Java Data Oriented Programming

What Will You Learn?

- Understand the principles of Data Oriented Programming (DOP) and how they reshape modern application design.
- Model business domains using clear, expressive, and intention-driven data structures.
- Improve communication between services through well-defined and meaningful data types.
- Handle uncertainty and failure with more nuance than rigid success/failure responses
- Embrace a mindset shift. From code that reacts to code that communicates.

Note

- DOP is about solving business problems through clear data modeling and writing code that is easy to read, reason about, and maintain.
- It is NOT primarily focused on performance or scalability. But on correctness, clarity, and intent.

Syllabus

- Records
- Sealed Types
- Pattern Matching
- Data Oriented Programming
- Domain Modeling
- Handling Uncertainty
- Error Handling
- DOP: Deserialization Challenges In Microservices
- Application Development

Records

Summary - Records

- Record classes
  - data carriers.
  - implicitly final fields
  - final classes - can not be extended
  - can not extend any other class / record
  - can implement interfaces
  - auto-generated methods
    - equals
    - hashCode
    - toString
    - accessor for every component

- We can not define any instance field.
- We can add custom instance methods.
- Records can have static fields / methods.

Summary - Record Equality!

- Two instances of a record are equal **<u>only if</u>** all of their components are equal.

```
// pseudo
record SomeRecord(rc1,
                  rc2,
                  rc3
                  ...) {
}


this.rc1.equals(that.rc1) &&
this.rc2.equals(that.rc2) &&
this.rc3.equals(that.rc3)
```

Summary - Record Immutability!

- A record is truly immutable **only if** all of its components are immutable.

```
record Person(String firstName,
              String lastName){
}
```

Summary - Constructor

- Records have auto-generated **canonical** constructor. We do NOT need to create this!
- For any input validation / processing, use ***compact*** constructor.
- For any optional inputs, use **non-canonical** constructor.

```java
record Person(String firstName,
              String lastName){

    Person {
        lastName = lastName.toUpperCase();  // do some processing
    }

}
```

```java
record Person(String firstName,
              String lastName){

    // auto-generated. you do not have to write this yourself.
    Person(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

}
```

```java
record Task(String title,
            LocalDate createdAt){

    Task(String title){
        this(title, LocalDate.now());
    }

}
```

Summary - Option 1: Separate Files

- model
  - Address.java
  - User.java
  - Business.java
  - …
  - …

Best for reusable data structures shared across modules or services.

Summary - Option 2: Feature/Domain Oriented Classes

```java
public class BillingDomain {

    public record Address(String street,
                          String city,
                          String zip,
                          String taxId) {
    }

    public record Invoice(String invoiceId,
                         Address address) {
    }

}
```

var invoice = new BillingDomain.Invoice(...)

Helps structure code around bounded contexts or microservice domains.

```java
public class ShippingDomain {

    public record Address(String street,
                          String city,
                          String zip,
                          String deliveryInstructions) {
    }

    public record Shipment(String trackingId,
                           Address address) {
    }

}
```

Summary - Option 3: Nested Records - Tightly Coupled

```java
public record MovieResponse(Movie movie) {

    public record Movie(String title,
                        int year,
                        Director director,
                        Rating rating) {
    }

    public record Director(String name,
                           String country) {
    }

    public record Rating(String source,
                         double score) {
    }

}
```

Ideal for structured API responses or encapsulated models.

Summary - Option 4: Private Inner Records (Helper)

```java
public class LoanEligibilityService {

    public String evaluateEligibility(String userId) {
        EligibilityData data = gatherEligibilityData(userId);

        if (data.creditScore() >= 750 && data.verifiedIncome() > 50000) {
            return "Approved";
        }
        else {
            return "Denied";
        }

    }

    private EligibilityData gatherEligibilityData(String userId) {
        int creditScore = fetchCreditScore(userId); // from credit-score se
        int income = fetchVerifiedIncome(userId); // from income-service
        return new EligibilityData(creditScore, income);
    }

    // private inner record to hold intermediate results
    private record EligibilityData(int creditScore,
                                   int verifiedIncome) {
    }

}
```

A private inner record is useful for local communication between methods inside a class.

