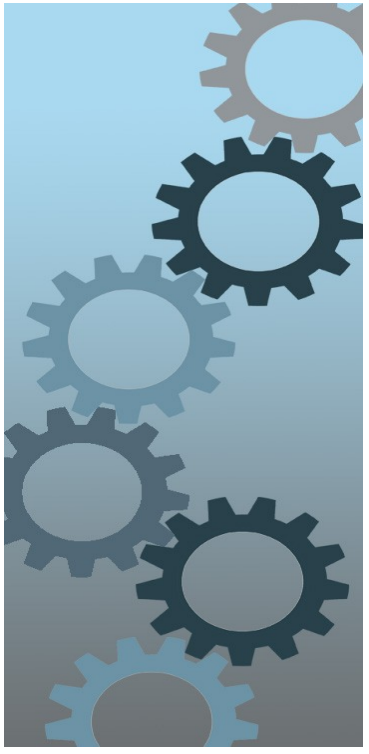
HSLU Hochschule Luzern

Programming Concepts And Paradigms

Algebraic Data Types
Records and Sealed Types



Marcel Baumann

Content

- Introduction: Conciseness, Immutability
- Algebraic Data Types
  - Records
  - Sealed Types
- Outlook Valhalla and Amber

Conciseness

- Modern approach when writing *Java* code
  - Diamond operator
  - Var declaration
  - Lambda instead of anonymous class
  - Annotation versus XML – or JSON, YAML, TOML – configuration
- Please write concise code
  - And use helper methods like *Stream.toList()*

Immutability

- Modern approach when writing concurrent code
  - Byte, Short, Integer, Long, Float, Double, Character, Boolean
  - String
  - BigDecimal, BigInteger
  - LocalDate, LocalTime, LocalDateAndTime, ZoneDateTime, ZoneId, Instant, Period, Duration
  - Enums, Optional<T>
  - Locale, UUID, URI, URL, etc.

Why Immutability?

- **safety**: can be sure that no one is able to change their state
- **thread-safety**: the same is also guaranteed in a multithreaded environment
- **cacheable**: instances could easily be cached by VM or a custom implementation
- **hashable**: such classes could be safely put inside the hash collections (like HashMap, HashSet, etc.)
  - hashCode(), equals() *have a contract in Java as documented in Java API*
  - Comparable*(),* Comparator*() have also contracts*

Algebraic Data Types

- Product Types - Cartesian Product
  - Tuples (not available in Java)
  - Records → *Immutable Record (shallow immutability)*
- Disjoint Types - Disjoint Unions
  - Enumerations → (Immutable) *Enum*
  - Sealed Types

Algebraic Data Types

- Enums *JDK 5*
- Records *JDK 16 (after two preview JDKs)*
- Sealed Types JDK 17 *(after two preview JDKs)*
  - *JDK 17 is an official LTS release released in September 2021*

Records

```
record Person(String firstname, String lastname)
{}
```

Records

- Are immutable
- Provides private final fields
- Provides getters → *new notation (and an improved one!)*
- Provides constructor – *constructor with all fields as parameters*
- Provides equals() and hashCode() implementations
- Provides toString() implementation
- *Construction is initialization (as e.g. in C++)*

Records

Should always be valid: *You should validate your record in the compact canonical constructor*

```
Person {
  Objects.requireNonNull(firstname);
  Objects.requireNonNull(lastname);
}
```

Records Details

- You can define additional constructors
- You can define additional instance and static methods
- You can overwrite provided methods
- You **cannot** add additional instance variables or change the declaration of existing ones
  - Instance variables must be **private final**
  - A record is always a **final** class

Records Code

```
record Person(String firstname, String lastname) {
    Person {
        Objects.requireNonNull(firstname);
        Objects.requireNonNull(lastname);
    }
    public Person(String lastname) {
        this("", lastname);
    }
    public String text() {
        return lastname + ", " + firstname;
    }
}
```

Record Advantages

- Concise
- Immutable (shallow immutability)
- Secure serialization
- Should always be valid
- Record implicit contract (see JavaDoc)

```
R copy = new R(r.c1(), r.c2(), ..., r.cn())
    → r.equals(copy)
```

Multiple Return Values Example

```
public record MinMax(int min, int max);


public MinMax minmax(int[] elements) {
    ...
    return new MinMax(minimum, maximum);
}
```

Record Disadvantages

- Missing constructor with builder features
  – with(…*)*
  see LocalDate or LocalTime for withXXX examples
  (Kotlin also provides a similar solution)

Control Questions

1. Is it possible to set a property for a record e.g.
   ```
   void name(String name) {
       this.name = name;
   }
   ```
2. Is it possible to define a constructor for a record? How?
3. Is it possible to define additional methods for a record?
4. How can you insure that the default constructor parameters fulfill preconditions?

