compile* at every such switch — the safety that makes enums better than integer codes. A switch **statement** does not check exhaustiveness; a `null` selector throws `NullPointerException` in both.

Parsing and printing

`valueOf` is exact and case-sensitive. For user input, normalise first (`Day.valueOf(s.trim().toUpperCase(Locale.ROOT))`) and catch `IllegalArgumentException` , or write a lenient static lookup (next lesson). Never rely on `ordinal()` for persistence — reordering the constants changes every stored number. Store `name()` , and make renaming a constant a deliberate migration.

Comparing and sorting

Enums sort by ordinal — declaration order — with `compareTo` , `Collections.sort` , `TreeSet` . Declare constants in the order that makes sense as a ranking (`LOW, MEDIUM, HIGH`) and `compareTo` becomes meaningful. `Comparator.comparing(Enum::name)` sorts alphabetically if needed.

Iterating

```
for (Day d : Day.values()) { … }
Arrays.stream(Day.values()).filter(…)
EnumSet.allOf(Day.class)          // a Set view, faster than the array for meml
```

`values()` clones the internal array each call — cheap for seven constants, but hoist it out of a hot loop or use `EnumSet`.

Enums are singletons — properly

Each constant is exactly one object per JVM (per class loader). The JVM guarantees this through serialization (`readResolve` is built in) and reflection (`Constructor.newInstance` refuses enums). That makes a one-constant enum the most robust singleton in Java:

```
public enum Registry {
    INSTANCE;
    private final Map<String, Object> entries = new HashMap<>();
    public void put(String k, Object v) { entries.put(k, v); }
}
Registry.INSTANCE.put("x", 1);
```

Type safety versus constants

```
static final int RED = 0, GREEN = 1;
paint(2);                         // compiles; meaningless

enum Color { RED, GREEN }
paint(Color.BLUE);                // compile error: no such constant
paint(2);                         // compile error: int is not a Color
```

The compiler knows the complete set, checks every use, and the debugger shows `GREEN` instead of `1`. Use enums for anything that is "one of a known set": states, kinds, levels, units, commands, directions.

Common mistakes

Mistake	Effect
`Day.valueOf("friday")`	`IllegalArgumentException` — exact name required
Persisting `ordinal()`	Silent corruption after reordering
`switch` on a null enum	`NullPointerException`
`case Day.MONDAY:`	Compile error — unqualified names only (Java 21 relaxes this)
Comparing enums from different types with `==`	Compile error — a feature
Mutable static state in an enum	Global mutable state with a nice name

Interview angle

"Can an enum extend a class?" No — it already extends `Enum`; it can implement interfaces.

"`==` or `equals` for enums?" `==`: single instances, null-safe, type-checked.

"What does `ordinal()` return and should you store it?" Declaration index; no — store `name()`.

"Why is an enum a good singleton?" One instance guaranteed by the JVM, serialization- and reflection-safe.

"Is a switch over an enum exhaustive without `default`?" A switch *expression* covering all constants is; a statement is not checked.

Key takeaways

An enum is a class with a fixed set of named, singleton instances; compare with `==`.

Free machinery: `values()`, `valueOf`, `name()`, `ordinal()`, `compareTo` by declaration order, `toString`.

Switch expressions over all constants need no `default` and break on new constants — the point.

Store `name()`, never `ordinal()`; `valueOf` is exact and throws.

EXERCISE 1

Days and their kinds

Declare `enum Day { MONDAY, …, SUNDAY }`. Read an integer `n` and `n` tokens. For each token: parse it with `Day.valueOf` after trimming and upper-casing; if it is not a day print `<token> unknown`; otherwise print `<NAME> <ordinal> <weekday|weekend> next=<the following day, wrapping from SUNDAY to MONDAY>`. Use a switch expression over the enum for weekday/weekend and `values()` with the ordinal for `next`.

Example: `3` then `friday Sunday funday` →

```
FRIDAY 4 weekday next=SATURDAY
SUNDAY 6 weekend next=MONDAY
funday unknown
```

INPUT 1

```
3
friday Sunday funday
```

OUTPUT 1

```
FRIDAY 4 weekday next=SATURDAY
SUNDAY 6 weekend next=MONDAY
funday unknown
```

INPUT 2

```
2
MONDAY saturday
```

OUTPUT 2

```
MONDAY 0 weekday next=TUESDAY
SATURDAY 5 weekend next=SUNDAY
```

REFERENCE SOLUTION

```java
import java.util.*;

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

public class Main {
    static String kind(Day d) {
        return switch (d) {
            case SATURDAY, SUNDAY -> "weekend";
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "weekday";
        };
    }

    static Day next(Day d) {
        Day[] all = Day.values();
        return all[(d.ordinal() + 1) % all.length];
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String token = in.next();
            try {
                Day d = Day.valueOf(token.trim().toUpperCase(Locale.ROOT));
                System.out.println(d.name() + " " + d.ordinal() + " " + kind(d) + "
            } catch (IllegalArgumentException e) {
                System.out.println(token + " unknown");
            }
        }
    }
}
```

EXERCISE 2

Traffic light cycle

Declare `enum Light { RED, GREEN, YELLOW }` with a method `Light next()` that cycles `RED → GREEN → YELLOW → RED` using `values()` and `ordinal()`. Read a starting colour and an integer `k`; print the light after each of `k` steps, one per line, then `ordinal=<ordinal of the final light>`. A starting colour that is not a constant makes `valueOf` throw — catch it and print `unknown light: <text>` instead.

Example: `RED 4` →

```
GREEN
YELLOW
RED
GREEN
ordinal=1
```

INPUT 1

```
RED 4
```

OUTPUT 1

```
GREEN
YELLOW
RED
GREEN
ordinal=1
```

INPUT 2

```
YELLOW 1
```

OUTPUT 2

```
RED
ordinal=0
```

REFERENCE SOLUTION

```java
import java.util.*;

enum Light {
    RED, GREEN, YELLOW;
    Light next() {
        Light[] all = values();
        return all[(ordinal() + 1) % all.length];
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String start = in.next();
        int k = in.nextInt();
        Light light;
        try {
            light = Light.valueOf(start);
        } catch (IllegalArgumentException e) {
            System.out.println("unknown light: " + start);
            return;
        }
        for (int i = 0; i < k; i++) {
            light = light.next();
            System.out.println(light);
        }
        System.out.println("ordinal=" + light.ordinal());
    }
}
```

QUIZ

1. Why is `==` preferred to `equals` for enum comparison?

A. `equals` is not defined for enums

B. Each constant is a single instance, so `==` is correct, null-safe and type-checked at compile time

C. `==` is faster only

D. `equals` compares ordinals

2. `Day.valueOf("friday")` …

A. Returns `FRIDAY`

B. Throws `IllegalArgumentException` — the name must match exactly

C. Returns null

D. Returns the closest match

3. Storing `ordinal()` in a database is risky because…

A. It is a `long`

B. Reordering or inserting constants changes every stored value's meaning

C. It is not unique

D. It is negative

4. A switch *expression* over an enum that covers every constant and has no `default` …

A. Fails to compile

B. Compiles, and fails to compile later if a constant is added — deliberately

C. Returns null for new constants

D. Needs `yield`

5. Why is a one-constant enum a good singleton?

A. It is shorter to type

B. The JVM guarantees one instance, including across serialization and reflection

C. It is faster

D. It can be extended

ANSWER KEY · ENUMS — A FIXED SET OF NAMED INSTANCES

1. B `a == b` cannot throw on a null left side and will not compile against a different enum type; `equals` gives neither guarantee.

