

Java

Лекція 2

Зміст

- Switch expression
- Enum
- Record
- Equals, hashCode

Switch-case

```
int day = 3;  
String dayName;  
  
switch(day) {  
    case 1:  
        dayName = "Monday";  
        break;  
    case 2:  
        dayName = "Tuesday";  
        break;  
    case 3:  
        dayName = "Wednesday";  
        break;  
    default:  
        dayName = "Other";  
}  
  
System.out.println(dayName);
```

Потрібні break, щоб уникнути “fall-through”.
Не повертає значення напряму.

```
int day = 3;  
  
switch (day) {  
    case 1:  
        System.out.println("Monday: Start work week!");  
        break;  
    case 2:  
        System.out.println("Tuesday: Team meeting");  
        break;  
    case 3:  
        System.out.println("Wednesday: Work from home");  
        break;  
    case 4:  
        System.out.println("Thursday: Project review");  
        break;  
    case 5:  
        System.out.println("Friday: Wrap up tasks");  
        break;  
    case 6:  
    case 7:  
        System.out.println("Weekend: Relax!");  
        break;  
    default:  
        System.out.println("Invalid day");  
}
```

Недоліки switch-case

Недоліки старого `switch` для присвоєння значень

Потрібно оголошувати змінну заздалегідь:

```
String dayName;  
  
switch(day) {  
    case 1: dayName = "Monday";  
    break;
```

```
        case 2: dayName = "Tuesday";  
        break;  
  
        default: dayName = "Other";  
    }  
  
    /
```

Необхідні `break` → легко забути → `fall-through`.

Код стає громіздким, особливо коли потрібно багато кейсів.

Switch expression

Використовується як вираз (*expression*), а не лише як оператор.

Менше *boilerplate* → немає *break*, компактно і безпечно.

Кожен *case* повертає значення

```
int day = 3;  
  
String dayName = switch(day) {  
    case 1 -> "Monday";  
    case 2 -> "Tuesday";  
    case 3 -> "Wednesday";  
    default -> "Other";  
};  
  
System.out.println(dayName);
```

Switch expression

Якщо потрібно виконати кілька дій і повернути значення, використовується блок з `yield`. `yield` повертає значення з блока.

```
int day = 3;

String message = switch (day) {
    case 1 -> {
        System.out.println("Monday: Start work week!");
        yield "Monday";
    }
    case 2 -> {
        System.out.println("Tuesday: Team meeting");
        yield "Tuesday";
    }
    case 3 -> {
        System.out.println("Wednesday: Work from home");
        yield "Wednesday";
    }
}
```

```
}

case 4 -> {
    System.out.println("Thursday: Project review");
    yield "Thursday";
}

case 5 -> {
    System.out.println("Friday: Wrap up tasks");
    yield "Friday";
}

case 6, 7 -> {
    System.out.println("Weekend: Relax!");
    yield "Weekend";
}

default -> {
    System.out.println("Invalid day");
    yield "Invalid";
}

};

System.out.println("Returned value: " + message);
```

Switch-case vs Switch expression

Старий switch (оператор)

✓ Використовувати, коли:

Потрібно виконати дії / побічні ефекти (наприклад, System.out.println, виклик методів).

Код логічно підходить до стилю "вибір і виконання дій".

Важливий сумісний стиль коду з legacy-проектами (до Java 14).

✗ Недоліки:

Потребує break, інакше → fall-through.

Не можна напряму використати у виразах (наприклад, для присвоєння).

Switch expression (з Java 14+)

✓ Використовувати, коли:

Потрібно обчислити значення (присвоєння, return).

Хочеться писати компактно та безпечніше (без break).

Кейсів багато, і код має бути expression-oriented.

```
String dayName = switch(day) {  
    case 1 -> "Monday";  
    case 2 -> "Tuesday";  
    default -> "Other";  
};
```

✗ Не завжди зручно для чистих дій без значення → тоді старий switch виглядає простіше.

Pattern Matching y Switch (Java 17+)

```
static String format(Object obj) {  
  
    if (obj instanceof String) {  
  
        String s = (String) obj;  
  
        return "String of length " + s.length();  
  
    } if (obj instanceof Integer) {  
  
        Integer i = (Integer) obj;  
  
        return "Integer value " + i;  
  
    }  
  
    return "Unknown type";  
}
```

```
static String format(Object obj) {  
  
    return switch (obj) {  
  
        case String s -> "String of length " + s.length();  
  
        case Integer i -> "Integer value " + i;  
  
        case null -> "Null value";  
  
        default -> "Unknown type";  
    };  
}
```

var та text blocks

◆ var (Java 10)

Використовується для локальних змінних.

