Java Evolution Vector: DOP

Felix Desyatirikov
T1 Holding, Innotech Group

09.10.2024

$ whoami

- Lead Developer at T1
- Certified Google Kubernetes Engine Architect
- IEEE conferences speaker

@felix_des
felixdesyatirikov@gmail.com

GitHub

Scopus

IEEE *Xplore*

Spoiler

1. **Java keeps evolving** just like any other natural language
2. New features make the code cleaner and safer:
    - **Records** – **reduce** the amount of code
    - **Sealed Classes** – ensure safety by **controlling inheritance**
    - **Pattern Matching** – simplifies code and minimises the risks
3. All of this (at this point in history) is called **Data-oriented programming**

Buzzwords

- OOP
- Instanceof
- Data processing
- Spring
- Java 21+

Agenda

1. Give a definition of **Data Oriented Programming**
2. Implement **DOP** concepts in **Java 11**
3. Bump version to **Java 17**
4. And finalize with **Java 21**
5. Trace the **evolution** of DOP in Java
6. Find out **cases for DOP** usage
7. Summarize

```
interface EntityDef {
    final class Customer implements EntityDef {
        private final  String name;

        public Customer(String name) {
            this.name = name;
        }
        public String name() {
            return name;
        }
    }
    final class Product implements EntityDef {
        private final int  id;

        public Product(int id) {
            this.id = id;
        }
        public int id() {
            return id;
        }
    }
    final class Purchase implements EntityDef {
        . . .
    }
}
```

```
void logEntityDef(EntityDef def) {
    String res;

    if (def instanceof Customer) {
        Customer cus = (Customer) def;
        res = "Customer: " + cus.name();

    } else if (def instanceof Purchase) {
        Purchase pur = (Purchase) def;
        res = format("Purchase: %s - [%s, %s]",
                pur.id(),
                pur.customer().name(),
                pur.product().id()
        );

    } else if (def instanceof Product) {
        Product prod = (Product) def;
        res = "Product: " + prod.id();
    }

    log.info(res);
}
```

```
sealed interface EntityDef {
   record Customer(String name) implements EntityDef { }
   record Purchase(int id, Customer customer, Product product) implements EntityDef { }
   record Product(int id) implements EntityDef { }
}

void logEntityDef(EntityDef def) {
   log.info(
           switch (def) {
               case Customer cus -> "Customer: " + cus.name();

               case Purchase(
                       int id,
                       Customer(var name),
                       Product(int pId)
               ) -> format("Purchase: %s - [%s, %s]", id, name, pId);

               case Product prod -> "Product: " + prod.id();
           }
   );
}
```

The DOP - definition

> What is Data-Oriented programming paradigm?

The DOP - definition

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***

Paradigm

A programming paradigm is a relatively high-level **way to** structure and **conceptualize the implementation** of a computer program

The DOP - definition

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***
> And what the purpose of a paradigm?

The DOP - definition

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***
> And what is a purpose of a paradigm?
> *To deal with **complexity***

The DOP - origins

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***
> And what is a purpose of a paradigm?
> *To deal with **complexity***
> How do we manage complexity nowadays?

The DOP - origins

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***
> And what is a purpose of a paradigm?
> *To deal with **complexity***
> How do we manage complexity nowadays?
> *With **OOP***

The DOP - origins

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***
> And what is a purpose of a paradigm?
> *To deal with **complexity***
> How do we manage complexity nowadays?
> *With **OOP***
> Why OOP?

OOP

OOP

- Methods
- Fields
- Access modifiers
- Polymorphism
- Encapsulation
- …

OOP

- Methods
- Fields
- Access modifiers
- Polymorphism
- Encapsulation
- …

Setting boundaries

OOP

- Methods
- Fields
- Access modifiers
- Polymorphism
- Encapsulation
- …

Setting boundaries

- Maintenance boundaries
- Versioning boundaries
- Encapsulation boundaries
- Compilation boundaries
- Compatibility boundaries
- Security boundaries

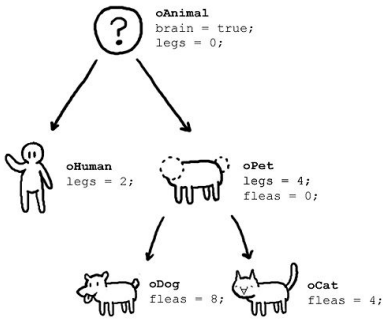
…

OOP

- Methods
- Fields
- Access modifiers
- Polymorphism
- Encapsulation
- …

Setting boundaries → Most suitable for **large systems**

What OOP users claim



OOP

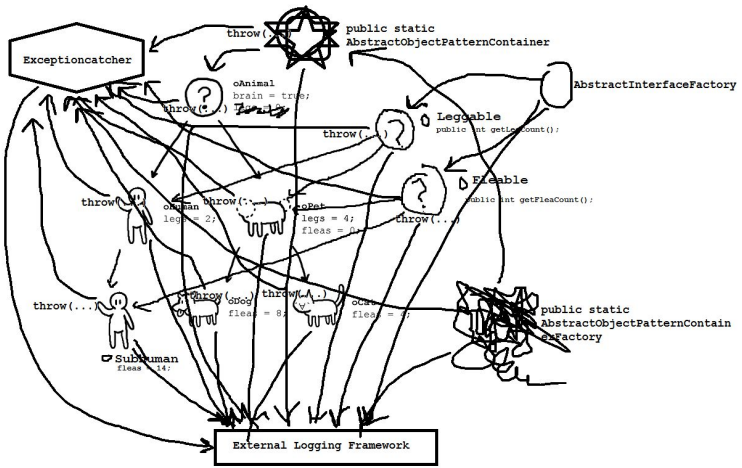
- Methods
- Fields
- Access modifiers
- Polymorphism
- Encapsulation
- …

Setting boundaries

Most suitable for **large systems**

Complexity

What actually happens

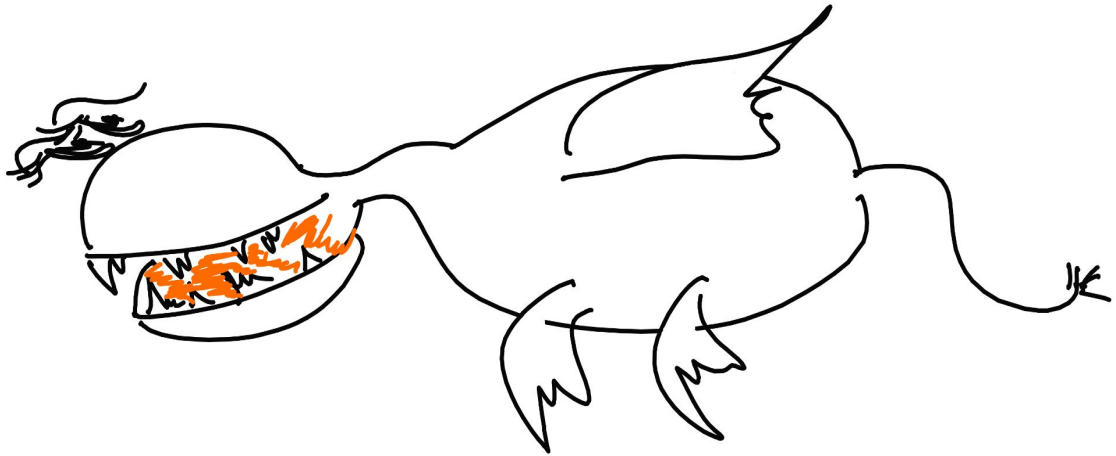


The DOP - origins

> What is Data-Oriented programming paradigm?
> *It's actually a **paradigm***
> And what is a purpose of a paradigm?
> *To deal with **complexity***
> How do we manage complexity nowadays?
> *With **OOP***
> Why OOP?
> *Convenient for **big apps***

But!

1. Why do we need big systems if we can **build** many small ones?



But!

1. Why do we need big systems if we can **build** many small ones?
2. And in small systems we don't really need clear boundaries!

But!

1. Why do we need big systems if we can **build** many small ones?
2. ~~And in small systems we don't really need clear boundaries!~~
2. If only we could **exchange** some of the **strictness** of the boundaries **for** something that would help us **do <u>things</u> more conveniently**.