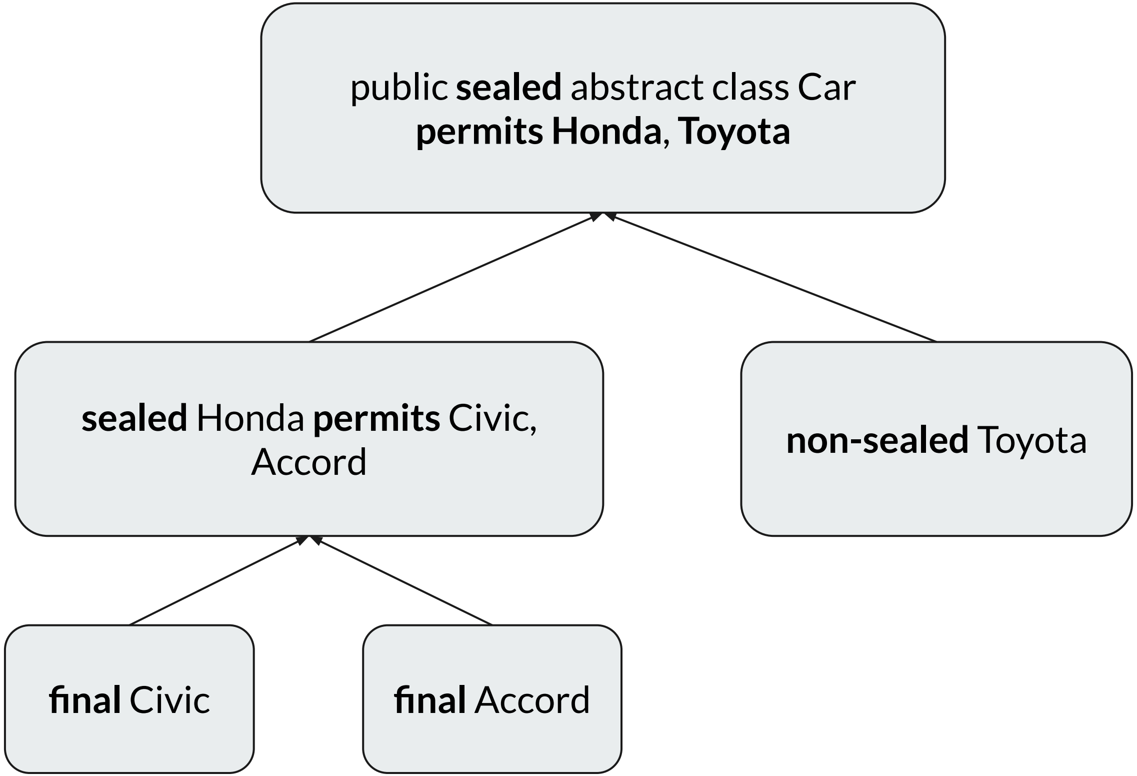
Summary - Option 5: Local Records (Helper)

```java
public class MathUtil {

    public List<String> doubleAndTriple(List<Integer> numbers) {

        // local record to hold original, double, and triple values
        record Combo(int original, int doubleVal, int tripleVal) {}
        String format = "Num: %d, Double: %d, Triple: %d";

        return numbers.stream()
                      .map(n -> new Combo(n, n * 2, n * 3))
                      .map(c -> String.format(format, c.original(), c.doubleVal(), c.tripleVal()))
                      .toList();
    }

}

// Num: 2, Double: 4, Triple: 6
```

Local record can be helpful in a stream pipelines

Sealed Types

Summary - Sealed



Pattern Matching

Summary - Pattern Matching

- Pattern matching allows us to check if an object is of a specific type or has a certain structure, then extract data from it.

Summary - Pattern Matching

- instanceof
- **switch expression**

Summary - instanceof

```java
obj instanceof String

//if true

((String) obj).isEmpty();
```

pattern variable

```java
obj instanceof String string

//if true

string.isEmpty();
```

Summary - instanceof

```java
public static boolean isEmpty(Object obj) {
    if (obj == null) {
        return true;
    } else if (obj instanceof String string) {
        return string.isEmpty();
    } else if (obj instanceof Collection<?> collection) {
        return collection.isEmpty();
    } else if (obj instanceof Map<?, ?> map) {
        return map.isEmpty();
    } else if (obj instanceof Object[] array) {
        return array.length == 0;
    }
    return false;
}
```

Summary - Problems With Switch Statement!