Тип визначається автоматично компілятором.

Зменшує "шум" у коді, але не робить Java динамічною.

```
var name = "Alice";      // String  
var age = 25;            // int  
var list = new ArrayList<String>();
```

✓ Зручно, коли тип і так зрозумілий з правої частини.

✗ Не можна для полів класів чи параметрів методів.

◆ Багаторядкові рядки (Text Blocks, Java 15)

Огортаються в потрійні лапки """.

Автоматично зберігають форматування.

Зручні для SQL, HTML, JSON тощо.

```
String json = """  
{  
    "name": "Alice",  
    "age": 25  
};  
""";
```

✓ Краще читається, ніж конкатенація "... + ...".

✓ Підтримка інтерполяції через `String::formatted` (Java 15).

“Привіт, ” + ім'я

Форматування рядків

Feature	Класичне <code>String.format</code> (Java 1.5+)	<code>String::formatted</code> (Java 15+)	<code>String Templates</code> (Java 21+, preview)
Синтаксис	<code>String.format("Привіт, %s", name)</code>	<code>"Привіт, %s".formatted(name)</code>	<code>STR."Привіт, \${name}"</code>
Вирази	Ні, тільки плейсхолдери	Ні, тільки плейсхолдери	Так, будь-які вирази всередині <code>\${}</code>
Типобезпека	Немає — помилки у типах виявляються тільки під час виконання	Немає — помилки під час виконання	Є — компілятор перевіряє типи
Зручність для багаторядкових рядків	Менш зручний, доводиться додавати <code>\n</code> та <code>+</code>	Краще, особливо з text blocks	Найзручніше, нагадує шаблони в інших мовах
Приклад	<code>String.format("Привіт, %s! Тобі %d років.", name, age)</code>	<code>"Привіт, %s! Тобі %d років.".formatted(name, age)</code>	<code>STR."Привіт, \${name}! Тобі \${age} років."</code>

Енім

- **Енім (enumeration)** – це спеціальний тип даних, який представляє обмежену множину констант.
- Використовується для кращої читабельності та безпеки коду.
- Замість чисел або рядків:
- **Енім** може містити:
 - Поля
 - Конструктори
 - Методи
- Може використовуватися у `switch` як значення, що перевіряється

Екзим. Приклад

```
enum Day {  
    MONDAY("Початок тижня"),  
    TUESDAY("Другий день тижня"),  
    WEDNESDAY("Середина тижня"),  
    THURSDAY("Четвер"),  
    FRIDAY("Кінець робочого тижня"),  
    SATURDAY("Вихідний"),  
    SUNDAY("Вихідний");  
  
    // Поле для опису  
    private final String description;  
  
    // Конструктор Enum  
    Day(String description) {  
        this.description = description;  
    }  
  
    // Метод для отримання опису  
    public String getDescription() {  
        return description;  
    }  
}
```