2. B `valueOf` is exact and case-sensitive. Normalise input first, or write a lenient lookup.

3. B Persist `name()` instead; renaming a constant then becomes an explicit migration rather than silent corruption.

4. B Exhaustiveness is checked against the enum's constants; that check is the reason to omit `default`.

5. B `Enum` handles `readResolve` and refuses reflective construction, closing the holes hand-written singletons have.

LESSON 2 · 14 MIN

Enums with fields, constructors and behaviour

An enum is a class, so its constants can carry data and its type can have methods. That turns a bare list of names into a table of related facts — a planet's mass, a currency's symbol, an operation's implementation — that lives with the type instead of in parallel arrays or `switch` statements scattered through the code. This lesson covers enum fields and constructors, per-constant behaviour, enums implementing interfaces, and the lookup patterns you will write for every enum that maps to external codes.

Fields and constructors

```
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    EARTH  (5.976e+24, 6.37814e6),
    JUPITER(1.9e+27,   7.1492e7);

    private final double mass;      // kg
    private final double radius;    // m

    Planet(double mass, double radius) {        // implicitly private; no acce
        this.mass = mass;
        this.radius = radius;
    }

    public double surfaceGravity() { return 6.67300E-11 * mass / (radius * rad
    public double mass() { return mass; }
}

Planet.EARTH.surfaceGravity();       // 9.8
```

Each constant is created by calling the constructor with its arguments, at class initialisation, in declaration order. Rules:

- The constant list comes **first** and ends with a semicolon when anything follows.
- Constructors are implicitly `private`; you cannot call them, only the constant declarations do.
- Fields should be `final`: an enum with mutable fields is a global variable with several names — legal, and almost always a mistake.

Static fields are not accessible from the constructor (they may not be initialised yet, since constants are created first); compute lookup tables in a static block after the constants exist.

Methods and overriding `toString`

```
public enum Level {
    LOW("low"), MEDIUM("medium"), HIGH("high");
    private final String label;
    Level(String label) { this.label = label; }
    @Override public String toString() { return label; }
    public boolean atLeast(Level other) { return compareTo(other) ⩾ 0; }
}
```

`toString` is for display; `name()` stays the identifier (`"LOW"`) and is `final`. Business methods using `ordinal`-based `compareTo` are fine when declaration order is meaningful and documented.

Per-constant behaviour

When each constant needs a different *implementation*, two styles:

Constant-specific class bodies — each constant overrides an abstract method:

```
public enum Operation {
    ADD("+") { public int apply(int a, int b) { return a + b; } },
    SUB("-") { public int apply(int a, int b) { return a - b; } },
    MUL("*") { public int apply(int a, int b) { return a * b; } };

    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    public abstract int apply(int a, int b);
    public String symbol() { return symbol; }
}
Operation.MUL.apply(6, 7);        // 42
```

Each constant with a body is an anonymous subclass (the enum is then not `final`, and `getClass()` differs from `getDeclaringClass()`). The compiler forces every constant to implement the abstract method — adding `DIV` without `apply` is an error.

A functional field — pass the behaviour as a lambda:

```
public enum Operation {
    ADD("+", (a, b) → a + b),
    SUB("-", (a, b) → a - b),
    MUL("*", (a, b) → a * b);

    private final String symbol;
    private final IntBinaryOperator op;
    Operation(String symbol, IntBinaryOperator op) { this.symbol = symbol; th:
    public int apply(int a, int b) { return op.applyAsInt(a, b); }
}
```

Shorter, and the preferred modern form. Either beats a `switch (this)` inside the method, which must be edited for every new constant and cannot be checked for completeness.

Enums implementing interfaces

```
public interface Shape { double area(); }
public enum UnitShape implements Shape {
    SQUARE { public double area() { return 1; } },
    CIRCLE { public double area() { return Math.PI; } };
}
```

An enum cannot extend a class but can implement any number of interfaces. This lets enums plug into polymorphic code and, for sealed interfaces, serve as fixed-instance variants alongside records.

Lookup by code

Enums frequently correspond to external codes — database values, protocol bytes, user strings. `valueOf` handles only the exact constant name; write a lookup for everything else:

```
public enum Status {
    ACTIVE("A"), SUSPENDED("S"), CLOSED("C");

    private final String code;
    Status(String code) { this.code = code; }
    public String code() { return code; }

    private static final Map<String, Status> BY_CODE = new HashMap<>();
    static {
        for (Status s : values()) BY_CODE.put(s.code, s);       // built once,
    }
    public static Status fromCode(String code) {
        Status s = BY_CODE.get(code);
        if (s == null) throw new IllegalArgumentException("unknown status code
        return s;
    }
    public static Optional<Status> tryFromCode(String code) { return Optional.
}
```

The static block runs after all constants are constructed (the constants are the first static initialisers), so `values()` is complete. A `Map` beats a linear scan for more than a handful of constants; `Arrays.stream(values()).filter(…).findFirst()` is fine for small enums.

Enums and `switch`, revisited

With behaviour on the enum, most `switch (status)` blocks elsewhere in the code become `status.method()` calls — polymorphism instead of dispatch tables. Keep `switch` for logic that belongs to the *caller* (a UI deciding colours), and put logic that belongs to the *concept* on the enum.

Initialisation order pitfalls

```
enum Bad {
    A, B;
    private static int count = 0;
    Bad() { count++; }          // compile error: illegal reference to static
}
```

Constants are constructed before other static fields are initialised, so constructors cannot touch static fields. Count in a static block instead, or use `values().length`.

`EnumSet` and `EnumMap` preview

Collections of enums should be `EnumSet` (a bit set) and `EnumMap` (an array indexed by ordinal) — orders of magnitude cheaper than `HashSet` / `HashMap` and iterating in declaration order. Next lesson.

Interview angle

"Can an enum have a constructor?" Yes, implicitly private, run once per constant at class initialisation.

"How do you give each constant different behaviour?" Abstract method with constant-specific bodies, or a functional-interface field.

"Why can't an enum constructor read a static field?" Constants initialise first; the static field would still be default.

"How do you map a database code to an enum?" A static `Map<String, Enum>` built in a static block, with a `fromCode` factory.

"Can an enum implement an interface?" Yes — any number.

Key takeaways

Constants first, then `final` fields, a private constructor and methods; each constant is built with its arguments.

Per-constant behaviour: an abstract method overridden per constant, or a lambda-typed field — not a `switch (this)`.

Enums implement interfaces and plug into polymorphic code.

Lookup by external code through a static map built in a static block; keep `name()` for persistence.

Constructors run before other statics — compute tables afterwards.

EXERCISE 1

Operations as an enum

Declare `enum Op` with constants `ADD("+")`, `SUB("-")`, `MUL("*")`, `DIV("/")`, a `symbol` field, an abstract `long apply(long a, long b)` implemented per constant (`DIV` uses integer division and throws `ArithmeticException` on zero — let it propagate to a `catch` in `main` that prints `error`), and a static `fromSymbol(String)` lookup built from a static `Map`.

Read an integer `n` and `n` lines `<a> <symbol> <b>`; print each result, or `error` for division by zero, or `unknown` for a symbol that is not an operation.