Data processing

1. Put input data to DTO
2. Transform it to domain model
3. Do some processing (merge entities with data source state DB/event bus/integration)
4. Put result to DTO

Data processing - Object

1. **Put input data to DTO**
2. **Transform it to domain model**
3. Do some processing (**merge entities** with data source state DB/event bus/integration)
4. **Put result to DTO**

Data processing - Action

Data processing - Action

Interface-based approach

Dispatch-based approach

Data processing - Action

Interface-based approach

- **Contract defines** the ways how data can be **processed**
- **Actor works** with the **data** using it's **contract**
- OOP native approach

Dispatch-based approach

Data processing - Action

Interface-based approach

- **Contract defines** the ways how data can be **processed**
- **Actor works** with the **data** using it's **contract**
- OOP native approach

Dispatch-based approach

- **Contract defines type**
- **Actor works with type**
- Combines with interface-based way

Data processing - Action

Interface-based approach

- **Contract defines** the ways how data can be **processed**
- **Actor works** with the **data** using it's **contract**
- OOP native approach

~~Disp~~ DOP ~~ased~~ approach

- **Contract defines type**
- **Actor works with type**
- Combines with interface based way

Dispatch-based approach

```
interface Type
class TxData(txId: Long) extends Type
class IpData(ip: UInt) extends Type

fun process(dto: Type) {
    match (dto) {
        instanceof TxData -> {
        print("Tx id is: " + dto.txId)
    }
        instanceof IpData -> {
        print("Ip is: " + dto.ip)
    }
        default -> print("???")
    }
}
```

Dispatch-based approach

```
interface Type
class TxData(txId: Long) extends Type
class IpData(ip: UInt) extends Type

fun process(dto: Type) {
    match (dto) {
        instanceof TxData -> {
        print("Tx id is: " + dto.txId)
    }
        instanceof IpData -> {
        print("Ip is: " + dto.ip)
    }
        default -> print("???")
    }
}
```

Violations

- Polymorphism
- Encapsulation
- Open-closed principle

Dispatch-based approach

```
interface Type
class TxData(txId: Long) extends Type
class IpData(ip: UInt) extends Type

fun process(dto: Type) {
    match (dto) {
        instanceof TxData -> {
        print("Tx id is: " + dto.txId)
    }
        instanceof IpData -> {
        print("Ip is: " + dto.ip)
    }
        default -> print("???")
    }
}
```

Implementation aspects

- Polymorphism
- Encapsulation
- Open-closed principle

DOP

- Model the data, the whole data, and nothing but the data
- Data is immutable
- Validate at the boundary
- Make illegal states unrepresentable

Data Oriented Programming in Java
by Brian Goetz

DOP

- **Model the data, the whole data, and nothing but the data**
- Data is immutable
- Validate at the boundary
- Make illegal states unrepresentable

DOP

- Model the data, the whole data, and nothing but the data
- **Data is immutable**
- Validate at the boundary
- Make illegal states unrepresentable

DOP

- Model the data, the whole data, and nothing but the data
- Data is immutable
- **Validate at the boundary**
- **Make illegal states unrepresentable**

That’s all?

- Model the data, the whole data, and nothing but the data
- Data is immutable
- Validate at the boundary
- Make illegal states unrepresentable

Data-oriented programming

- These features work together to better model and process "plain data"
  - **Records** + **sealed** types = **algebraic data types**
  - **Pattern matching** = ad-hoc polymorphism (or, "Visitor without the visitors")
  - **Switch** has been considerably **upgraded** to make this even easier
- Why is this important?
  - **Java programs** are more frequently **dealing with "pure data"** coming from outside the application
  - **Program** units are **getting smaller**
  - **Boundaries defined by JSON**, not Java APIs
  - Services exchange data, not objects

Java 22 ... and beyond
W-JAX 2024
by Brian Goetz

Project Amber

Algebraic Data Types

- Records - JDK 16
- Sealed Classes - JDK 17

Pattern Matching

- Pattern Matching for instanceof - JDK 16
- Pattern Matching for switch - JDK 21
- Record Patterns - JDK 21
- Unnamed Variables & Patterns - JDK 22
- Primitive Types in Patterns, instanceof, and switch - JEP 455
- Derived Record Creation - JEP 468

Project Amber

	Summary	Pain point	“Obvious” Competition
Loom	Lightweight concurrency	"Threads are too expensive, don't scale"	Go, Elixir
ZGC	Sub-millisecond GC pauses	"GC pauses are too long"	C, Rust
Panama	Native code and memory interop SIMD Vector support	"Using native libraries is too hard" "Numeric loops are too slow"	Python, C
Amber	Right-sizing language ceremony	"Java is too verbose" "Java is hard to teach"	C#, Kotlin
Leyden	Faster startup and warmup	"Java starts up too slowly"	Go
Valhalla	Value types and specialized generics	"Cache misses are too expensive" "Generics and primitives don't mix"	C, C#
Babylon	Foreign programming model interop	"Using GPUs is too hard"	LinQ, Julia

Source

DOP 1.0 vs DOP 1.1

DOP v1.0:

- Model the data, the whole data, and nothing but the data
- Data is immutable
- Validate at the boundary
- Make illegal states unrepresentable

DOP v1.1:

- Model data immutably and transparently
- Model the data, the whole data, and nothing but the data
- Make illegal states unrepresentable
- Separate operations from data

Data-Oriented Programming in Java - Version 1.1
by Nicolai Parlog

DOP 1.0 vs DOP 1.1

DOP v1.0:

- Model the data, the whole data, and nothing but the data
- Data is immutable
- Validate at the boundary
- Make illegal states unrepresentable

DOP v1.1:

- Model data immutably and transparently
- Model the data, the whole data, and nothing but the data
- Make illegal states unrepresentable
- Separate operations from data

Relevance

Data Oriented Programming in Java
by Brian Goetz

Data-Oriented Programming in Java - Version 1.1
by Nicolai Parlog

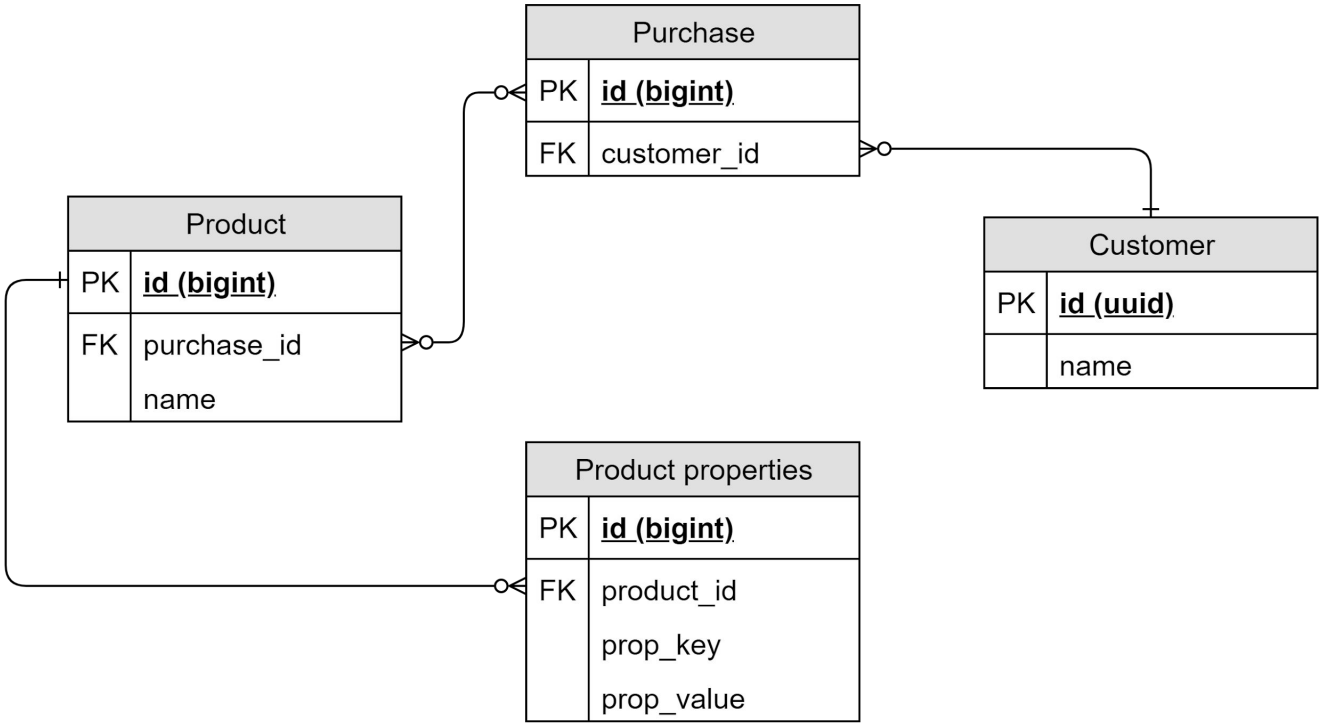
Data-Oriented Programming in Java 21
by Nicolai Parlog
Voxxed Days CERN