- Falls-through / We need break!
- You have to declare a variable and ***assign it inside for each case***.
- It is a statement. Not an ***expression***.

```java
double getTax(String country, Integer price){
    double taxRate;
    switch (country){
        case "US":
            taxRate = 0.05;
        case "UK":
            taxRate = 0.06;
            break;
        default:
            taxRate = 0.08;
    }
    log.info("country: {}, taxRate: {}", country, taxRate);
    return price * taxRate;
}
```

Summary - Switch Expression

```java
double getTax(String country, Integer price){
    var taxRate = switch (country){
        case "US" -> 0.05;
        case "UK", "AU" -> 0.06;
        case null -> throw new IllegalArgumentException("country can not be null");
        default -> {
            log.info("default tax rate is used for country: {}", country);
            yield  0.08;
        }
    };
    log.info("country: {}, taxRate: {}", country, taxRate);
    return taxRate * price;
}
```

Summary - Type Pattern

```java
boolean isEmpty(Object object){
    return switch (object) {
        case null -> true;
        case String string -> string.isEmpty();
        case Collection<?> collection -> collection.isEmpty();
        case Map<?, ?> map -> map.isEmpty();
        case Object[] array -> array.length == 0;
        default -> false;
    };
}
```

Summary - Guarded Pattern Label

```
case Integer i when i < 0 -> log.info("negative int {}", i);
```

Summary - Pattern Label Dominance

```
switch(object) {

    case ... ->
    case ... ->

}
```

```
switch (object){
    case Number n -> log.info("number {}", n); // superclass
    case Short s -> log.info("short {}", s);
    case Double d -> log.info("double {}", d);
    case Integer i -> log.info("int {}", i);
    case null, default -> log.info("default: {}", object);
}
```

Summary - Record Pattern

```
case ApiResponse(Integer data, _) -> log.info("int data: {}", data);
case ApiResponse(String data, _) -> log.info("string data: {}", data);
case ApiResponse(_, HttpTimeoutException ex) -> log.error("timed out: {}", ex.getMessage());
```

Principles Of Data Oriented Programming

Summary - Data Oriented Programming

- A programming style focused on separating the ***data*** (using simple, immutable structures) from the ***behavior*** that operates on that data.

Summary - Algebraic Data Types

- Composite Data Types - type formed by ***combining*** other types!
  - AND / Product
  - OR / Sum / Choice

Summary - Algebraic Data Types

- Meal Preference
  - Veg
  - Non-Veg
- Drink
  - Coke
  - Water
- Main Course
  - Pasta
  - Pizza
  - Burger
- Flight Class
  - Economy
  - Business
  - First
- Seat Type
  - Window
  - Aisle
  - Middle
- Contact Method
  - Phone
  - Email

Summary - Core Principles

- Model Data as Data
- Make Data Immutable
- Validate At The Boundary
- Make Illegal States Unrepresentable

Summary - Core Principles

- **Model Data as Data**
  - Treat data as facts about the world. Not as objects that know what to do (no behavior)!
  - Define your business concepts using well structured data
    - record
    - sealed (for choice types)

```java
record Product(String name, BigDecimal price) {}

record Customer(String name, String email) {}

sealed interface ShippingType {

    record Express(...) implements ShippingType {}

    record Standard(...) implements ShippingType {}

}
```

Summary - Core Principles

- Make Data Immutable
    - Once created, data should NOT change. This leads to predictable behavior and fewer bugs.
    - A record with a mutable field does not model data. It models a time-varying state.

```java
record Address(String street, String city, String zipCode) { }

record Customer(String name, String email, Address address) { }
```

Summary - Core Principles

- Validate At The Boundary
    - Perform validation when the data enters your system (e.g., HTTP request, DB read), so internal logic deals only with valid, trusted data.

```
record Email(String value) {
    public Email {
        if (!value.contains("@")){
            throw new IllegalArgumentException("Invalid email");
        }
    }
}
```