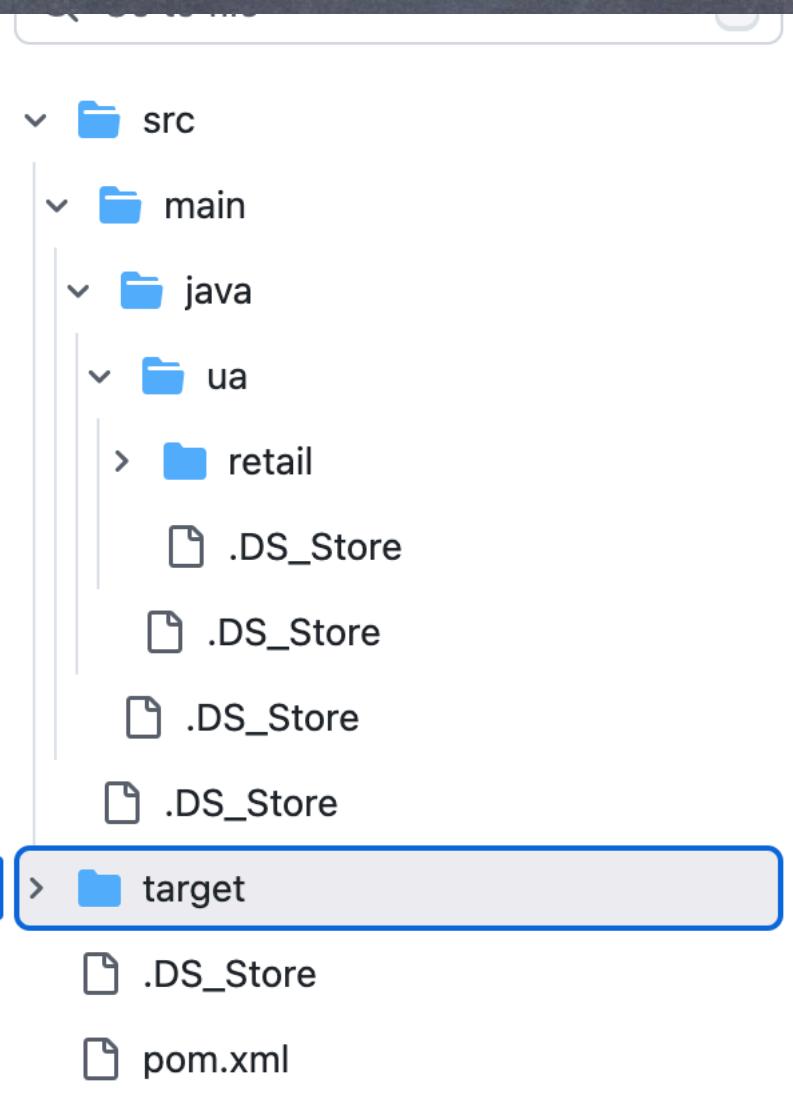
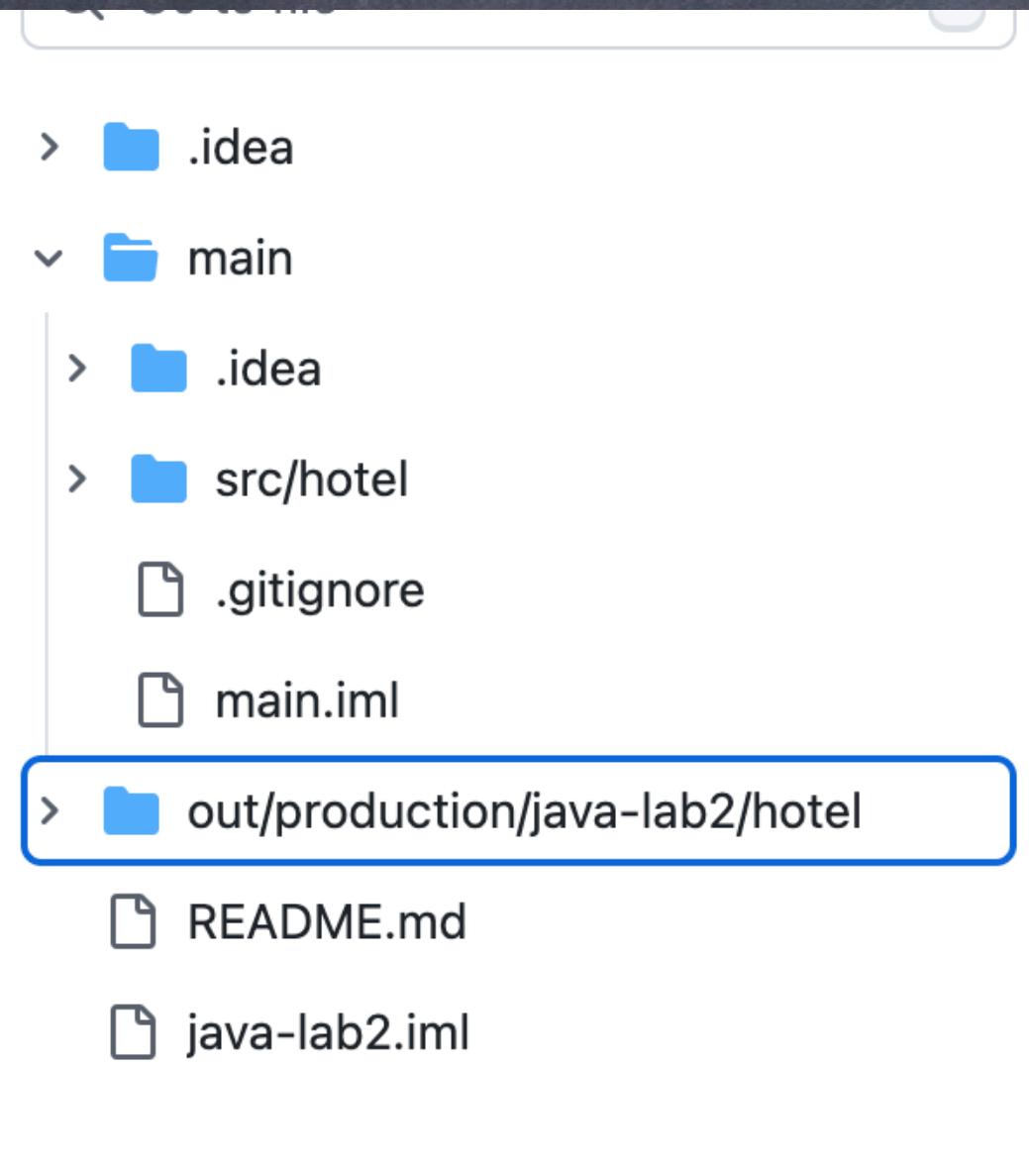
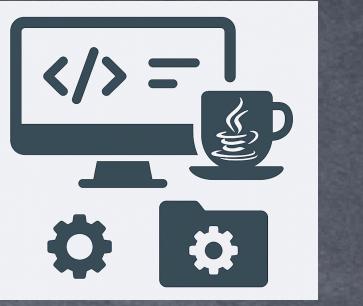
```
public class EnumDemo {  
    public static void main(String[] args) {  
        Day today = Day.WEDNESDAY;  
  
        System.out.println("Сьогодні: " + today);  
        System.out.println("Опис: " + today.getDescription());  
  
        System.out.println("\nУсі дні тижня:");  
        for (Day d : Day.values()) {  
            System.out.println(d + " - " + d.getDescription());  
        }  
  
        switch (today) {  
            case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY ->  
                System.out.println("\nРобочий день");  
            case SATURDAY, SUNDAY ->  
                System.out.println("\nВихідний день");  
        }  
  
        Day d = Day.valueOf("FRIDAY");  
        System.out.println("\nМетод valueOf: " + d);  
        System.out.println("Позиція у Enum (ordinal): " + d.ordinal());  
    }  
}
```