Example: `3` then `6 * 7`, `1 / 0`, `2 ^ 3` →

```
42
error
unknown
```

INPUT 1

```
3
6 * 7
1 / 0
2 ^ 3
```

OUTPUT 1

```
42
error
unknown
```

INPUT 2

```
3
10 - 15
7 / 2
3 + 4
```

OUTPUT 2

```
-5
3
7
```

REFERENCE SOLUTION

```java
import java.util.*;

enum Op {
    ADD("+") { long apply(long a, long b) { return a + b; } },
    SUB("-") { long apply(long a, long b) { return a - b; } },
    MUL("*") { long apply(long a, long b) { return a * b; } },
    DIV("/") { long apply(long a, long b) { return a / b; } };

    private final String symbol;
    Op(String symbol) { this.symbol = symbol; }
    abstract long apply(long a, long b);

    private static final Map<String, Op> BY_SYMBOL = new HashMap<>();
    static {
        for (Op op : values()) BY_SYMBOL.put(op.symbol, op);
    }
    static Op fromSymbol(String s) { return BY_SYMBOL.get(s); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < n; i++) {
            String[] t = in.nextLine().trim().split("\\s+");
            long a = Long.parseLong(t[0]);
            long b = Long.parseLong(t[2]);
            Op op = Op.fromSymbol(t[1]);
            if (op == null) {
                System.out.println("unknown");
                continue;
            }
            try {
                System.out.println(op.apply(a, b));
            } catch (ArithmeticException e) {
                System.out.println("error");
            }
        }
    }
}
```

EXERCISE 2

Coins with values

Declare `enum Coin { QUARTER(25), DIME(10), NICKEL(5), PENNY(1) }` with a `cents` field, a constructor and an accessor. Read an integer `n` and `n` amounts in cents. For each, make change

greedily from the largest coin down (`values()` is already in that order) and print `<amount>: <k> QUARTER, <k> DIME, …` listing only coins used, then `coins=<total count>`.

Example: `2` then `41 30` →

```
41: 1 QUARTER, 1 DIME, 1 NICKEL, 1 PENNY coins=4
30: 1 QUARTER, 1 NICKEL coins=2
```

INPUT 1

```
2
41 30
```

OUTPUT 1

```
41: 1 QUARTER, 1 DIME, 1 NICKEL, 1 PENNY
coins=4
30: 1 QUARTER, 1 NICKEL coins=2
```

INPUT 2

```
1
99
```

OUTPUT 2

```
99: 3 QUARTER, 2 DIME, 4 PENNY coins=9
```

REFERENCE SOLUTION

```java
import java.util.*;

enum Coin {
    QUARTER(25), DIME(10), NICKEL(5), PENNY(1);
    private final int cents;
    Coin(int cents) { this.cents = cents; }
    int cents() { return cents; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            int amount = in.nextInt();
            int left = amount, total = 0;
            List<String> parts = new ArrayList<>();
            for (Coin c : Coin.values()) {
                int k = left / c.cents();
                if (k > 0) {
                    parts.add(k + " " + c);
                    total += k;
                    left -= k * c.cents();
                }
            }
            System.out.println(amount + ": " + String.join(", ", parts) + " coins="
        }
    }
}
```

QUIZ

1. An enum constructor is…

A. Public

B. Implicitly private, called once per constant at class initialisation

C. Static

D. Not allowed

2. `enum E { A, B; static int count = 0; E() { count++; } }` fails because…

A. Enums cannot have static fields

B. Constants are constructed before other static fields are initialised, so the constructor may not reference them

C. `count` must be final

D. Constructors cannot have bodies

3. Which gives each constant its own implementation of `apply` most safely?

A. A `switch (this)` inside `apply`

B. An abstract `apply` with a body per constant, or a functional-interface field

C. Reflection

D. Overloading

4. A `fromCode(String)` lookup should be built…

A. Inside the constructor

B. In a static block after the constants, into a `Map`

C. With `valueOf`

D. By scanning `ordinal()`

5. Can an enum implement an interface?

A. No

B. Yes, any number

C. Only functional interfaces

D. Only if it has no fields

ANSWER KEY · ENUMS WITH FIELDS, CONSTRUCTORS AND BEHAVIOUR

1. B Only the constant declarations invoke it; `new` on an enum is a compile error.

2. B Count in a static block after the constants, or use `values().length`.

3. B The compiler forces every constant to supply an implementation; a `switch (this)` silently misses new constants.

4. B `valueOf` only knows constant names; a static map keyed by the external code is built once when the class initialises.

5. B Enums cannot extend classes (they extend `Enum`) but implement interfaces freely — including sealed ones.

LESSON 3 · 10 MIN

EnumSet and EnumMap

A set of days, a map from status to count, a bitmask of permissions — collections keyed by enum constants are everywhere, and Java has two classes built for exactly that shape. `EnumSet` stores membership as bits; `EnumMap` stores values in an array indexed by ordinal. Both are faster and smaller than their hash-based cousins, iterate in declaration order, and are what a reviewer expects to see wherever the element or key type is an enum.

`EnumSet`

```
EnumSet<Day> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY);
EnumSet<Day> weekdays = EnumSet.complementOf(weekend);
EnumSet<Day> all = EnumSet.allOf(Day.class);
EnumSet<Day> none = EnumSet.noneOf(Day.class);              // an empty set o
EnumSet<Day> midweek = EnumSet.range(Day.TUESDAY, Day.THURSDAY);
EnumSet<Day> copy = EnumSet.copyOf(someCollectionOfDays);

weekend.contains(Day.SUNDAY)        // true — a bit test
weekdays.add(Day.SATURDAY);         // it is a normal mutable Set
for (Day d : weekdays) …            // declaration order: MONDAY … FRIDAY
```

Inside, an `EnumSet` of up to 64 constants is a single `long`; membership, add and remove are bit operations, union and intersection (`addAll`, `retainAll`) are `|` and `&`. Larger enums use a `long[]`. It is a full `Set<E>` — `size`, `removeIf`, `stream()` all work — with iteration in ordinal order. It does not accept `null`.

Use it for: flag sets (`EnumSet<Permission>`), "which days" / "which options" selections, and any `Set` whose element type is an enum. It replaces integer bitmasks (`FLAG_A | FLAG_B`) with something type-safe and readable — *Effective Java* item 36.

There is no immutable `EnumSet`; wrap with `Collections.unmodifiableSet` or copy defensively.

`EnumMap`

```
EnumMap<Status, Integer> counts = new EnumMap<>(Status.class);      // the cl
for (Order o : orders) counts.merge(o.status(), 1, Integer::sum);

counts.get(Status.ACTIVE)            // an array read at index ordinal()
counts.keySet()                      // in declaration order
counts.getOrDefault(Status.CLOSED, 0)
```

An `EnumMap` is an `Object[]` of size `values().length`; `get` and `put` index by ordinal with no hashing and no boxing of the key. Keys iterate in declaration order. Null keys are rejected; null values are allowed.

Use it for: counts and tallies per constant, lookup tables keyed by enum (`EnumMap<Level, Color>`), state-machine transition tables (`EnumMap<State, EnumSet<State>>`). Compared with a `HashMap<Status, Integer>` it is faster, smaller, ordered, and says in its type that the key set is closed.

Why not `ordinal()` and an array?

```
int[] counts = new int[Status.values().length];
counts[status.ordinal()]++;             // works — and is exactly what EnumMap
```

The manual version couples the code to ordinal order and needs a comment; `EnumMap` says the same thing in the type system with a real `Map` API. Reach for the raw array only in a measured hot loop.

`Enum.valueOf`, `EnumSet` and `switch` together

A typical flags-style API:

```
public enum Permission { READ, WRITE, DELETE, ADMIN }

public final class Role {
    private final EnumSet<Permission> permissions;
    public Role(Collection<Permission> ps) { this.permissions = ps.isEmpty() 
    public boolean can(Permission p) { return permissions.contains(p); }
    public Set<Permission> permissions() { return Collections.unmodifiableSet(
}
```

`EnumSet.copyOf(collection)` throws on an empty non-`EnumSet` collection (it cannot infer the element type), hence the `noneOf` branch — a small trap worth knowing.

Streams and enums

```
Map<Status, Long> tally = orders.stream()
    .collect(Collectors.groupingBy(Order::status, () → new EnumMap<>(Status.c
```

`groupingBy` with a map factory produces an `EnumMap`; without it you get a `HashMap` in arbitrary order. `Collectors.toCollection(() → EnumSet.noneOf(Day.class))` does the same for sets.