Jakarta Validation

```java
import jakarta.validation.constraints.Email;

public record EMailAddress(@Email String value) {

}
```

```java
import jakarta.validation.constraints.Size;

public record Message(@Size(min = 1, max = 10) String value) {

}
```

Summary - Core Principles

- **Make Illegal States Unrepresentable**
    - Design your data structures in such a way that it is impossible to create a ***bad*** or ***inconsistent*** state.
- Benefits
    - Sealed types guarantee all possible states are explicitly defined, preventing unexpected data shapes.
    - Compile-Time Checks ⇒ Pattern matching with switch ensures all states are handled, catching errors at compile time.

```java
public sealed interface TaskStatus {

    record Todo(String taskId) implements TaskStatus {}

    record InProgress(String taskId,
                      String assignee) implements TaskStatus {}

    record Done(String taskId,
                LocalDate completedDate) implements TaskStatus {}

}
```

Do NOT Hesitate To Create Explicit Types

```
// could be confusing! Celsius or Fahrenheit
public record Temperature(double value) {}
```

```
// much better
public record Celsius(double value) {
    public Celsius {
        if (value < -273.15) throw new IllegalArgumentException("Temperature cannot be below absolute zero!");
    }
}

public record Fahrenheit(double value) {
    public Fahrenheit {
        if (value < -459.67) throw new IllegalArgumentException("Temperature cannot be below absolute zero!");
    }
}
```

Domain Modeling

Summary - Domain Modeling

- It is a process of designing data structures that represent the key concepts in a specific business domain (e-commerce, banking, health care..)!
    - Identifying real world entities (Customer, Order, Product...).
    - Defining relationships among them.
    - Define valid states and transitions.

Summary - Domain Modeling

- Good modeling makes the software
  - Easier to read
  - Easier to test
  - Easier to extend
  - More aligned with business language

Summary - Recognizing Patterns In Domain

- Simple Values (they have domain specific meanings)
  - *EMail Address / Phone / Product Code / Price*
- ADT - AND Types (combination of values)
  - *Address, Order…etc*
- ADT - OR Types (choices)
  - *Shipping (Express, Standard)*
- Processes / Workflows
  - Actions that take inputs, perform transformations and produce outputs
  - *Pending → Paid → Shipped → Delivered*

Modeling Uncertainty With Types

Modeling Uncertainty With Types

- When we ask a question, sometimes we get an answer or sometimes we do not!

Option<T>

- Represents a value that might be present or not.
- Rewrite ***Optional<T>*** using sealed type!
- See this as an exercise - We might create some new types using this as an idea!

Either<L, R>

- Representing 2 possible outcomes!
- A value that can be one of two types - but never both!
  - Either Left (L) or Right (R)

```
// 3 rd party lib. not sealed
record Phone(String number) {}
record EMail(String address) {}
```

What To Do Now?

```java
// 3 rd party lib. not sealed
record Phone(String number) {}
record EMail(String address) {}
record PostalAddress(String street, String city, String zip) {}
```

What To Do Now?

```java
// Create a sealed wrapper
public sealed interface ContactMethod {

    record ByPhone(Phone phone) implements ContactMethod {}

    record ByEmail(EMail email) implements ContactMethod {}

    record ByPostal(PostalAddress address) implements ContactMethod {}

}
```

Error Handling

Modeling Uncertainty With Types

- Option<T>
- Either<L, R>

Modeling Uncertainty With Types

- What about File Not Found / Network error?
- How can we model errors in a structured and predictable way?

Checked Exceptions

- Exceptions are not first class citizens
  - We cannot return them / compose them
- They make the code hard to read
  - It is not always clear which part of the code throws what!

```java
try {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder(); // ParserConfigurationException
    Document doc = builder.parse(new File(inputPath));      // IOException, SAXException
} catch (ParserConfigurationException e) {
    log.error("Parser configuration error: ", e);
} catch (IOException e) {
    log.error("File read error: ", e);
} catch (SAXException e) {
    log.error("Parsing error: ", e);
}
```

File Reader

- Create a simple utility to read a file.
- Outcomes
  - File is present, we can get the data
  - File is not found
  - File is present, but access denied

Modeling Uncertainty With Types

- Option<T>
- Either<L, R>
- Result<T> - either success or failure

Is Throwing Exception Bad?