Аналогічний клас до Єніон

```
public class DayClass {  
    // Статичні поля (імітація констант Enum)  
    public static final DayClass MONDAY = new DayClass("MONDAY", "Початок  
тижня");  
    public static final DayClass TUESDAY = new DayClass("TUESDAY", "Другий  
день тижня");  
    public static final DayClass WEDNESDAY = new DayClass("WEDNESDAY",  
"Середина тижня");  
    public static final DayClass THURSDAY = new DayClass("THURSDAY",  
"Четвер");  
    public static final DayClass FRIDAY = new DayClass("FRIDAY", "Кінець  
робочого тижня");  
    public static final DayClass SATURDAY = new DayClass("SATURDAY",  
"Вихідний");  
    public static final DayClass SUNDAY = new DayClass("SUNDAY", "Вихідний");  
  
    // Поля для назви та опису  
    private final String name;  
    private final String description;  
  
    // Приватний конструктор  
    private DayClass(String name, String description) {  
        this.name = name;  
        this.description = description;  
    }  
  
    // Геттери  
    public String getName() {  
        return name;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
    // Метод для зручного виводу  
    @Override  
    public String toString() {  
        return name + " (" + description + ")";  
    }  
  
    // Масив всіх значень (імітація values())  
    public static DayClass[] values() {  
        return new DayClass[]{MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
                           FRIDAY, SATURDAY, SUNDAY};  
    }  
}
```

Переваги використання **Енім**

- ⦿ Зрозумілість коду
- ⦿ Безпечность типів
- ⦿ Легко розширювати (поля, методи)
- ⦿ Підходить для `switch` та колекцій



Ignore build directories
out/
target/

Ignore macOS system file
.DS_Store

Record (Java 16+)

- Це спеціальний клас у Java (Java 16+), який використовується для зберігання даних.
- Основна ідея: менше шаблонного коду для класів, що містять лише поля та геттери.
- Record автоматично генерує:
 - Поля `final` (значення не можна змінювати після створення об'єкта)
 - Конструктор з усіма полями
 - Методи `equals()`, `hashCode()`, `toString()`
- Може містити свої методи