Performance in one table

Operation	`HashSet` / `HashMap`	`EnumSet` / `EnumMap`
contains / get	hash + equals	one bit test / one array read
Memory per element	entry object + boxing	one bit / one array slot
Iteration order	unspecified	declaration order
Null elements/keys	allowed	rejected

Interview angle

"How would you represent a set of flags type-safely?" `EnumSet<Flag>` — a bit vector under a `Set` API.

"Why prefer `EnumMap` to `HashMap` for enum keys?" Array indexing by ordinal: faster, smaller, ordered.

"What does `EnumSet.copyOf(emptyList)` do?" Throws `IllegalArgumentException` — use `noneOf(Class)` for empties.

"Is `EnumSet` thread-safe?" No, like the other basic collections; wrap or copy.

Key takeaways

`EnumSet` : bit-vector `Set` of enum constants — `of` , `allOf` , `noneOf` , `complementOf` , `range` ; ordinal-order iteration; no nulls.

`EnumMap` : array-backed `Map` keyed by enum — pass the class token; ordered; fast tallies and lookup tables.

Both replace hash collections wherever the element/key is an enum, and replace integer bitmasks and ordinal-indexed arrays.

`copyOf` of an empty plain collection throws; `groupingBy` takes a map factory for `EnumMap` .

EXERCISE 1

Tally by status

Declare `enum Status { OPEN, PAID, SHIPPED, CLOSED }` . Read an integer `n` and `n` status names (exact upper-case names). Count them in an `EnumMap<Status, Integer>` and print one line per constant **in declaration order**, `<NAME>=<count>` (0 for statuses never seen). Then print `seen=<the EnumSet of statuses that appeared, via toString>` and `missing=<EnumSet.complementOf(seen)>` .

Example: `3` then `PAID OPEN PAID` →

```
OPEN=1
PAID=2
SHIPPED=0
CLOSED=0
seen=[OPEN, PAID]
missing=[SHIPPED, CLOSED]
```

INPUT 1

```
3
PAID OPEN PAID
```

OUTPUT 1

```
OPEN=1
PAID=2
SHIPPED=0
CLOSED=0
seen=[OPEN, PAID]
missing=[SHIPPED, CLOSED]
```

INPUT 2

```
0
```

OUTPUT 2

```
OPEN=0
PAID=0
SHIPPED=0
CLOSED=0
seen=[]
missing=[OPEN, PAID, SHIPPED, CLOSED]
```

REFERENCE SOLUTION

```java
import java.util.*;

enum Status { OPEN, PAID, SHIPPED, CLOSED }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        EnumMap<Status, Integer> counts = new EnumMap<>(Status.class);
        EnumSet<Status> seen = EnumSet.noneOf(Status.class);
        for (int i = 0; i < n; i++) {
            Status s = Status.valueOf(in.next());
            counts.merge(s, 1, Integer::sum);
            seen.add(s);
        }
        for (Status s : Status.values()) {
            System.out.println(s + "=" + counts.getOrDefault(s, 0));
        }
        System.out.println("seen=" + seen);
        System.out.println("missing=" + EnumSet.complementOf(seen));
    }
}
```

EXERCISE 2

Permission sets

Declare `enum Perm { READ, WRITE, EXEC, ADMIN }`. Read two lines, each a space-separated list of permissions (or `-` for none), into two `EnumSet`s `a` and `b`. Print `union=<…>`, `intersection=<…>`, `onlyA=<…>` (`a` minus `b`) and `missingFromA=<EnumSet.complementOf(a)>`, each as the set's `toString` — EnumSets print in declaration order whatever the input order. Then, with an `EnumMap<Perm, Integer>`, count how many of the two sets contain each permission and print it.

Example: `WRITE READ` / `READ EXEC` →

```
union=[READ, WRITE, EXEC]
intersection=[READ]
onlyA=[WRITE]
missingFromA=[EXEC, ADMIN]
counts={READ=2, WRITE=1, EXEC=1}
```

INPUT 1

```
WRITE READ
READ EXEC
```

OUTPUT 1

```
union=[READ, WRITE, EXEC]
intersection=[READ]
onlyA=[WRITE]
missingFromA=[EXEC, ADMIN]
counts={READ=2, WRITE=1, EXEC=1}
```

INPUT 2

```
-
ADMIN
```

OUTPUT 2

```
union=[ADMIN]
intersection=[]
onlyA=[]
missingFromA=[READ, WRITE, EXEC, ADMIN]
counts={ADMIN=1}
```

REFERENCE SOLUTION

```
import java.util.*;

enum Perm { READ, WRITE, EXEC, ADMIN }

public class Main {
    static EnumSet<Perm> parse(String line) {
        EnumSet<Perm> set = EnumSet.noneOf(Perm.class);
        for (String t : line.trim().split("\\s+")) if (!t.isEmpty() && !t.equals("-'
        return set;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        EnumSet<Perm> a = parse(in.nextLine()), b = parse(in.nextLine());
        EnumSet<Perm> union = EnumSet.copyOf(a); union.addAll(b);
        EnumSet<Perm> inter = EnumSet.copyOf(a); inter.retainAll(b);
        EnumSet<Perm> onlyA = EnumSet.copyOf(a); onlyA.removeAll(b);
        System.out.println("union=" + union);
        System.out.println("intersection=" + inter);
        System.out.println("onlyA=" + onlyA);
        System.out.println("missingFromA=" + EnumSet.complementOf(a));
        EnumMap<Perm, Integer> counts = new EnumMap<>(Perm.class);
        for (Perm p : a) counts.merge(p, 1, Integer::sum);
        for (Perm p : b) counts.merge(p, 1, Integer::sum);
        System.out.println("counts=" + counts);
    }
}
```

QUIZ

1. Internally, an `EnumSet` of an enum with 20 constants is…

A. A hash table

B. A single `long` used as a bit vector

C. An `ArrayList`

D. A tree

2. `new EnumMap<>(Status.class)` needs the class token because…

A. Generics are erased, and the map needs the constant count to size its array

B. It is a convention

C. To sort the keys

D. To make it immutable

3. Iteration order of `EnumMap.keySet()` is…

A. Insertion order

B. Hash order

C. Declaration (ordinal) order

D. Reverse order

4. `EnumSet.copyOf(new ArrayList<Day>())` (an empty list)…

A. Returns an empty EnumSet

B. Throws `IllegalArgumentException` — the element type cannot be inferred

C. Returns null

D. Compile error

5. An `EnumSet<Permission>` replaces…

A. Strings

B. Integer bitmask flags (`FLAG_A | FLAG_B`) with a type-safe `Set`

C. Enums

D. Arrays

ANSWER KEY · ENUMSET AND ENUMMAP

1. B Up to 64 constants fit in one `long`; membership is a bit test and set operations are bitwise.

2. A The backing array has one slot per constant, indexed by ordinal.

3. C The array is indexed by ordinal, so keys come out in the enum's declared order.

4. B Use `EnumSet.noneOf(Day.class)` for empties; `copyOf` of an `EnumSet` argument is always fine.

5. B Same bit-level efficiency, plus the type system and the `Set` API.

LESSON 4 · 14 MIN

Records — transparent immutable data

Most classes that hold data — a point, a pair, a DTO, a parse result — need the same forty lines: private final fields, a constructor, accessors, `equals`, `hashCode`, `toString`. Java 16's **records** generate all of it from one line, and in doing so make a promise the compiler enforces: a record is a *transparent, immutable carrier of its components*. This lesson covers what a record generates, what you can customise, and the rules that keep the promise.