- Lead to verbosity & complexity
- Makes the code harder to read
- Exceptions break the normal flow of control where application might be left with inconsistent state
- Throwing Exceptions do have some performance overhead

Exceptions

- Checked Exceptions
  - Any business/domain related errors can be modeling using sealed types which can make the code readable.
- Run Time Exceptions
  - It is OK to throw RunTimeException whenever we need to abort the workflow

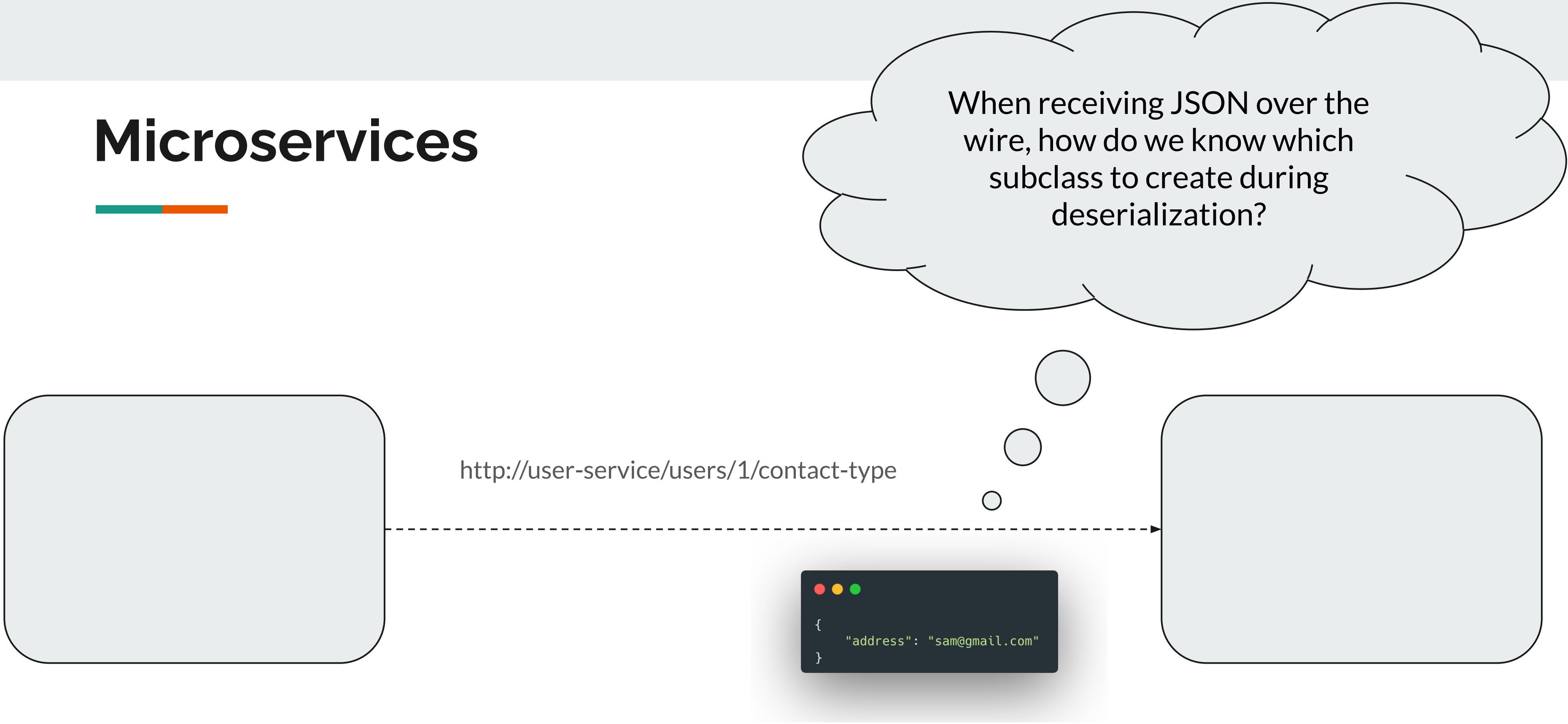
Polymorphic Deserialization

Sealed Types

```java
public sealed interface ContactType {

    record EMail(String address) implements ContactType {

    }

    record Phone(String countryCode,
                 String number) implements ContactType {

    }

}
```

```java
ver contactType = new ContactType.Phone("+1", "123-456-7890");

...
...

switch(contactType){
    case Phone phone -> ...
    case EMail eMail -> ...
}
```

Microservices



Polymorphic Deserialization

- It is the ability of a serialization library (like Jackson) to automatically determine and instantiate the correct subclass when converting JSON into Java objects.
- This makes it possible to use sealed hierarchies effectively even across microservice boundaries, without giving up type safety or flexibility.