Data-Oriented Programming Reduce software complexity
by Yehonathan Sharvit
Book

Project Amber: news from the front
by Tagir Valeev
Joker 2017

Pattern matching and its imaginary friends
by Tagir Valeev
Joker 2018

Practical application


Purchase
PK
id (bigint)
FK
customer_id
Product
PK
id (bigint)
FK
purchase_id
name
Customer
PK
id (uuid)
name
Product properties
PK
id (bigint)
FK
product_id
prop_key
prop_value

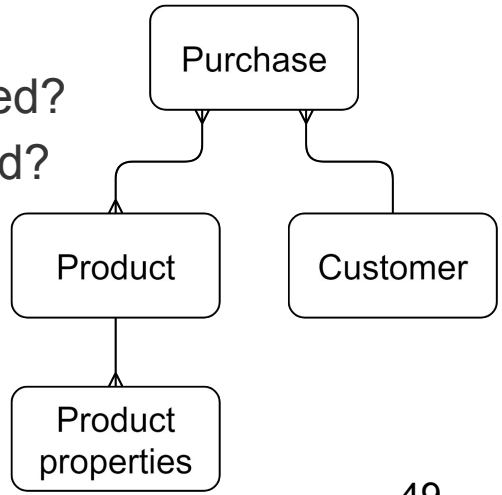
The task

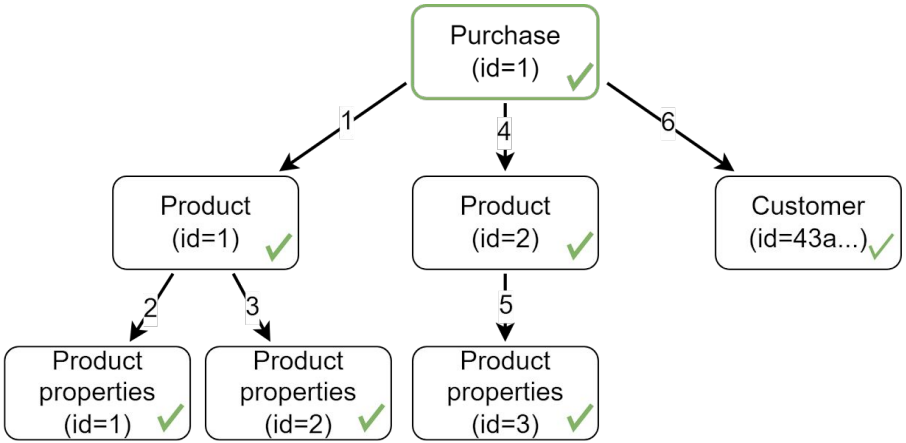
Case: build up a **composite access control** check **system for** an entity **hierarchy**

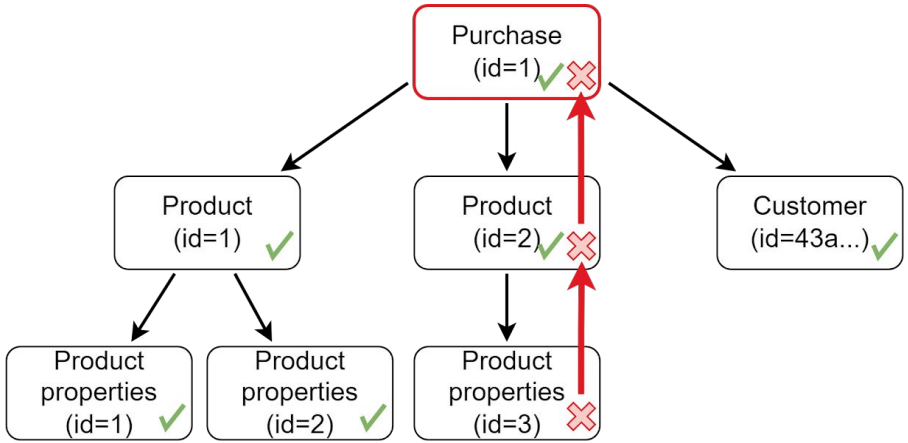
If we receive an access request for the entity Purchase with id = 1, we need to check:

- Is access to **Purchase** with id = 1 permitted?
- Is access to **Customer** related to Purchase(id=1) permitted?
- Is access to **Products** related to Purchase(id=1) permitted?
- Is access to **Product Properties** related to all Products from the previous step permitted?

If all checks pass, access is granted.




Purchase
(id=1)
1
4
6
Product
(id=1)
Product
(id=2)
Customer
(id=43a...)
2
3
5
Product
properties
(id=1)
Product
properties
(id=2)
Product
properties
(id=3)


Purchase
(id=1)
Product
(id=1)
Product
(id=2)
Customer
(id=43a...)
Product
properties
(id=1)
Product
properties
(id=2)
Product
properties
(id=3)

API

```
###
POST /composite-access-resolution/check
Content-Type: application/json


{
 "entityDefinition" : {
   "qualifier" : "PRODUCT",
   "id" : 1
 },
 "actionType" : "DELETE"
}
```

```
###
POST /composite-access-resolution/check
Content-Type: application/json


{
 "entityDefinition" : {
   "qualifier" : "CUSTOMER",
   "id" : "3fa85f64-5717-4562-b3fc-2c963f66afa6"
 },
 "actionType" : "READ"
}
```

Let's start in Java 11

```
@Data
public final class AccessCheckRequest {
   private final EntityDefinition<?> entityDefinition;
   private final ActionType actionType;
}


public enum ActionType {
   READ, WRITE, DELETE
}
```

Entity Definition

```
public enum SecuredEntityQualifier {
    CUSTOMER,
    PRODUCT,
    PRODUCT_PROPERTIES,
    PURCHASE;
}

@JsonTypeInfo(. . .)
@JsonSubTypes({. . .})
public interface EntityDefinition<T> {
    T getId();
    SecuredEntityQualifier getQualifier();
}

public interface LongIdEntityDefinition extends EntityDefinition<Long> {
}


public interface UUIDEntityDefinition extends EntityDefinition<UUID> {
}
```

Polymorphism in JSON

```
@JsonTypeInfo(
        use = JsonTypeInfo.Id.NAME,
        property = "qualifier",
        visible = true
)
@JsonSubTypes({
        @Type(value = ProductDefinition.class, name = PRODUCT),
        @Type(value = CustomerDefinition.class, name = CUSTOMER),
        @Type(value = PurchaseDefinition.class, name = PURCHASE),
        @Type(value = ProductPropertiesDefinition.class, name = PRODUCT_PROPERTIES),
})
public interface EntityDefinition<T> {
    . . .
```

```java
@Data
public final class CustomerDefinition
        implements UUIDEntityDefinition {
    private final UUID id;

    @Override
    public SecuredEntityQualifier getQualifier() {
        return CUSTOMER;
    }
}

@Data
public final class ProductDefinition implements LongIdEntityDefinition {
    private final Long id;

    @JsonCreator
    public ProductDefinition(long id) {
        if (id <= 0) throw new IllegalArgumentException("id must be positive");
        this.id = id;
    }

    @Override
    public SecuredEntityQualifier getQualifier() {
        return PRODUCT;
    }
}
```

DOP principles

1. Model the data, the whole data, and nothing but the data
2. Data is immutable
3. Validate at the boundary
4. Make illegal states unrepresentable