Sealed Types

- Sealed types purpose is a class hierarchy modeling various possibilities that exist in a domain
- Sealed types must be in same named module or same package part of an unnamed module
- *Simplify programmatic visitor pattern*
- *Communicates better your design*

Constraints

1) The sealed class and its permitted subclasses must belong to the **same module**, and, if declared in an unnamed module, the same package
2) Every permitted subclass must **directly** extend the sealed class
3) Every permitted subclass must choose a modifier to describe how it continues the sealing initiated by its superclass:
    1) A permitted subclass may be declared **final** to prevent its part of the class hierarchy from being extended further
    2) A permitted subclass may be declared **sealed** to allow its part of the hierarchy to be extended further than envisaged by its sealed superclass, but in a restricted fashion
    3) A permitted subclass may be declared **non-sealed** so that its part of the hierarchy reverts to being open for extension by unknown subclasses. (A sealed class cannot prevent its permitted subclasses from doing this.)

Remarks

- One and only one of the modifiers **final**, **sealed**, and **non-sealed** must be used by each permitted subclass
- All involved classes are the same module

Sealed Interfaces

```
public sealed interface Service permits Car, Truck {
    int getMaxServiceIntervalInMonths();
    default int
        getMaxDistanceBetweenServicesInKilometers() {
            return 100000;
    }
}
```

Sealed Classes

```
public abstract sealed class Vehicle permits Car, Truck {
    protected final String registrationNumber;

    public Vehicle(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public String getRegistrationNumber() {
        return registrationNumber;
    }
}
```

Subclasses

```
public final class Truck extends Vehicle implements Service {
    private final int loadCapacity;

    public Truck(int loadCapacity, String registrationNumber) {
        super(registrationNumber);
        this.loadCapacity = loadCapacity;
    }

    public int getLoadCapacity() {
        return loadCapacity;
    }

    @Override
    public int getMaxServiceIntervalInMonths() {
        return 18;
    }
}
```

UnSealed Subclasses

```
public non-sealed class Car extends Vehicle implements Service {
    private final int numberOfSeats;

    public Car(int numberOfSeats, String registrationNumber) {
        super(registrationNumber);
        this.numberOfSeats = numberOfSeats;
    }

    public int getNumberOfSeats() {
        return numberOfSeats;
    }

    @Override
    public int getMaxServiceIntervalInMonths() {
        return 12;
    }
}
```

Sealed Type Advantages

- Pattern matching
  - Switch expression without default
  - Compiler error if you extend your sealed hierarchy and forget to expand your switch statements
- Programmatic control of subclasses

Sealed Type Disadvantages

- Mainstream only in JDK 17 with *JEP 409*
- Pattern matching in switch
  - *JEP 406 for JDK-17 is be first step*
  - *JEP 420 for JDK-18 is next step*
  - *Candidate JEP 405 is future step*

Records and Sealed Types

```
sealed interface Expr permits … { }
record ConstantExpr(int i) implements Expr { }
record PlusExpr(Expr a, Expr b) implements Expr { }
record TimesExpr(Expr a, Expr b) implements Expr { }
record NegExpr(Expr e) implements Expr { }
```

Deconstruction

```
int eval(Expr e) {
    return switch (e) {
        case ConstantExpr(var i) -> i;
        case PlusExpr(var a, var b) -> eval(a) + eval(b);
        case TimesExpr(var a, var b) -> eval(a) * eval(b);
        case NegExpr(var e) -> -eval(e);
        // no default needed, Expr is sealed
    }
}
```

Control Questions

1. Why should you use record as sealed classes implementation?
2. Why should you NOT use record as sealed classes implementation?
3. Do sealed classes have restrictions how they are packaged and delivered?

Switch Deconstruction (JEP 420)

```
String formatterPatternSwitch(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

Switch Deconstruction (JEP 420)

```
static void testFooBar(String s) {
    switch (s) {
        case null         -> System.out.println("Oops");
        case "Foo", "Bar" -> System.out.println("Great");
        default           -> System.out.println("Ok");
    }
}
```

Switch Deconstruction (JEP 420)

```
void testTriangle(Shape s) {
    switch (s) {
        case Triangle t && (t.calculateArea() > 100) ->
            System.out.println("Large triangle");
        case Triangle t ->
            System.out.println("Small triangle");
        default ->
            System.out.println("Non-triangle");
    }
}
```

Switch Deconstruction (JEP 420)

- Dominance of pattern labels
  - Dominance is a compile error
- Exhaustiveness of switch statements and expressions
  - Incompleteness is a compile error
- Scope of pattern variable declarations
  - Block or statement after declaration, no fall through

Record Deconstruction (JEP 405)

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

void printSum(Object o) {
    if (o instanceof Point(int x, int y)) {
        System.out.println(x+y);
    }
}
```

Control Questions

1. What happens if no null case is specified?
2. Can the compiler checks if a switch statement or expression is exhaustive?
3. What is a dominant case branch?

Outlook Valhalla

- Value Types
  - Evolution of processor architecture
  - Memory management
  - Potential primitive records (deep immutability is needed)
- Reference Types versus Value Types

Outlook Valhalla

- No constructors for value classes
  - LocalDate.of(2020, Month.OCTOBER, 20);
- JEP 390: Warning for value based classes
  - Remove **new** constructor for all primitive wrapper classes
  - Implemented in JDK 16

Outlook Amber

- Switch statements
- Deconstruction
- Construction

Links

- Open JDK
- JEP List
- Project Valhalla
- Project Amber
- Project Coin

Exercises

- Immutable List (similar to Scheme version)
  - Including providing a stream() operation
- Expression with sealed classes