Polymorphic Deserialization

```
@JsonTypeInfo // tells Jackson how to figure out which subtype to use during deserialization
@JsonSubTypes // lists all the possible subtypes
```

Polymorphic Deserialization

```
{ "address":"sam@gmail.com" }

{ "countryCode":"+1", "number":"123-456-7890" }
```

Polymorphic Deserialization

```
{ "info":"sam@gmail.com", "type": "email" }

{ "info":"+1-123-456-7890", "type": "phone" }
```

Polymorphic Deserialization

```java
@JsonTypeInfo(use = JsonTypeInfo.Id.DEDUCTION, defaultImpl = ContactType.EMail.
@JsonSubTypes({
        @JsonSubTypes.Type(ContactType.EMail.class),
        @JsonSubTypes.Type(ContactType.Phone.class)
})
public sealed interface ContactType {

    record EMail(String address) implements ContactType {

    }

    record Phone(String countryCode,
                 String number) implements ContactType {

    }

}
```

These annotations clutter our domain classes with technical concerns!

Microservices

When receiving JSON over the wire, how do we know which subclass to create during deserialization?

Polymorphic Deserialization

```java
@JsonTypeInfo(use = JsonTypeInfo.Id.DEDUCTION, defaultImpl = ContactType.EMail.class)
@JsonSubTypes({
        @JsonSubTypes.Type(ContactType.EMail.class),
        @JsonSubTypes.Type(ContactType.Phone.class)
})
public sealed interface ContactType {

    record EMail(String address) implements ContactType {

    }

    record Phone(String countryCode,
                 String number) implements ContactType {

    }

}
```

Polymorphic Deserialization

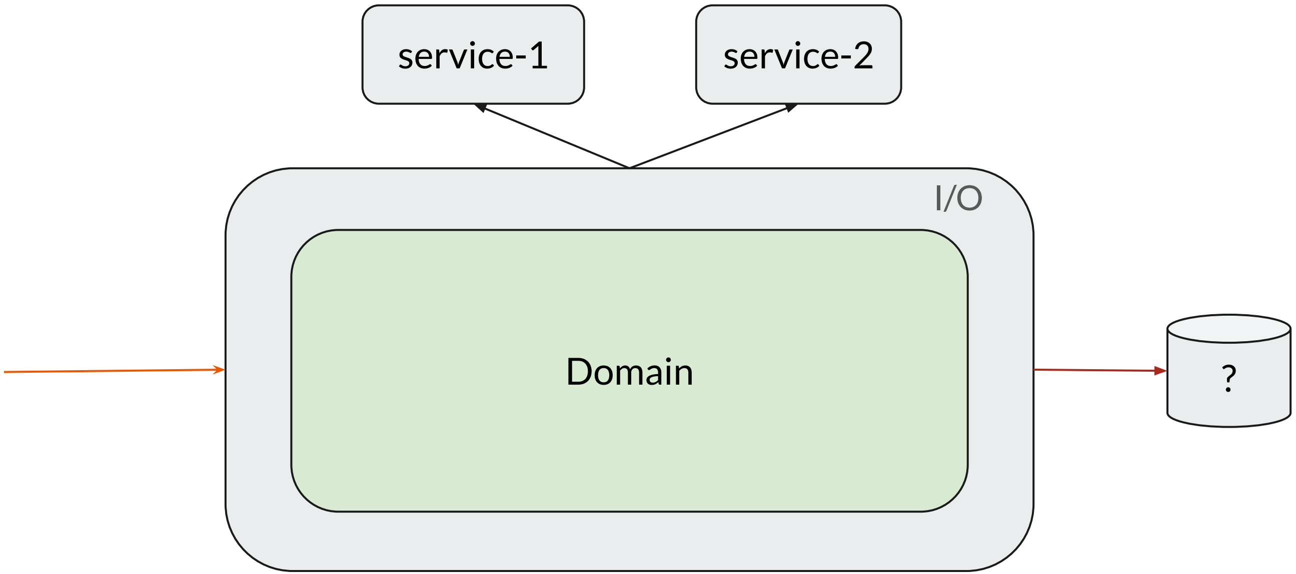
```java
@JsonTypeInfo(
        use = JsonTypeInfo.Id.NAME,
        include = JsonTypeInfo.As.PROPERTY,
        property = "type",
        defaultImpl = ContactType.EMail.class)
@JsonSubTypes({
        @JsonSubTypes.Type(value = ContactType.EMail.class, name = "email"),
        @JsonSubTypes.Type(value = ContactType.Phone.class, name = "phone")
})
public sealed interface ContactType {

    record EMail(String info) implements ContactType {

    }

    record Phone(String info) implements ContactType {

    }

}
```