Entity

```
@Data
public final class
      Customer implements SecuredEntity {

   @Id
   private final UUID id;
   private final String name;

   @Override
   public SecuredEntityQualifier getQualifier() {
       return CUSTOMER;
   }
}
```

```
@Data
public final class
      Product implements SecuredEntity {

   @Id
   private final Long id;
   private final String name;

   @Override
   public SecuredEntityQualifier getQualifier()
{
       return PRODUCT;
   }
}
```

Permission issuance

```
public boolean checkAccess(AccessCheckRequest accessCheckRequest ) {
    // Build mapper
    var accessType = accessCheckRequest.actionType();
    var authorities = SecurityContextHolder.getContext().getAuthentication().getAuthorities();
    var mapper = mapperResolver.getMapperByAccessTypeAndRoleSet( accessType, authorities);

    // Fetch entity
    var entityDefinition = accessCheckRequest.entityDefinition();
    var rootEntity = entityFetcher.findEntityByDefinition( entityDefinition).orElseThrow();

    // Process the entity tree
    return entityWalkerService
            .processEntityTree( rootEntity, mapper)
            .stream()
            // If all entities down the tree are accessible -> root accessible
            .allMatch(Predicate.isEqual(true));
}
```

Entity fetcher - Strategy

```
public interface SecuredEntityFetcherStrategy<ENTITY extends SecuredEntity<ID>, ID> {
   Optional<ENTITY> fetch(ID id); // delegation
   Class<? extends EntityDefinition<ID>> getKey();
   . . .
```

Entity fetcher - Strategy

```
public class CustomerFetcherStrategy implements SecuredEntityFetcherStrategy<Customer, UUID> {
    private final CustomerRepository customerRepository;

    @Override
    public Class<? extends EntityDefinition<UUID>> getKey() {
        return CustomerDefinition.class;
    }
    @Override
    public Optional<Customer> fetch(UUID uuid) {
        return customerRepository.findById(uuid);
    }
}
```

Entity fetcher - Strategy

```
public interface SecuredEntityFetcherStrategy<ENTITY extends SecuredEntity<ID>, ID> {
    Optional<ENTITY> fetch(ID id); // delegation
    Class<? extends EntityDefinition<ID>> getKey();

    @Autowired
    default void register(SecuredEntityFetcherStrategyRegistry registry ) {
        registry.register(this);
    }
}
```

Entity fetcher - Registry

```
public class SecuredEntityFetcherStrategyRegistry  {
   private final Map<
       Class<? extends EntityDefinition<?>>,
       SecuredEntityFetcherStrategy<?, ?>
       > strategies = new HashMap<>();


   public void register(SecuredEntityFetcherStrategy<?, ?> strategy) {
       if (strategies.containsKey(strategy.getKey()))
           throw new IllegalStateException("This strategy already exist in registry");
       else
           strategies.put(strategy.getKey(), strategy);
   }

   public <T> SecuredEntityFetcherStrategy<?, T> getStrategy(EntityDefinition<T> definition) {
       return (SecuredEntityFetcherStrategy<?, T>) strategies.get(definition.getClass());
   }
}
```

Entity fetcher - Service

```
@Service
public class SecuredEntityByDefinitionRegistryFetcherImpl  {
   private final SecuredEntityFetcherStrategyRegistry  registry;

   @Override
   public <T> Optional<? extends SecuredEntity<T>> findEntityByDefinition (
       EntityDefinition<T> entityDefinition
   ) {
       var fetchStrategy = registry.getStrategy(entityDefinition);
       var id = entityDefinition.getId();
       return fetchStrategy.fetch(id);
   }
}
```

Drawbacks

1. Overengineered (a bit)

   Registry + BaseStrategy + N * [Entity]Strategy

Drawbacks

1. Overengineered (a bit)

   Registry + BaseStrategy + N * [Entity]Strategy
2. Security issues

Drawbacks

1. Overengineered (a bit)

   Registry + BaseStrategy + N * [Entity]Strategy
2. Security issues
3. Testability

Entity fetcher - Summary

\+ **Dispatch enveloped** in registry pattern

– **Too much boilerplate code**

Why is it so verbose?

Why is it so verbose?

Registry + BaseStrategy + n * [Entity]Strategy

Non-existent problems

- Dealing with **unlimited** number of **strategies**
- **Dynamic** strategy **registration**

```
public class SecuredEntityFetcherStrategyRegistry  {
   private final  Map</**/> strategies  = new HashMap<>();

   public void register(SecuredEntityFetcherStrategy <?, ?> strategy) {
       if (strategies.containsKey(strategy.getKey())) {
           throw new IllegalStateException("This strategy already exist in registry");
       } else {
           strategies.put(strategy.getKey(), strategy);
       }
   }

   public <T> SecuredEntityFetcherStrategy <?, T> getStrategy(EntityDefinition <T> definition) {
       return (SecuredEntityFetcherStrategy <?, T>) strategies.get(definition.getClass());
   }
}
```

Entity fetcher…

```
Optional<? extends SecuredEntity> result;
if (entityDefinition instanceof LongIdEntityDefinition) {
   var longIdEntityDefinition = (LongIdEntityDefinition) entityDefinition;
   if (longIdEntityDefinition instanceof ProductDefinition) {
       result = productRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition) {
       result = purchaseRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition) {
       result = productPropertiesIntegrationService.findById(longIdEntityDefinition.getId());
   } else throw new IllegalArgumentException("Not expected");

} else if (entityDefinition instanceof UUIDEntityDefinition) {
   var uuidEntityDefinition = (UUIDEntityDefinition) entityDefinition;
   if (uuidEntityDefinition instanceof CustomerDefinition) {
       result = customerRepository.findById(((CustomerDefinition) entityDefinition).getId());
   } else throw new IllegalArgumentException("Not expected");
} else throw new IllegalArgumentException("Not expected");

return result;
```

Entity fetcher

```
Optional<? extends SecuredEntity> result;
if (entityDefinition instanceof LongIdEntityDefinition) {
   var longIdEntityDefinition = (LongIdEntityDefinition) entityDefinition;
   if (longIdEntityDefinition instanceof ProductDefinition) {
       result = productRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition) {
       result = purchaseRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition) {
       result = productPropertiesIntegrationService.findById(longIdEntityDefinition.getId());
   } else throw new IllegalArgumentException("Not expected");

} else if (entityDefinition instanceof UUIDEntityDefinition) {
   var uuidEntityDefinition = (UUIDEntityDefinition) entityDefinition;
   if (uuidEntityDefinition instanceof CustomerDefinition) {
       result = customerRepository.findById(((CustomerDefinition) entityDefinition).getId());
   } else throw new  IllegalArgumentException ("Not expected");
} else throw new  IllegalArgumentException ("Not expected");

return result;
```

Issue 1 - instanceof

```
Optional<? extends SecuredEntity> result;
if (entityDefinition instanceof LongIdEntityDefinition) {
   var longIdEntityDefinition = (LongIdEntityDefinition) entityDefinition;
   if (longIdEntityDefinition instanceof ProductDefinition) {
       result = productRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition) {
       result = purchaseRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition) {
       result = productPropertiesIntegrationService.findById(longIdEntityDefinition.getId());
   } else throw new  IllegalArgumentException ("Not expected");

} else if (entityDefinition instanceof UUIDEntityDefinition) {
   var uuidEntityDefinition = (UUIDEntityDefinition) entityDefinition;
   if (uuidEntityDefinition instanceof CustomerDefinition) {
       result = customerRepository.findById(((CustomerDefinition) entityDefinition).getId());
   } else throw new  IllegalArgumentException ("Not expected");
} else throw new  IllegalArgumentException ("Not expected");

return result;
```

Issue 2 - Openness

```
Optional<? extends SecuredEntity> result;
if (entityDefinition instanceof LongIdEntityDefinition) {
   var longIdEntityDefinition = (LongIdEntityDefinition) entityDefinition;
   if (longIdEntityDefinition instanceof ProductDefinition) {
       result = productRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition) {
       result = purchaseRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition) {
       result = productPropertiesIntegrationService.findById(longIdEntityDefinition.getId());
   } else throw new IllegalArgumentException("Not expected");

} else if (entityDefinition instanceof UUIDEntityDefinition) {
   var uuidEntityDefinition = (UUIDEntityDefinition) entityDefinition;
   if (uuidEntityDefinition instanceof CustomerDefinition) {
       result = customerRepository.findById(((CustomerDefinition) entityDefinition).getId());
   } else throw new IllegalArgumentException("Not expected");
} else throw new IllegalArgumentException("Not expected");

return result;
```