Declaring a record

```
public record Point(int x, int y) { }

Point p = new Point(3, 4);
p.x()            // 3   — accessor, named after the component (no "get")
p.y()            // 4
p.toString()     // "Point[x=3, y=4]"
p.equals(new Point(3, 4))   // true — component-wise
p.hashCode()     // combined from x and y, consistent with equals
```

The header `(int x, int y)` declares the **components**. From it the compiler generates:

- a `private final` field per component;
- a **canonical constructor** taking all components in order;
- a public accessor per component, named exactly like it;
- `equals`, `hashCode` and `toString` over all components.

Records are implicitly `final`, extend `java.lang.Record` (so cannot extend anything else), and may implement interfaces. They can be top-level, nested (implicitly static) or **local** — declared inside a method, handy for a temporary pair in an algorithm.

What you can add

```java
public record Range(int lo, int hi) implements Comparable<Range> {

    // compact canonical constructor: validate / normalise; assignment is impl
    public Range {
        if (lo > hi) throw new IllegalArgumentException(lo + " > " + hi);
    }

    // extra constructors must delegate to the canonical one
    public Range(int single) { this(single, single); }

    // instance methods
    public int length() { return hi - lo; }
    public boolean contains(int x) { return lo ⩽ x && x < hi; }

    // static members and factories
    public static Range empty() { return new Range(0, 0); }
    public static final Range UNIT = new Range(0, 1);

    // an accessor may be overridden (rarely wise); its signature is fixed
    @Override public int lo() { return lo; }

    @Override public int compareTo(Range o) { return Integer.compare(lo, o.lo)
}
```

The **compact constructor** has no parameter list; it runs before the fields are assigned and may reassign the *parameters* (`lo = Math.min(lo, hi)`) to normalise them. It cannot assign `this.lo` directly — the assignment happens automatically after the body.

What you cannot do

No instance fields beyond the components. `private int cache;` in a record is a compile error. Static fields are fine.

No setters, no mutation of components: the fields are final. "Modifying" means creating a new record: `new Point(p.x() + 1, p.y())` — a `withX(int)` method is the idiom if you want it.

No `extends`; records may not be abstract; components cannot be `volatile` / `transient` in the usual sense.

Accessor names are fixed to the component names; a `getX()` alongside is legal but pointless.

Equality is by value, always

`equals` compares each component with `==` for primitives (`Double.compare` semantics for `float` / `double`) and `equals` for references. Two records with equal components are equal — which is what a data carrier should mean, and what makes records good map keys and set elements without any hand-written code. Caveat: an array component uses `Object.equals` on the array (identity), so `record Bag(int[] items)` compares arrays by reference — use a `List` instead.

Records and collections: shallow immutability

```
public record Team(String name, List<String> members) {
    public Team {
        members = List.copyOf(members);      // defensive, immutable copy in t
    }
}
```

A record's *fields* are final; the *objects* they refer to are whatever they are. A `List` component can be mutated by whoever holds the list unless you copy it on the way in. `List.copyOf` produces an immutable copy (and rejects nulls); for maps `Map.copyOf`. Make this a habit for every collection or array component.

Records as tuples and multi-value returns

```
record MinMax(int min, int max) { }
static MinMax minMax(int[] a) { … return new MinMax(lo, hi); }
MinMax r = minMax(values);
r.min(); r.max();
```

The answer to "how do I return two values" (Module 5) is a record — named, typed, and one line to declare, even locally inside the method that needs it.

Records with sealed interfaces

```
sealed interface Shape permits Circle, Square { }
record Circle(double r) implements Shape { }
record Square(double side) implements Shape { }
```

Records are the natural leaves of a sealed hierarchy (Module 9): final, immutable, with generated equality. Java 21's record patterns destructure them in a switch: `case Circle(double r) → Math.PI * r * r`.

Records versus classes

Need	Use
Immutable data with value equality	record
Mutable state, identity, lifecycle	class
Inheritance from a class	class (records cannot extend)
Extra hidden state or caching fields	class (records forbid instance fields)
JPA entities, JavaBeans frameworks needing setters	class
DTOs, keys, tuples, messages, configuration values	record

Records are not "classes with less typing"; they are a *semantic* declaration that the type is nothing but its components. If that is not true — if two instances with equal components should be distinguishable, or the state changes — it is not a record.

Serialization, reflection and frameworks

Records serialise through their canonical constructor (safer than field-poking), and `Class.getRecordComponents()` exposes the components to frameworks. Jackson, Gson and most modern libraries support records directly. Older frameworks that require a no-arg constructor and setters do not.

Interview angle

"What does a record generate?" Final fields, canonical constructor, accessors, `equals` / `hashCode` / `toString`.

"Can a record have a mutable field?" No instance fields besides the components, all final; the referenced objects may be mutable — copy them in.

"Can a record extend a class?" No; it can implement interfaces.

"What is a compact constructor?" The canonical constructor without a parameter list, for validation and normalisation before the implicit assignments.

"Are records immutable?" Shallowly: components cannot be reassigned; deep immutability depends on the component types.

Key takeaways

`record Name(T a, U b)` generates fields, constructor, accessors `a()` / `b()`, and value-based `equals` / `hashCode` / `toString`.

Customise with a compact constructor (validate, normalise), extra constructors delegating to the canonical one, methods, statics, interfaces.

No extra instance fields, no setters, no `extends`; equality is by components — array components compare by reference.

Copy collection components defensively (`List.copyOf`) for real immutability.

Use records for data; classes for identity and mutation.

EXERCISE 1

Points as records

Declare `record Point(int x, int y)` with a method `double distanceTo(Point o)` and a static factory `origin()`. Read an integer `n` and `n` pairs `x y`. Put every point into a `HashSet<Point>` and print `distinct=<size>`; print `hasOrigin=<set contains Point.origin()>`; then print each **distinct** point in input order (use a `LinkedHashSet`) as its `toString()` followed by its distance to the origin with two decimals.

Example: `3` then `3 4`, `0 0`, `3 4` →

```
distinct=2
hasOrigin=true
Point[x=3, y=4] 5.00
Point[x=0, y=0] 0.00
```

INPUT 1

```
3
3 4
0 0
3 4
```

OUTPUT 1

```
distinct=2
hasOrigin=true
Point[x=3, y=4] 5.00
Point[x=0, y=0] 0.00
```

INPUT 2

```
2
1 1
-1 -1
```

OUTPUT 2

```
distinct=2
hasOrigin=false
Point[x=1, y=1] 1.41
Point[x=-1, y=-1] 1.41
```

REFERENCE SOLUTION

```
import java.util.*;

record Point(int x, int y) {
    double distanceTo(Point o) {
        double dx = x - o.x, dy = y - o.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
    static Point origin() { return new Point(0, 0); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Set<Point> points = new LinkedHashSet<>();
        for (int i = 0; i < n; i++) points.add(new Point(in.nextInt(), in.nextInt()));
        System.out.println("distinct=" + points.size());
        System.out.println("hasOrigin=" + points.contains(Point.origin()));
        for (Point p : points) {
            System.out.printf("%s %.2f%n", p, p.distanceTo(Point.origin()));
        }
    }
}
```

EXERCISE 2

Rational numbers as records

Write `record Rational(long num, long den)` whose **compact constructor** rejects a zero denominator (`IllegalArgumentException`), moves any sign to the numerator and reduces by the gcd — so `new Rational(2, -4)` becomes `-1/2` . Add `plus` and `times` returning new records, and

`toString` as `num/den` (just `num` when `den == 1`). Read `n` lines `a b op c d` and print `a/b op c/d = <result>`; record equality then works by value: also print `equal=<result.equals(new Rational(…))>` where the second is the same fraction unreduced, given on the line as `e f`.

