Data-Oriented Programming in Java

Igor Serov

PRODYNA SE

igor.serov@prodyna.com

International Symposium of Java
November 17, 2024

Presentation Overview

1. DOP
2. Model data immutably and transparently
3. Model Data, the Whole Data, and Nothing but the Data
4. Model data immutably and transparently
5. Separate Operations From Data
6. Summary

Why DOP

- A programming paradigm focused on reducing system complexity
- Treats data as a first-class citizen
- Separates data representation from operations
- Alternative to traditional Object-Oriented Programming

Comparison with OOP

OOP	DOP
Everything is an object	Everything is data
Combines state and behavior	Separates state from behavior
Mutable state common	Immutable data preferred
Class-based hierarchy	Data-centric structures

DOP 4 Principles

Model data immutably and transparently

Model the data, the whole data, and nothing but the data.

Make illegal states unrepresentable.

Separate operations from data.

1. Model data immutably and transparently

- **Immutability**
  - guarantees **correctness** by preventing changes to objects
  - **eliminating risks from subsystems modifying** shared objects
- **Transparency**
  - There is an **access method for each field** that returns the same or an equal value.
  - There is a **constructor that accepts values for all fields** and saves them (directly or as a copy) if valid.

Records

```
record Book(String title, ISBN isbn, List<Author> authors
    ) { }
```

- field for each component
- fields must be final
- a canonical constructor that accepts and assigns exactly these values
- accessor methods that return them
- type must be final
- The equals and hashCode methods are based on this data

Records Java 16

Record fields are final, but that doesn’t magically apply to what they reference:

```
record Book(String title, ISBN isbn, List<Author> authors
    ) {
    Book {
        authors = List.copyOf(authors);
        // create a copy, so references to the ‘isbn‘ can
            ’t change the record’s internal state
        isbn = new ISBN(isbn);
    }
    @Override
    public ISBN isbn() {
        // don’t expose mutable inner state
        return new ISBN(isbn);
    }
}
```

2. Model Data, the Whole Data, and Nothing but the Data

- **Records should model data**.
- Each record model **one thing**
- Make a **clear separation** what each record models
- Choose **clear names** for its components
- Multiple choices: **model each alternative with a record**

Sealed Classes Java 17

- Records make it easy to aggregate data
- Sealed types make it easy to express alternatives
- Together allow modeling even complicated structures
- Sealed types useful when the system cannot be expected to simply work when a new implementation is added

```
sealed interface Item permits Book, Furniture,
   ElectronicItem {
    // ...
}
```

Sealed Classes Java

- Allowed **subtypes must be in the same modul**e as the sealed type.
- If the sealed type and permitted subtypes are contained in the same source code file, the permits clause can be omitted.
- Allowed **subtypes must inherit directl**y from the sealed type.
- Allowed subtypes must be **final, sealed or explicitly non-sealed**.

Sealed Classes

- Traditional interfaces define behavior (what a type does)
- Sealed interfaces with records instead:
  - Define categorization
  - Group related data variants
  - Establish a fixed set of alternatives
  - Focus on data organization, not behavior
  - Require no method definitions
  - Create clear boundaries for possible data variants
  - Serve as pure grouping mechanisms

This structure makes it ideal for modeling distinct but related data structures while maintaining clear categorization.

Best Practices to Record's Methods

- **Methods without parameters**: can't do anything other than return the record's data.
  For example: get email top level domain: email.tld().
- **Methods that accept the type itself**
  For example: Book has a method *commonAuthors(Book()*.
- **Methods that accept other records**: No states are changed and all results are communicated via the return value.
- **Avoid implementing non-trivial domain logic**: better move to external systems.
- **Avoid mutable parameters**: blurs this boundary and should be avoided.

3. Make Illegal States Unrepresentable

Focus on only legal combinations of the data for the system

- **Precisely modeled types** : describe the data (usually records).
- **sealed interface for modeling the alternatives**: avoid exclusive or conditional requirements.
- For other cases use *Compact Constructor*.

Make Illegal States Unrepresentable

Modeling Variants

```
sealed interface User permits UnregisteredUser,
   RegisteredUser { }
record UnregisteredUser(/*...*/) { }
record RegisteredUser(/*...*/, Email email) {
// constructor enforces presence of ‘email‘
}
```

Validate at the Boundary

```
record Book(String title, ..., List<Author> authors) {
Book {
    Objects.requireNonNull(title);
    if (title.isBlank())
        throw new IllegalArgumentException("Title must
           not be blank");
    Objects.requireNonNull(authors);
    if (authors.isEmpty())
        throw new IllegalArgumentException("There must be
            at least one author");
    }
}
```

Separate Operations From Data

- Records should avoid non-trivial domain logic
- Operations belong in separate classes, not records
- Example: Shopping Cart Implementation return new Cart
  - Avoid: `Item.addToCart(Cart)`
  - Avoid: `Cart.add(Item)`
  - Use: `Orders.add(Cart, Item)`
- State changes restricted to responsible subsystems

Pattern Matching Java 21

- It can be used as an expression, for example to assign a value to a variable with var foo = switch ....
- If a case label is followed by **->** (instead of a **:**), there is no fall-through.
- The selector expression can have any type.

```
public ShipmentInfo ship(Item item) {
    return switch (item) {
        case Book book -> // use ‘book’
        case Furniture furniture -> // use ‘furniture’
        case ElectronicItem eItem -> // use ‘eItem’
    }
}
```

Record Patterns with deconstruction

```
switch(item) {
    //deconstruct record
    case Book(String title, ISBN isbn, List<
        Author> authors) -> // use ‘title’, ‘isbn
        ’, and ‘authors’
    // more cases...
}
```

Unnamed Patterns Java 22

```
switch(item) {
    case Book(_, ISBN isbn, _) -> // use ‘isbn’
    // more cases...
}
```

```
switch(item) {
    case Book book -> // use ‘book’
    case Furniture _ -> // no additional variable in
        scope
    // more cases...
}
```

Nested Patterns Java 21

```
switch(item) {
    case Book(_, ISBN(String isbn), _) -> // use
        ‘isbn‘
    // more cases...
}
```

Guarded Patterns Java 21

```
    switch(item) {
        case Book(String title, _, _) when title.length()
            > 30 -> // handle long title
        case Book(String title, _, _) -> // handle
           regular title
        // more cases...
    }
```

Requirements

- switching over a sealed interface
- listing all implementations (do not use default)

```
    switch(item) {
        case Book book -> createTableOfContents(book);
        case Furniture _ -> { }
        case ElectronicItem _ -> { }
    }
```

Summary

- **Use types to represent data:**
  - Model data transparently and immutably (usually with **records**).
  - Model alternatives with **sealed interfaces**.
  - Model the data as closely as possible and only represent legal states.
- **Implement operations as methods on other classes:**
  - Use **exhaustive switch** statements, predominantly over sealed interfaces and without a default branch.
  - Use **pattern matching** to identify and decompose data.

The End

Questions?