Issue 3 - Casts

```
Optional<? extends SecuredEntity> result;
if (entityDefinition instanceof LongIdEntityDefinition) {
   var longIdEntityDefinition =  (LongIdEntityDefinition) entityDefinition;
   if (longIdEntityDefinition instanceof ProductDefinition) {
       result = productRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition) {
       result = purchaseRepository.findById(longIdEntityDefinition.getId());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition) {
       result = productPropertiesIntegrationService.findById(longIdEntityDefinition.getId());
   } else throw new IllegalArgumentException("Not expected");

} else if (entityDefinition instanceof UUIDEntityDefinition) {
   var uuidEntityDefinition =  (UUIDEntityDefinition) entityDefinition;
   if (uuidEntityDefinition instanceof CustomerDefinition) {
       result = customerRepository.findById ((CustomerDefinition) entityDefinition).getId());
   } else throw new IllegalArgumentException("Not expected");
} else throw new IllegalArgumentException("Not expected");

return result;
```

Entity tree walker

```
Stream<T> processEntityTreeToStream (SecuredEntity entity, Function<SecuredEntity, T> mapper) {
   var result = Stream.of(mapper.apply(entity));

   if (entity instanceof Purchase) {
       var purchase = (Purchase) entity;
       result = Stream.concat(
               result, Stream.concat(
                       purchase.getProducts()
                               .stream()
                               .flatMap(ref -> productRepo.findById(ref.getId())  .stream())
                               .flatMap(p -> processEntityTreeToStream(p, mapper)),

                       processEntityTreeToStream(
                               customerRepo.findById(purchase.getCustomerId().getId())
                                       .orElseThrow(),
                               mapper
                       )));
   } else if (entity instanceof Product) {
       . . .
   } else throw new IllegalArgumentException( "Not expected");
   return result;
}
```

Entity tree walker

```
Stream<T> processEntityTreeToStream(SecuredEntity entity,  Function<SecuredEntity, T> mapper) {
   var result = Stream.of(mapper.apply(entity));

   if (entity instanceof Purchase) {
       var purchase = (Purchase) entity;
       result = Stream.concat(
               result, Stream.concat(
                       purchase.getProducts()
                               .stream()
                               .flatMap(ref -> productRepo.findById(ref.getId()).stream())
                               .flatMap(p -> processEntityTreeToStream(p, mapper)),

                       processEntityTreeToStream(
                               customerRepo.findById(purchase.getCustomerId().getId())
                                       .orElseThrow(),
                               mapper
                       )));
   } else if (entity instanceof Product) {
       . . .
   } else throw new IllegalArgumentException("Not expected");
   return result;
}
```

Issues identified

- Instanceof
- Exception in case of unknown subclass
- Casts

Java 17

- Sealed Classes (JEP 409)
- Records (JEP 395)
- Pattern Matching for instanceof (JEP 394)

Sealed Classes

- Allows **inheritance** only for **specific predefined** types
- **Operates within** the scope of a **package**
- Defined by the **sealed/non-sealed** and **permits** modifiers

Put it into practice

```
public sealed interface EntityDefinition<T>
       permits LongIdEntityDefinition, UUIDEntityDefinition {
   T id();
   SecuredEntityQualifier getQualifier();
}

public sealed interface LongIdEntityDefinition extends EntityDefinition<Long>
       permits
       ProductDefinition,
       ProductPropertiesDefinition,
       PurchaseDefinition {
}

public sealed interface UUIDEntityDefinition extends EntityDefinition<UUID>
       permits CustomerDefinition {
}
```

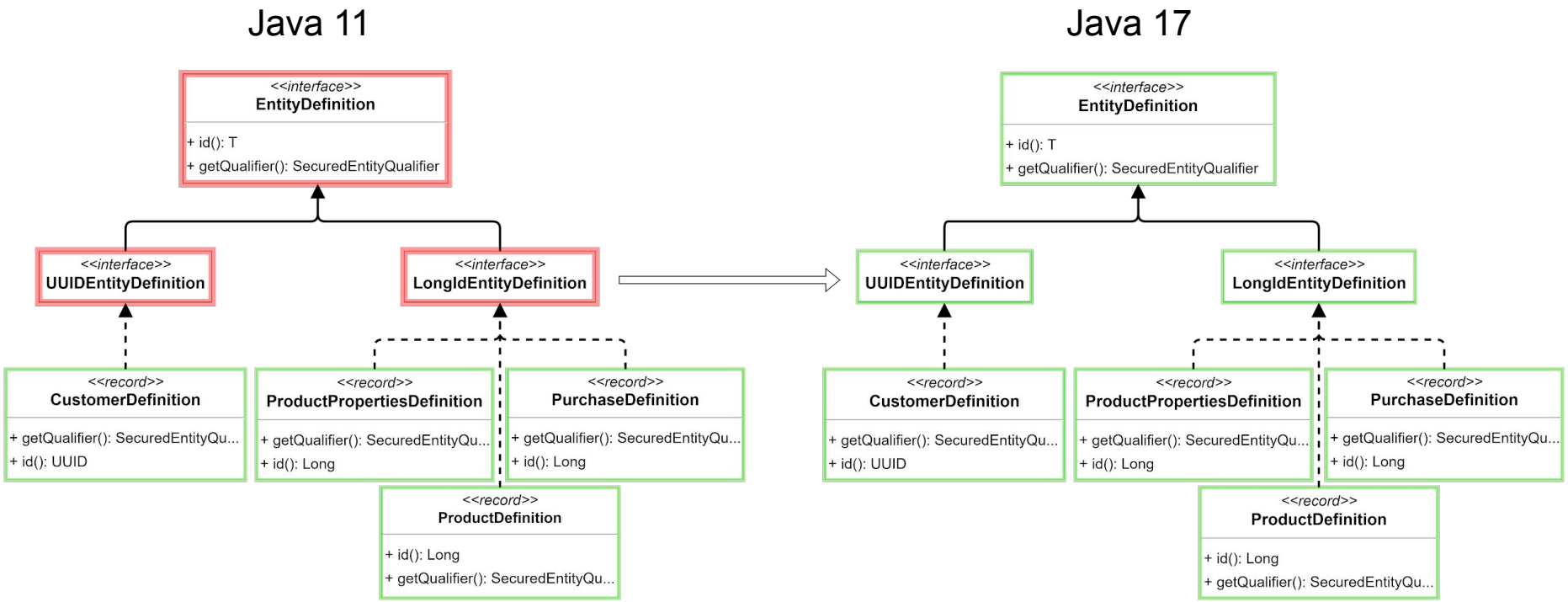
Entity definitions

```
public record ProductPropertiesDefinition(
        Long id
) implements LongIdEntityDefinition {
    public ProductPropertiesDefinition {
        if (id <= 0) throw new IllegalArgumentException("id must be positive");
    }

    @Override
    public SecuredEntityQualifier getQualifier() {
        return PRODUCT_PROPERTIES;
    }
}

public record CustomerDefinition(
        UUID id
) implements UUIDEntityDefinition {
    public SecuredEntityQualifier getQualifier() {
        return CUSTOMER;
    }
}
```

Sealed hierarchy



Entity

```
public sealed interface  SecuredEntity
       permits
       Customer,
       Product,
       ProductProperties,
       Purchase {
   SecuredEntityQualifier  getQualifier();
}
```

```
public record  ProductProperties(
       @Id
       Long id,
       String propKey,
       String propValue
) implements  SecuredEntity {
   @Override
   public  SecuredEntityQualifier  getQualifier() {
       return  PRODUCT_PROPERTIES;
   }
}
```

```
public record  Customer(
       @Id
       UUID id,
       String name
) implements  SecuredEntity {
   @Override
   public  SecuredEntityQualifier  getQualifier() {
       return  CUSTOMER;
   }
}
```