Example: `1 2 + 1 3 5 6` (one line, `n = 1`) →

```
1/2 + 1/3 = 5/6 equal=true
```

INPUT 1

```
1
1 2 + 1 3 5 6
```

OUTPUT 1

```
1/2 + 1/3 = 5/6 equal=true
```

INPUT 2

```
2
2 -4 * 3 1 -3 2
1 3 + 2 3 2 2
```

OUTPUT 2

```
-1/2 * 3 = -3/2 equal=true
1/3 + 2/3 = 1 equal=true
```

REFERENCE SOLUTION

```java
import java.util.*;

record Rational(long num, long den) {
    Rational {
        if (den == 0) throw new IllegalArgumentException("zero denominator");
        if (den < 0) { num = -num; den = -den; }
        long g = gcd(num, den);
        if (g > 1) { num /= g; den /= g; }
    }
    Rational plus(Rational o) { return new Rational(num * o.den + o.num * den, den *
    Rational times(Rational o) { return new Rational(num * o.num, den * o.den); }
    @Override public String toString() { return den == 1 ? String.valueOf(num) : num
    static long gcd(long a, long b) { return b == 0 ? Math.abs(a) : gcd(b, a % b); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            long a = in.nextLong(), b = in.nextLong();
            String op = in.next();
            long c = in.nextLong(), d = in.nextLong();
            long e = in.nextLong(), f = in.nextLong();
            Rational x = new Rational(a, b), y = new Rational(c, d);
            Rational r = op.equals("+") ? x.plus(y) : x.times(y);
            System.out.println(x + " " + op + " " + y + " = " + r + " equal=" + r.eq
        }
    }
}
```

QUIZ

1. Which is **not** generated for `record P(int a, String b)`?

A. Accessors `a()` and `b()`

B. A canonical constructor `P(int, String)`

C. Setters `setA` / `setB`

D. `equals`, `hashCode`, `toString`

2. A record may…

A. Extend another class

B. Declare an extra instance field

C. Implement interfaces and declare methods and static members

D. Be abstract

3. A compact constructor `public Range { if (lo > hi) throw …; }` …

A. Replaces the fields

B. Runs before the implicit field assignments and may validate or reassign the parameters

C. Is a static factory

D. Runs after construction

4. `record Bag(int[] items)` — `new Bag(new int[]{1}).equals(new Bag(new int[]{1}))` is…

A. `true`

B. `false` — array components compare by reference

C. A compile error

D. `true` only with `Arrays.equals`

5. Records are described as *shallowly* immutable because…

A. Components can be reassigned

B. Component fields are final, but the objects they reference may be mutable unless copied in

C. They can be subclassed

D. Accessors can be overridden

ANSWER KEY · RECORDS — TRANSPARENT IMMUTABLE DATA

1. C Records are immutable: no setters, and the fields are final.

2. C Records are final, extend `Record`, and allow no instance fields beyond the components — but methods, statics and interfaces are fine.

3. B It has no parameter list; the component parameters are in scope and are assigned to the fields after the body.

4. B Generated `equals` calls `Object.equals` on reference components; arrays do not override it. Use a `List`.

5. B A `List` component is still a mutable list unless the compact constructor stores `List.copyOf(list)`.

LESSON 5 · 12 MIN

Records in practice — validation, wither methods and modelling

Knowing what a record generates is the easy half. The practical half is the handful of patterns that come up every time you use them: validating and normalising in the compact constructor, producing modified copies, nesting records into larger models, using them as map keys and in streams, and recognising the situations where a record is the wrong tool. This lesson is those patterns, with the idioms that experienced Java code uses.

Validate and normalise in the compact constructor

```
public record Email(String address) {
    public Email {
        Objects.requireNonNull(address, "address");
        address = address.strip().toLowerCase(Locale.ROOT);          // normal
        if (!address.contains("@")) throw new IllegalArgumentException("not an
    }
}
```

The compact constructor sees the raw parameters, may check and rewrite them, and the rewritten values are what get stored. Every path that creates an `Email` — including deserialisation — passes through it, so an `Email` object is *always* valid and canonical. This is the same "establish the invariant in the constructor" rule as for classes, with less code.

Wither methods for modified copies

```
public record Config(String host, int port, boolean tls) {
    public Config withPort(int newPort) { return new Config(host, newPort, tls
    public Config withTls(boolean newTls) { return new Config(host, port, newT
}
Config c = base.withPort(8443).withTls(true);
```

Records are immutable, so "change one field" means "construct a new record with one difference". A `withX` method per commonly changed component keeps call sites readable. (Java has discussed `with` syntax for records; until then, write the methods.)

Nested records model structured data

```
record Address(String street, String city, String postcode) { }
record Customer(long id, String name, Address address, List<String> tags) {
    public Customer {
        tags = List.copyOf(tags);
    }
}
```

A record of records is a tree of immutable values with structural equality all the way down — comparing two `Customer`s compares their addresses component-wise. This is the natural shape for configuration, API payloads and domain values. Keep collections inside records immutable (`List.copyOf`), and the whole tree is safe to share and cache.

Records as map keys and set elements

```
record Cell(int row, int col) { }
Map<Cell, Integer> grid = new HashMap<>();
grid.put(new Cell(2, 3), 7);
grid.get(new Cell(2, 3));            // 7 — a different instance, equal by va
Set<Cell> visited = new HashSet<>();
```

Generated `equals` / `hashCode` and immutability make records ideal keys — the pair-of-ints key that used to require a hand-written class or an encoded `long`. Grid searches, memoisation keyed on `(i, j)`, and composite ids are all one-line records.

Records in streams

```
record NameLength(String name, int length) { }
List<NameLength> longest = names.stream()
    .map(n -> new NameLength(n, n.length()))
    .sorted(Comparator.comparingInt(NameLength::length).reversed())
    .limit(3)
    .toList();
```

Intermediate results that carry two or three values want a record — a local record inside the method if it is used nowhere else. `Map.entry` is the anonymous alternative; a record names

the fields.

Records implementing interfaces

```
interface Priced { long priceCents(); }
record Book(String title, long priceCents) implements Priced { }      // the c
record Ticket(String event, long priceCents) implements Priced { }
```

An accessor with the right name and type implements an interface method automatically. Combined with sealed interfaces this gives closed variant types; combined with `Comparable` it gives sortable values.

Local records for clarity

```
static String busiestHour(List<Event> events) {
    record Count(int hour, long n) { }
    return events.stream()
        .collect(Collectors.groupingBy(Event::hour, Collectors.counting()))
        .entrySet().stream()
        .map(e → new Count(e.getKey(), e.getValue()))
        .max(Comparator.comparingLong(Count::n))
        .map(c → c.hour() + ":00")
        .orElse("none");
}
```

A record declared inside a method is implicitly static (it cannot capture locals) and invisible outside — perfect for a shape that exists only to make one pipeline readable.

When a record is wrong

The type has identity: two users with the same name and email are still different users. Use a class with an id.

The state changes: a shopping cart, a connection, a counter. A record would force `new` on every change and lose identity.

You need hidden state: a cache, a lazily computed field, a listener list. Records forbid extra instance fields.

A framework needs a no-arg constructor and setters (JPA entities, older serialisation). Records are for the data that crosses the boundary, not the managed entity itself.

Inheritance from a base class is required. Records extend `Record` only; use interfaces or composition.

Records and `null`

Components may be `null` unless you forbid it; `List.copyOf` and `Map.copyOf` reject null elements, and `Objects.requireNonNull` in the compact constructor rejects null components. Decide per component and enforce it in the constructor; the generated `equals` / `hashCode` / `toString` all tolerate nulls.

Migrating a data class to a record

If a class has only final fields, a constructor assigning them, accessors, and generated `equals` / `hashCode` / `toString`, it is a record waiting to happen: replace the body with the header, keep validation as a compact constructor, and rename `getX()` calls to `x()` (or keep `getX` as extra methods during a transition). Behaviour that belonged to the class stays as record methods.