Record. Приклад

```
public record Person(String surname, String name,  
LocalDate birthDate) {  
  
    public Person {  
        if (surname == null || surname.isBlank()) {  
            throw new IllegalArgumentException("Surname  
cannot be empty");  
        }  
        if (name == null || name.isBlank()) {  
            throw new IllegalArgumentException("Name  
cannot be empty");  
        }  
        if (birthDate == null ||  
birthDate.isAfter(LocalDate.now())) {  
            throw new IllegalArgumentException("Birth date  
is invalid");  
        }  
    }  
}
```

```
public Person(String surname, String name) {  
    this(surname, name, LocalDate.now());  
}
```

```
public boolean isAdult() {  
    return Period.between(birthDate,  
LocalDate.now()).getYears() >= 18;  
}
```

```
Person p = new Person("Іваненко", "Олександр",  
LocalDate.of(2005, 5, 15));  
System.out.println(p); //  
Person[surname=Іваненко, name=Олександр,  
birthDate=2005-05-15]  
System.out.println("Повнолітній? " +  
(p.isAdult() ? "Так" : "Ні"));
```

Переваги та обмеження

Record

✓ Переваги

- Менше шаблонного коду
(автоматично генерує конструктор, геттери, equals, hashCode, toString)
- Immutable (поля final) → безпечний для багатопоточності
- Ідеально підходить для DTO / value objects
- Чітка структура даних і типобезпека

! Обмеження

- Поля final → значення не можна змінювати після створення
- Не можна успадковувати Record від інших класів (Record extends не допускається)
- Не можна створювати пустий конструктор без полів (обов'язково треба вказати всі компоненти)
- Обмежена гнучкість порівняно зі звичайними класами (наприклад, складна логіка у конструкторах)

Object

Метод	Призначення	Приклади використання / перевизначення
<code>toString()</code>	Повертає рядкове представлення об'єкта	Для зрозумілого виводу, логування
<code>equals(Object obj)</code>	Перевірка рівності об'єктів	Для порівняння об'єктів за значенням, а не посиланням
<code>hashCode()</code>	Повертає числовий хеш об'єкта	Разом з <code>equals()</code> для колекцій HashMap, HashSet
<code>clone()</code>	Створює копію об'єкта	Іноді для створення копій об'єктів (часто замінюють copy constructor)
<code>getClass()</code>	Повертає Class об'єкта	Використовується для reflection, порівняння типів

Правила для equals та hashCode

1 Правила для equals()

- Симетричність: `a.equals(b)` має давати той самий результат, що і `b.equals(a)`.
- Рефлексивність: `a.equals(a)` завжди має бути `true`.
- Транзитивність: якщо `a.equals(b) і b.equals(c) → a.equals(c)` теж має бути `true`.
- Консистентність: багаторазові виклики `a.equals(b)` без зміни об'єктів дають один і той самий результат.
- null-check: `a.equals(null)` має повертати `false`.
- Використовувати ті поля, які визначають “логічну рівність”:
 - Краще `final` поля, які рідко змінюються

- Не включати колекції, які часто змінюються, бо це може порушити консистентність

2 Правила для hashCode()

- Якщо `a.equals(b) → hashCode(a) == hashCode(b)`.
- Якщо `a.equals(b)` повертає `false` → hash-коди можуть бути однакові, але бажано різні для розподілу у хеш-структурах.
- Використовувати ті ж поля, що й у `equals()`.
- Не включати mutable поля, які часто змінюються, щоб `hashCode` не змінювався після додавання в колекції типу `HashMap` або `HashSet`.

Приклад

```
public class Person {  
    private final String surname;  
    private final String name;  
    private final LocalDate birthDate;  
    private List<String> courses;  
  
    public Person(String surname, String name, LocalDate birthDate,  
List<String> courses) {  
        this.surname = Objects.requireNonNull(surname);  
        this.name = Objects.requireNonNull(name);  
        this.birthDate = Objects.requireNonNull(birthDate);  
        this.courses = courses; // змінюване поле  
    }  
  
    public String getSurname() { return surname; }  
    public String getName() { return name; }  
    public LocalDate getBirthDate() { return birthDate; }  
    public List<String> getCourses() { return courses; }  
  
    @Override
```

```
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Person person)) return false;  
        return surname.equals(person.surname) &&  
               name.equals(person.name) &&  
               birthDate.equals(person.birthDate);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(surname, name, birthDate);  
    }  
  
    @Override  
    public String toString() {  
        return surname + " " + name + ", народився: " + birthDate +  
               ", курси: " + courses;  
    }
```

Підсумок

- `var` - спрощене оголошення локальних змінних
- ◆ `Switch expression` - компактний код із можливістю повернати значення
- `Enum` - безпечні перелічувані константи з полями та методами
- `Record` - `immutable` клас для даних із автоматичними `constructor`,
`equals()`, `hashCode()`, `toString()`
- ⚡ `equals()` & `hashCode()` - перевизначаємо для коректної роботи колекцій,
враховуючи стабільні поля
- `toString()` - зрозуміле рядкове представлення об'єктів