DOP & Hibernate

```
@Entity
public non-sealed class Customer implements SecuredEntity {
. . .
```

```
@Entity
public non-sealed class Product implements SecuredEntity {
. . .
```

Pattern Matching for instanceof

```java
// Java 11
if (unknownVar instanceof Type) {
    var typedVar = (Type) unknownVar;
    System.out.println(typedVar);
}


// Java 17
if (unknownVar instanceof Type typedVar) {
    System.out.println(typedVar);
}
```

Pattern Matching for instanceof (entity fetcher v.2)

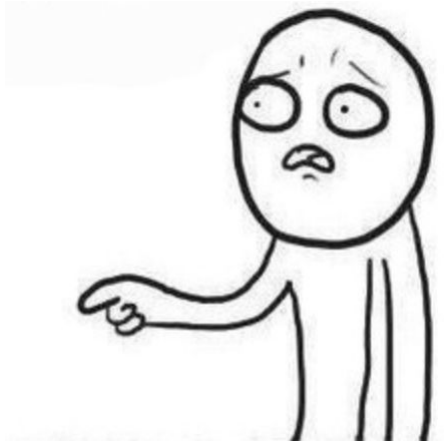
```
. . .
if (entityDefinition instanceof LongIdEntityDefinition) {
   var longIdEntityDefinition = (LongIdEntityDefinition) entityDefinition;
   if (longIdEntityDefinition instanceof ProductDefinition) {
       result = productRepository.findById(longIdEntityDefinition.getId());
. . .
```

```
. . .
if (entityDefinition instanceof LongIdEntityDefinition longIdEntityDefinition) {
   if (longIdEntityDefinition instanceof ProductDefinition def) {
       result = productRepository.findById(def.id());
. . .
```

Yes!

```
Optional<? extends SecuredEntity> result;

if (entityDefinition instanceof LongIdEntityDefinition longIdEntityDefinition ) {
   if (longIdEntityDefinition instanceof ProductDefinition def ) {
       result = productRepository.findById(def.id());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition def ) {
       result = purchaseRepository.findById(def.id());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition def ) {
       result = productPropertiesIntegrationService.findById(def.id());
. . .
```



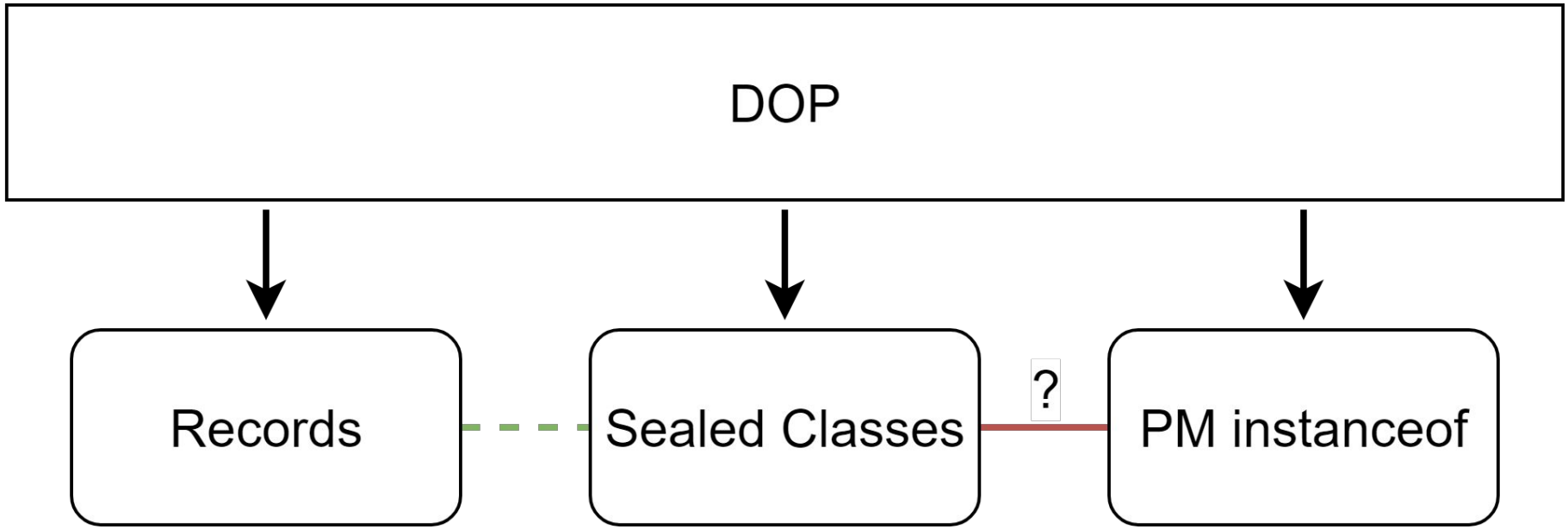
Yes, but…

```
Optional<? extends SecuredEntity> result;

if (entityDefinition instanceof LongIdEntityDefinition longIdEntityDefinition) {
   if (longIdEntityDefinition instanceof ProductDefinition def) {
       result = productRepository.findById(def.id());
   } else if (longIdEntityDefinition instanceof PurchaseDefinition def) {
       result = purchaseRepository.findById(def.id());
   } else if (longIdEntityDefinition instanceof ProductPropertiesDefinition def) {
       result = productPropertiesIntegrationService.findById(def.id());
   } else {
      throw new IllegalArgumentException("Absurdly impossible");
   }
} else if (entityDefinition instanceof UUIDEntityDefinition  uuidEntityDef) {
   if (uuidEntityDef  instanceof CustomerDefinition def) {
       result = customerRepository.findById(def.id());
   } else {
       throw new IllegalArgumentException("Absurdly impossible");
   }
} else {
   throw new IllegalArgumentException("Absurdly impossible");
}
return result;
```

Java 17 - Summary

In Java 17, it's already **possible** to build **inheritance-protected hierarchies**, but the **dispatch** mechanism for them is still **not as good** as we would like


DOP
Records
Sealed Classes
?
PM instanceof

Pattern Matching?

~~Pattern Matching~~

Data Type

A **data type** is the **allowable set of all values** of a named abstraction.

boolean: { true, false }

short: { -32768, … 32767 }

Boolean: { true, false, **null** }

Short: { -32768, … 32767 } U { **null** }

Algebraic(Composite) Data Types - ADT

Product type

Sum type

ADT

Product type

```
record type(
       byte id,
       boolean value
) {}
```

type = { **{ -2^7, … 2^7-1 } x { true, false }** } ∩ { null } =
{ **(-2^7, true), (-2^7, false), … (2^7-1, false)**, null }

Sum type

ADT

Product type

```
record type(
       byte id,
       boolean value
) {}
```

type = { **{ -2^7, … 2^7-1 } x { true, false }** } ∩ { null } = { **(-2^7, true), (-2^7, false), … (2^7-1, false)**, null }

Sum type

```
enum type{
   VALUE1,
   VALUE2
}
```

type = { **VALUE1**, null } U { **VALUE2**, null }

ADT

Product type

```
record type(
       byte id,
       boolean value
) {}
```

type = { **{ -2^7, … 2^7-1 } x { true, false }**, null} =
{ **(-2^7, true), (-2^7, false), … (2^7-1, false)**, null }

Sum type

```
enum type{
   VALUE1,
   VALUE2
}
```

type = { **VALUE1**, null } U { **VALUE2**, null }

```
sealed interface type permits typeA, typeB {}

record typeA(byte id,boolean value)
    implements type {}

record typeB(int value)
    implements type {}
```

type = { **{{ -2^7, … 2^7-1 } x { true, false }**, null **} U**
{{ -2^{31}, … $2^{31}-1$ }, null **}**

Pattern Matching

Pattern Matching - mechanism for **checking** whether an **object matches** a **pattern**

A **pattern** can be either a **supertype** in the case of ADT sum or a **field type** in ADT product (in the case of deconstruction)

Pattern Matching

```
interface Type {}

class TypeA implements Type {
   private final byte id;
   private final boolean value;
   . . .
}

class TypeB implements Type {
   private final int value;
   . . .
}
```

```
// Java 11
String match(Type type) {
   if (type instanceof TypeA) {
       return "Its A: " + ((TypeA) type).fmt();
   } else if (type instanceof TypeB) {
       return "Its B: " + ((TypeB) type).fmt();
   }
   throw new IllegalArgumentException("???");
}
```