Interview angle

"How do you enforce invariants in a record?" Compact constructor: validate, normalise, throw.

"How do you 'modify' a record?" Construct a new one — wither methods.

"Are records good `HashMap` keys?" Yes: immutable with value-based `equals` / `hashCode`.

"When would you not use a record?" Identity, mutability, hidden state, framework constraints, class inheritance.

"Can a record be declared inside a method?" Yes — a local record, implicitly static.

Key takeaways

Validate and normalise in the compact constructor; every instance is then canonical.

Immutable means wither methods (`withX`) for changes and `List.copyOf` for collection components.

Nest records for structured values; use them as map keys, stream intermediates and interface implementations.

Local records tidy single-method pipelines.

Not for entities with identity, mutable state, hidden fields or framework-required setters.

EXERCISE 1

Validated emails, grouped by domain

Declare `record Email(String address)` whose compact constructor trims and lower-cases the address and throws `IllegalArgumentException` if it does not contain exactly one `@` with non-empty parts on both sides. Add `String domain()` .

Read an integer `n` and `n` tokens; try to build an `Email` from each, counting rejects. Group the valid ones by domain in a `TreeMap<String, List<Email>>` and print `<domain> <count>` per domain in sorted order, then `rejected=<count>` . Duplicates (after normalisation) count once: collect into a `LinkedHashSet<Email>` first.

Example: `4` then `Ada@Example.com ada@example.com bob@test.org broken` →

```
example.com 1
test.org 1
rejected=1
```

INPUT 1

```
4
Ada@Example.com ada@example.com
bob@test.org broken
```

OUTPUT 1

```
example.com 1
test.org 1
rejected=1
```

INPUT 2

```
3
@x.com a@ a@@b.com
```

OUTPUT 2

```
rejected=3
```

REFERENCE SOLUTION

```java
import java.util.*;

record Email(String address) {
    public Email {
        address = address.trim().toLowerCase(Locale.ROOT);
        int at = address.indexOf('@');
        if (at ⩽ 0 || at ≠ address.lastIndexOf('@') || at == address.length() - 1)
            throw new IllegalArgumentException("not an email: " + address);
        }
    }
    String domain() { return address.substring(address.indexOf('@') + 1); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Set<Email> valid = new LinkedHashSet<>();
        int rejected = 0;
        for (int i = 0; i < n; i++) {
            try {
                valid.add(new Email(in.next()));
            } catch (IllegalArgumentException e) {
                rejected++;
            }
        }
        TreeMap<String, List<Email>> byDomain = new TreeMap<>();
        for (Email e : valid) byDomain.computeIfAbsent(e.domain(), k → new ArrayLis
        for (Map.Entry<String, List<Email>> entry : byDomain.entrySet()) {
            System.out.println(entry.getKey() + " " + entry.getValue().size());
        }
        System.out.println("rejected=" + rejected);
    }
}
```

EXERCISE 2

Records as map keys

Declare `record Key(String city, int year)`. Read `n` lines `<city> <year> <amount>` and total the amounts per `(city, year)` in a `TreeMap<Key, Long>` ordered by city then year (`Comparator.comparing(Key::city).thenComparingInt(Key::year)`). Print `<city> <year>: <total>` per key, then `keys=<count>`. Records make correct, immutable composite keys with no code of your own.

Example: `4` then `oslo 2023 10`, `rome 2023 5`, `oslo 2023 7`, `oslo 2022 1` →

```
oslo 2022: 1
oslo 2023: 17
rome 2023: 5
keys=3
```

INPUT 1

```
4
oslo 2023 10
rome 2023 5
oslo 2023 7
oslo 2022 1
```

OUTPUT 1

```
oslo 2022: 1
oslo 2023: 17
rome 2023: 5
keys=3
```

INPUT 2

```
1
pune 2024 100
```

OUTPUT 2

```
pune 2024: 100
keys=1
```

REFERENCE SOLUTION

```java
import java.util.*;

record Key(String city, int year) { }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        TreeMap<Key, Long> totals = new TreeMap<>(Comparator.comparing(Key::city).th
        for (int i = 0; i < n; i++) {
            Key k = new Key(in.next(), in.nextInt());
            totals.merge(k, in.nextLong(), Long::sum);
        }
        totals.forEach((k, v) -> System.out.println(k.city() + " " + k.year() + ": '
        System.out.println("keys=" + totals.size());
    }
}
```

QUIZ

1. To "change" one component of a record you…

A. Call its setter

B. Construct a new record — typically via a `withX` method

C. Assign the field

D. Use reflection

2. Records make good `HashMap` keys because…

A. They are small

B. They are immutable with value-based generated `equals` / `hashCode`

C. They are final

D. They cache their hash

3. A record declared inside a method (a local record)…

A. Can capture local variables

B. Is implicitly static and visible only in that method

C. Must be public

D. Is not allowed

4. Which should **not** be a record?

A. A `(row, col)` grid coordinate

B. An immutable configuration value

C. A shopping cart whose contents change over time

D. A parse result

5. `record Team(String name, List<String> members)` with `members = List.copyOf(members)` in the compact constructor ensures…

A. Members are sorted

B. The stored list is an immutable copy the caller cannot mutate later

C. Nulls are allowed

D. Faster equality

ANSWER KEY · RECORDS IN PRACTICE — VALIDATION, WITHER METHODS AND MODELLING

1. B Records are immutable; modified copies are new instances.

2. B A `record Cell(int row, int col)` is the two-int key that used to need a hand-written class.

3. B Local records tidy one pipeline's intermediate shape without polluting the class.

4. C Mutable state with identity is a class. Records are for data defined entirely by their components.

5. B Without the copy the record would alias the caller's mutable list — shallow immutability made deep for that component.

CHECKPOINT · 22 MIN

Checkpoint — Enums & records

This checkpoint covers plain enums, enums with state and behaviour, `EnumSet` / `EnumMap`, records, and the practical record patterns.

How it works. Twelve questions and two programs; 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer:

- Why `==` is right for enums, and what `valueOf("friday")` does.
- Why an enum constructor cannot read a static field, and how a `fromCode` lookup is built.
- What `EnumSet` and `EnumMap` are made of and why they beat hash collections.
- Everything a record generates, and the two things it forbids.
- What a compact constructor can do, and why a `List` component should be copied.

The programs are an enum with per-constant behaviour driving a small calculator, and a record-based model with validation and grouping into an `EnumMap`.

EXERCISE 1

Planets

Declare `enum Planet` with `MERCURY(3.303e23, 2.4397e6)`, `EARTH(5.976e24, 6.37814e6)` and `JUPITER(1.9e27, 7.1492e7)` (mass in kg, radius in m), a method `surfaceGravity()` = `G * mass / (radius²)` with `G = 6.67300e-11`, and `surfaceWeight(double otherMass)` = `otherMass * surfaceGravity()`. Read a weight on Earth in newtons; compute the mass as `weight / EARTH.surfaceGravity()` and print, for every planet in declaration order, `<NAME> <weight with 2 decimals>`.

Example: `175` →

```
MERCURY 66.11
EARTH 175.00
JUPITER 442.85
```

INPUT 1

```
175
```

OUTPUT 1

```
MERCURY 66.11
EARTH 175.00
JUPITER 442.85
```

INPUT 2

```
0
```

OUTPUT 2

```
MERCURY 0.00
EARTH 0.00
JUPITER 0.00
```

REFERENCE SOLUTION

```
import java.util.*;

enum Planet {
    MERCURY(3.303e23, 2.4397e6),
    EARTH(5.976e24, 6.37814e6),
    JUPITER(1.9e27, 7.1492e7);

    private static final double G = 6.67300e-11;
    private final double mass;
    private final double radius;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    double surfaceGravity() { return G * mass / (radius * radius); }
    double surfaceWeight(double otherMass) { return otherMass * surfaceGravity(); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        double earthWeight = in.nextDouble();
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values()) {
            System.out.printf("%s %.2f%n", p, p.surfaceWeight(mass));
        }
    }
}
```

EXERCISE 2

Inventory by category

Declare `enum Category { FOOD, TOOLS, TOYS }` and `record Item(String name, Category category, int qty, long priceCents)` whose compact constructor rejects a negative quantity or price with `IllegalArgumentException`, plus `long valueCents()` = qty × price. Read an integer `n` and `n` lines `<name> <CATEGORY> <qty> <priceCents>`. Skip invalid items (count them). Total the value per category in an `EnumMap` and print, in declaration order, `<CATEGORY> <total>` for every category (0 if none), then `invalid=<count>`.