Polymorphic Deserialization

```java
@JsonTypeInfo(use = JsonTypeInfo.Id.DEDUCTION, defaultImpl =
ContactType.EMail.class)
@JsonSubTypes({
        @JsonSubTypes.Type(ContactType.EMail.class),
        @JsonSubTypes.Type(ContactType.Phone.class)
})
public class ContactTypeMixIn {
}



mapper.addMixIn(ContactType.class, ContactTypeMixIn.class);
```

Persistence

What About Persistence?



Assumption

- DOP concepts might seem like overkill for a simple CRUD application.
- I assume, we use these concepts in a relatively big application.

Sealed Types → DB

- 1 Table for all Subclasses
- 1 Table per Subclass

1 Table For All Subclasses

```java
public sealed interface Payment {

    record CreditCard(String number,
                      String cvv) implements Payment {

    }


    record Paypal(String email) implements Payment {

    }

}
```

```java
public enum PaymentType {

    CREDIT_CARD,
    PAYPAL

}
```

```java
@Entity
@Table(name = "payment")
public class PaymentEntity {

    @Id
    private Integer id;

    @Enumerated(EnumType.STRING)
    private PaymentType type;

    private String number;  // for credit card
    private String cvv;     // for credit card
    private String email;   // for paypal

    // getters & setters

}
```

1 Table For All Subclasses

```java
public sealed interface Payment {

    record CreditCard(String number,
                      String cvv) implements Payment {

    }


    record Paypal(String email) implements Payment {

    }

}
```

```java
public enum PaymentType {

    CREDIT_CARD,
    PAYPAL

}
```

```java
@Entity
@Table(name = "payment")
public class PaymentEntity {

    @Id
    private Integer id;

    @Enumerated(EnumType.STRING)
    private PaymentType type;

    private String number;  // for credit card
    private String cvv;     // for credit card
    private String email;   // for paypal

    // getters & setters

}
```

id	payment_type	number	cvv	email
1	CREDIT_CARD	123-456-789	123	NULL
2	PAYPAL	NULL	NULL	sam@gmail.com

1 Table Per Subclass

```java
public sealed interface AccountType {

    Integer accountNumber();
    Integer balance();

    record Checking(Integer accountNumber,
                    Integer balance,
                    Integer overDraftLimit) implements AccountType {

    }

    record Savings(Integer accountNumber,
                   Integer balance,
                   Double interestRate) implements AccountType {

    }

}
```

checking_account

account_number	balance	overdraft_limit

savings_account

account_number	balance	interest_rate

What About Document DB?

```java
// pseudo

@Document
public sealed class Account permits SavingsAccount, CheckingAccount {

    @Id
    private String id;
    private Ineteger balance;

    ...
    ...
}

public final class SavingsAccount extends Account {
    private double interestRate;

    ...
    ...
}

public final class CheckingAccount extends Account {
    private Integer overdraftLimit;

    ...
    ...
}
```

```json
{
  "_id": "abc123",
  "_class": "com.vinsguru.persistence.SavingsAccount",
  "balance": 5000,
  "interestRate": 4.5
}
```