Pattern Matching

```
sealed interface Type
    permits TypeA, TypeB {}

record TypeA(
        byte id,
        boolean value
) implements Type {}

record TypeB(
        int value
) implements Type {}
```

```
// Java 17
String match(Type type) {
    if (type instanceof TypeA a) {
        return "Its A: " + a;
    } else if (type instanceof TypeB b) {
        return "Its B: " + b;
    }
    throw new IllegalArgumentException("???");
}
```

```
// Java 21
String match(Type type) {
    return switch (type) {
        case TypeA a -> "Its A: " + a;
        case TypeB b -> "Its B: " + b;
    }
}
```

Pattern Matching for switch (JEP 441)

```
String match(Type type) {
   return switch (type) {
        case TypeA a -> "Its A: " + a;
        case TypeB b -> "Its B: " + b;
   }
}
```

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(
       byte id,
       boolean value
) implements Type {}

record TypeB(
       int value
) implements Type {}
```

Pattern Matching for switch

```
String match(Type type) {
   return switch (type) {
        case TypeA a -> "Its A: " + a;
        case TypeB b -> "Its B: " + b;
   }
}
```

1. Readability

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(
       byte id,
       boolean value
) implements Type {}

record TypeB(
       int value
) implements Type {}
```

Pattern Matching for switch

```
String match(Type type) {
   return switch (type) {
        case TypeA a -> "Its A: " + a;
        case TypeB b -> "Its B: " + b;
   }
}
```

1. Readability
2. No exceptions

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(
       byte id,
       boolean value
) implements Type {}

record TypeB(
       int value
) implements Type {}
```

Pattern Matching for switch

```
String match(Type type) {
   return switch (type) {
        case TypeA a -> "Its A: " + a;
        case TypeB b -> "Its B: " + b;
   }
}
```

1. Readability
2. No exceptions
3. Shortcut for casts

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(
       byte id,
       boolean value
) implements Type {}

record TypeB(
       int value
) implements Type {}
```

Record Patterns

```
String match(Type type) {
   return switch (type) {
       case TypeA(byte id, var ignored) ->
           "Its A! Id: " + id;

       case TypeB b -> "Its B: " + b;
   }
}
```

1. Readability
2. No exceptions
3. Shortcut for casts
4. In place deconstruction

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(
       byte id,
       boolean value
) implements Type {}

record TypeB(
       int value
) implements Type {}
```

Record Patterns

```
String match(Type type) {
   return switch (type) {
       case TypeA(ByteId(var id), var ign) ->
           "Its A! Id: " + id;

       case TypeB b -> "Its B: " + b;
   }
}

sealed interface Type permits TypeA, TypeB {}

record TypeA(
       ByteId id,
       boolean value
) implements Type {}

record ByteId(byte id) {}

record TypeB(
       int value
) implements Type {}
```

1. Readability
2. No exceptions
3. Shortcut for casts
4. In place deconstruction
5. Nested in place deconstruction

Pattern Matching for switch

```
String match(Type type) {
   return switch (type) {
       case TypeA(ByteId(byte id), var ign)
           when id >= 100 -> "Id (>=100): " + id;

       case TypeA(ByteId(byte id), var ign)
           when id < 100 -> "Id (<100): " + id;

       case TypeA ign -> "null => no id checks";

       case TypeB b -> "Its B: " + b;
   };
}
```

1. Readability
2. No exceptions
3. Shortcut for casts
4. In place deconstruction
5. Nested in place deconstruction
6. Guards

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(ByteId id, boolean value) implements Type {}
record ByteId(byte id) {}

record TypeB(int value) implements Type {}
```

Pattern Matching for switch

```
String match(Type type) {
   return switch (type) {
       case TypeA(ByteId(int id), var ign)
           when id >= 100 -> "Id (>=100): " + id;

       case TypeA(ByteId(int id), var ign)
           when id < 100 -> "Id (<100): " + id;

       case TypeA ign -> "null => no id checks";

       case TypeB b -> "Its B: " + b;
   };
}
```

1. Readability
2. No exceptions
3. Shortcut for casts
4. In place deconstruction
5. Nested in place deconstruction
6. Guards

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(ByteId id, boolean value) implements Type {}
record ByteId(byte id) {}

record TypeB(int value) implements Type {}
```

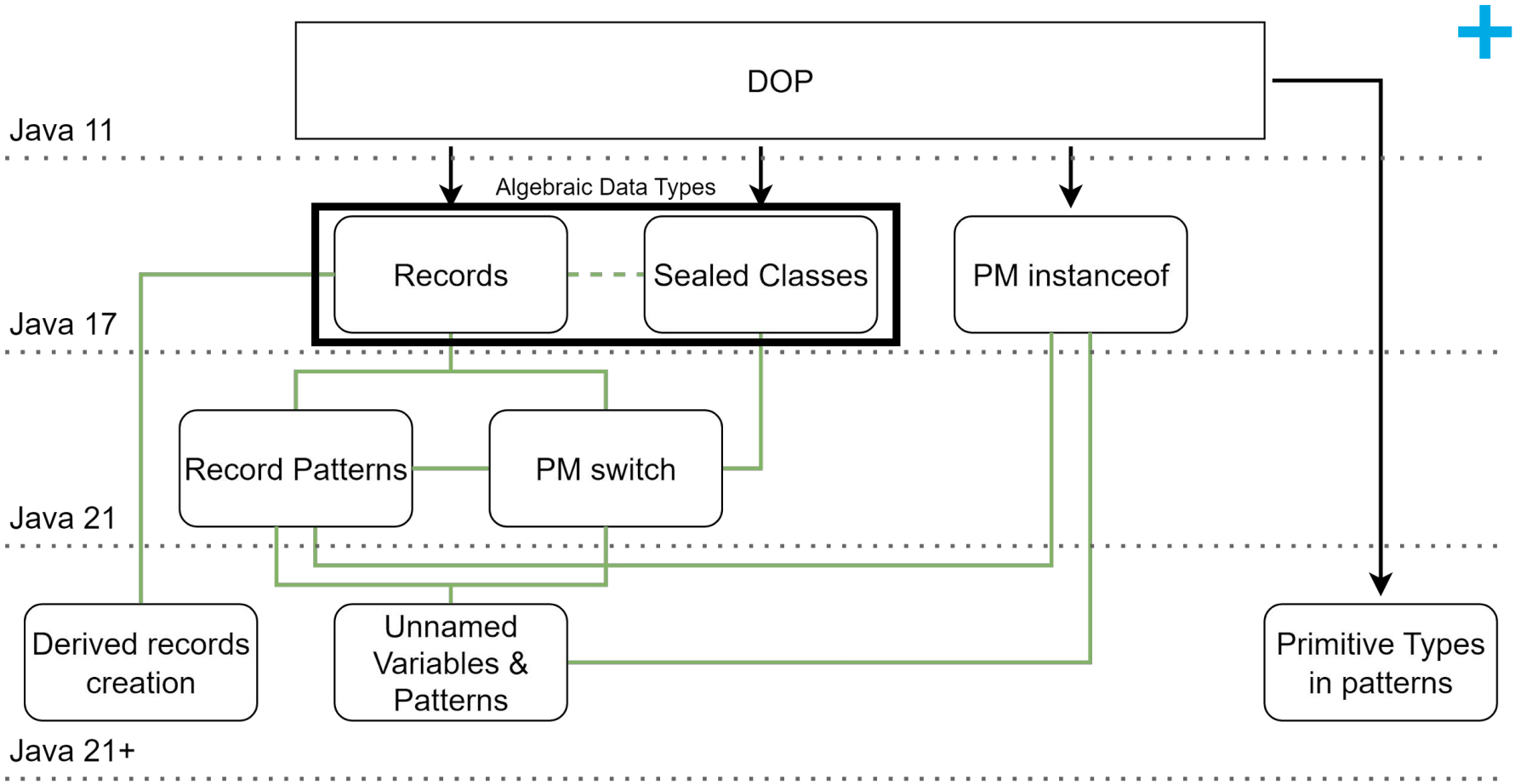
Java 21

- Pattern Matching for switch (JEP 441)
- Record Patterns (JEP 440)

Entity fetcher v.3