Example: `3` then `apple FOOD 10 50`, `saw TOOLS -1 900`, `ball TOYS 2 300` →

```
FOOD 500
TOOLS 0
TOYS 600
invalid=1
```

INPUT 1

```
3
apple FOOD 10 50
saw TOOLS -1 900
ball TOYS 2 300
```

OUTPUT 1

```
FOOD 500
TOOLS 0
TOYS 600
invalid=1
```

INPUT 2

```
2
x TOOLS 1 1
y TOOLS 2 2
```

OUTPUT 2

```
FOOD 0
TOOLS 5
TOYS 0
invalid=0
```

REFERENCE SOLUTION

```
import java.util.*;

enum Category { FOOD, TOOLS, TOYS }

record Item(String name, Category category, int qty, long priceCents) {
    public Item {
        if (qty < 0 || priceCents < 0) throw new IllegalArgumentException("negative'
    }
    long valueCents() { return qty * priceCents; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        EnumMap<Category, Long> totals = new EnumMap<>(Category.class);
        int invalid = 0;
        for (int i = 0; i < n; i++) {
            String name = in.next();
            Category c = Category.valueOf(in.next());
            int qty = in.nextInt();
            long price = in.nextLong();
            try {
                Item item = new Item(name, c, qty, price);
                totals.merge(c, item.valueCents(), Long::sum);
            } catch (IllegalArgumentException e) {
                invalid++;
            }
        }
        for (Category c : Category.values()) {
            System.out.println(c + " " + totals.getOrDefault(c, 0L));
        }
        System.out.println("invalid=" + invalid);
    }
}
```

EXERCISE 3

A deck of cards

Declare `enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }`, `enum Rank { TWO, THREE, …, TEN, JACK, QUEEN, KING, ACE }` and `record Card(Rank rank, Suit suit)`. Read `n` cards as `<RANK> <SUIT>`. Print them sorted by rank then suit (both by ordinal) as `RANK of SUIT`, then `highest=<the highest card>`, `flush=<all the same suit>`, and `pairs=<how many ranks appear at least twice>` using an `EnumMap<Rank, Integer>`.

Example: `3` then `ACE SPADES`, `TWO HEARTS`, `TWO CLUBS` →

```
TWO of CLUBS
TWO of HEARTS
ACE of SPADES
highest=ACE of SPADES
flush=false
pairs=1
```

INPUT 1

```
3
ACE SPADES
TWO HEARTS
TWO CLUBS
```

OUTPUT 1

```
TWO of CLUBS
TWO of HEARTS
ACE of SPADES
highest=ACE of SPADES
flush=false
pairs=1
```

INPUT 2

```
2
KING HEARTS
TEN HEARTS
```

OUTPUT 2

```
TEN of HEARTS
KING of HEARTS
highest=KING of HEARTS
flush=true
pairs=0
```

REFERENCE SOLUTION

```
import java.util.*;

enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING,
record Card(Rank rank, Suit suit) {
    @Override public String toString() { return rank + " of " + suit; }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Card> hand = new ArrayList<>();
        for (int i = 0; i < n; i++) hand.add(new Card(Rank.valueOf(in.next()), Suit.
        hand.sort(Comparator.comparing(Card::rank).thenComparing(Card::suit));
        for (Card c : hand) System.out.println(c);
        System.out.println("highest=" + hand.get(hand.size() - 1));
        boolean flush = hand.stream().map(Card::suit).distinct().count() == 1;
        System.out.println("flush=" + flush);
        EnumMap<Rank, Integer> counts = new EnumMap<>(Rank.class);
        for (Card c : hand) counts.merge(c.rank(), 1, Integer::sum);
        long pairs = counts.values().stream().filter(v -> v >= 2).count();
        System.out.println("pairs=" + pairs);
    }
}
```

CHECKPOINT QUESTIONS

1. `enum Color { RED, GREEN }` — `Color.RED.ordinal()` and `Color.RED.name()` are…

A. `"RED"` and 0

B. 0 and `"RED"`

C. 1 and `"Red"`

D. 0 and `"red"`

2. Can an enum extend a class?

A. Yes

B. No — it implicitly extends `Enum`; it may implement interfaces

C. Only abstract classes

D. Only in the same package

3. `switch (day)` where `day` is null…

A. Runs `default`

B. Throws `NullPointerException`

C. Matches nothing

D. Is a compile error

4. Per-constant behaviour via constant-specific bodies means each such constant is…

A. A separate enum

B. An anonymous subclass of the enum type

C. Static

D. Final and abstract

5. `EnumMap` versus `HashMap` for enum keys — the *ordering* difference is…

A. None

B. `EnumMap` iterates in declaration order; `HashMap` order is unspecified

C. `HashMap` sorts keys

D. `EnumMap` uses insertion order

6. `EnumSet.complementOf(set)` returns…

A. The same set

B. Every constant of the enum not in `set`

C. An empty set

D. The set sorted

7. Which is true of record components?

A. They are mutable

B. Each becomes a private final field with a same-named public accessor

C. They are static

D. They must be primitives

8. A record's generated `equals` …

A. Compares references

B. Compares every component (primitives by value, references by `equals`)

C. Compares only the first component

D. Is not generated

9. Inside a compact constructor, `this.x = x` is…

A. Required

B. A compile error — assignment to the fields happens implicitly after the body

C. Optional

D. Only allowed for the first component

10. Returning two values from a method is idiomatically done with…

A. An `Object[]`

B. A small record declared for the purpose (possibly local)

C. Two globals

D. An `int` pair packed into a `long`

11. A record can implement `Comparable<Self>` by…

A. Nothing — records cannot implement interfaces

B. Declaring `implements Comparable<Self>` and writing `compareTo`

C. Overriding `equals`

D. Using `ordinal()`

12. Which framework need makes a record unsuitable?

A. JSON serialisation with Jackson

B. A JPA entity requiring a no-arg constructor and setters

C. Use as a map key

D. Use in streams

ANSWER KEY · CHECKPOINT — ENUMS & RECORDS

1. B `ordinal` is the zero-based declaration index; `name` is the exact identifier.

2. B Single inheritance is already used by `java.lang.Enum`.

3. B The switch dispatches on `ordinal()`; guard for null before switching (or use Java 21's `case null`).

4. B The enum is then not final; `getClass()` of such a constant differs from `getDeclaringClass()`.

5. B The backing array is indexed by ordinal, giving a natural, stable order.

6. B A bit-flip over the enum's full range — `allOf` minus the given set.

7. B `record P(int x)` gives `private final int x` and `public int x()`.

8. B Value semantics over all components, consistent with the generated `hashCode`.

9. B You may reassign the *parameter* `x`; the field assignment is generated.

10. B Named, typed, one line — and structurally equal for tests.

11. B Records implement interfaces like any class; accessors with matching names satisfy interface methods automatically.

12. B Records have neither; use them for DTOs crossing the boundary and a class for the managed entity.

CodeKairo · Java study plan · Module 10: Enums & records. Solve the exercises online to track your progress and earn XP.