```
Optional<? extends SecuredEntity> findEntityByDefinition(
   EntityDefinition<?> entityDefinition
) {
   return switch (entityDefinition) {
       case LongIdEntityDefinition longIdEntityDefinition -> {
           yield switch (longIdEntityDefinition) {
               case ProductDefinition(Long id) -> productRepository.findById(id);
               case PurchaseDefinition(Long id) -> purchaseRepository.findById(id);
               case ProductPropertiesDefinition(Long id) -> productPropsService.getById(id);
           };
       }

       case CustomerDefinition(UUID id) -> customerRepository.findById(id);
   };
}
```


DOP
Java 11
Algebraic Data Types
Records
Sealed Classes
PM instanceof
Java 17
Record Patterns
PM switch
Java 21
Derived records creation
Unnamed Variables & Patterns
Primitive Types in patterns
Java 21+

Unnamed Patterns & Variables

```
Stream<T> processEntityTreeToStream(SecuredEntity entity, Function<SecuredEntity, T> mapper) {
   var result = Stream.of(mapper.apply(entity));
   switch (entity) {
       case Purchase(var ignored, var products, var customerRef) -> result = Stream.concat(
               result, Stream.concat(
                       products.stream()
                               .flatMap(ref -> productRepository.findById(ref.id()).stream())
                               .flatMap(p -> processEntityTreeToStream(p, mapper)),
                       processEntityTreeToStream(
                               customerRepository.findById(customerRef.getId()).orElseThrow(),
                               mapper
                       )
               )
       );
       case Product(Long id,  var ignored) -> result = Stream.concat(
               . . .
       );
       case Customer ignored -> { }
       case ProductProperties ignored -> { }
   }
   return result;
}
```

Unnamed Patterns & Variables

```
Stream<T> processEntityTreeToStream(SecuredEntity entity, Function<SecuredEntity, T> mapper) {
   var result = Stream.of(mapper.apply(entity));
   switch (entity) {
       case Purchase(var _, var products, var customerRef) -> result = Stream.concat(
               result, Stream.concat(
                       products.stream()
                               .flatMap(ref -> productRepository.findById(ref.id()).stream())
                               .flatMap(p -> processEntityTreeToStream(p, mapper)),
                       processEntityTreeToStream(
                               customerRepository.findById(customerRef.getId()).orElseThrow(),
                               mapper
                       )
               )
       );
       case Product(Long id, var _) -> result = Stream.concat(
               . . .
       );
       case Customer _, ProductProperties _ -> { }
   }
   return result;
}
```

Derived Record Creation / Primitive types in patterns

Derived Record Creation

```
record Response(int code, String body) { }

var basicResponse = new Response(400, "Client error;");

// JEP 468
Response notFoundResponse = basicResponse with {
    code += 4;
    body += " Not Found;";
};
```

Derived Record Creation

```
record Response(int code, String body) { }

var basicResponse = new Response(400, "Client error;");

// JEP 468
Response notFoundResponse = basicResponse with {
    code += 4;
    body += " Not Found;";
};

// Java 21
Response notFoundResponse = switch (basicResponse) {
    case Response(var code, var body) -> {
        code += 4;
        body += " Not Found;";
        yield new Response(code, body);
    }
};
```

Derived Record Creation

Goals:

1. Provide a **concise means** to create **new record** values derived **from existing record** values.
2. Streamline the declaration of record classes by eliminating the need to provide explicit wither methods, which are the immutable analogue of setter methods.

Primitive Types in Patterns, instanceof, and switch

- Enable uniform data exploration by allowing type patterns for all types, whether primitive or reference.
- Align type patterns with instanceof, and align instanceof with safe casting.
- Allow pattern matching to use primitive type patterns in both nested and top-level contexts.
- Provide easy-to-use constructs that eliminate the risk of losing information due to unsafe casts.
- Following the enhancements to switch in Java 5 (enum switch) and Java 7 (string switch), allow switch to process values of any primitive type.

Primitive Types in Patterns, instanceof, and switch

- Enable **uniform** data exploration by allowing **type patterns for all types**, whether primitive or reference.
- **Align type patterns with instanceof, and align instanceof with safe casting.**
- Allow **pattern matching to use primitive type patterns** in both nested and top-level contexts.
- Provide easy-to-use constructs that **eliminate the risk of losing information due to unsafe casts**.
- Following the enhancements to **switch** in Java 5 (enum switch) and Java 7 (string switch), allow switch to process **values of any primitive type**.

Primitive Types in Patterns, instanceof, and switch

```
sealed interface Type permits TypeA, TypeB {}

record TypeA(ByteId id, boolean value) implements Type {}
record ByteId(byte id) {}

record TypeB(int value) implements Type {}


String match(Type type) {
   return switch (type) {
       case TypeA(ByteId(int id), var _) when id >= 10 -> "Id (>=10):" + id;

       case TypeA(ByteId(int id), var _) when id < 10 -> "Id (<10)";

       case TypeB b -> "Its B: " + b;
   };
}
```

Primitive Types in Patterns, instanceof, and switch

```
// Java 17
switch (x.getStatus()) {
    case 0 -> "okay";
    case 1 -> "warning";
    case 2 -> "error";
    default -> "unknown status: " + x.getStatus();
}

// Java 23 ( preview )
switch (x.getStatus()) {
    case 0 -> "okay";
    case 1 -> "warning";
    case 2 -> "error";
    case int i -> "unknown status: " + i;
}
```

Common cases

State manipulation with **fixed** data **structure**

Common cases

State manipulation with **fixed** data **structure**

- State machines

Common cases

State manipulation with **fixed** data **structure**

- State machines
- CQRS

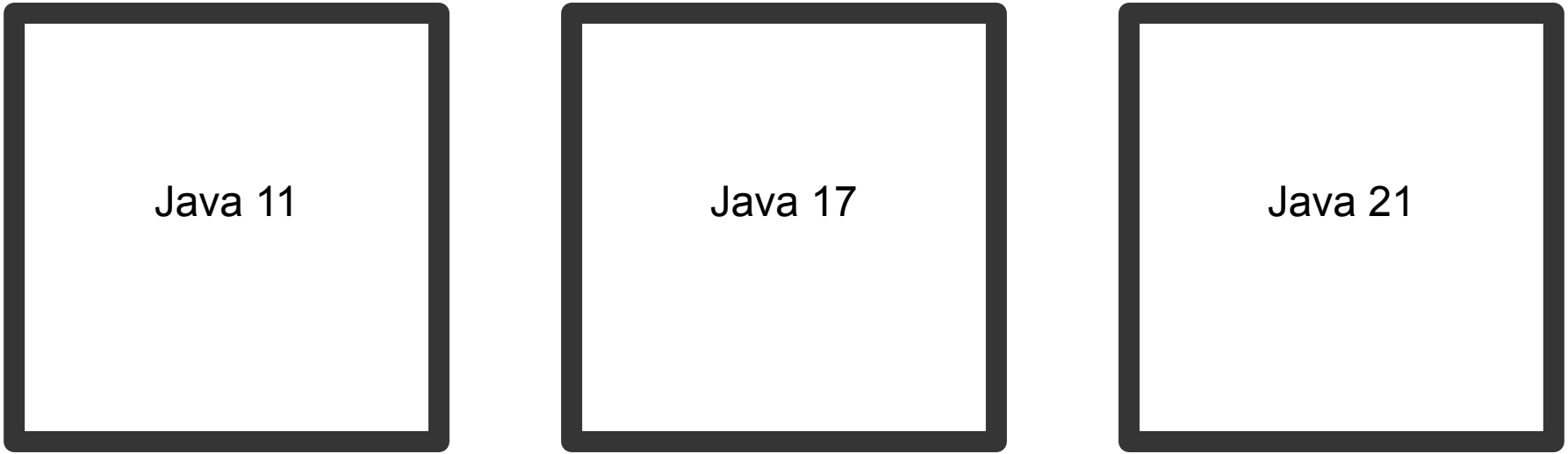
Simon Martinelli,
“CQRS and DOP with modern Java”,
Java aktuell, 2024, pp. 70-74

Common cases

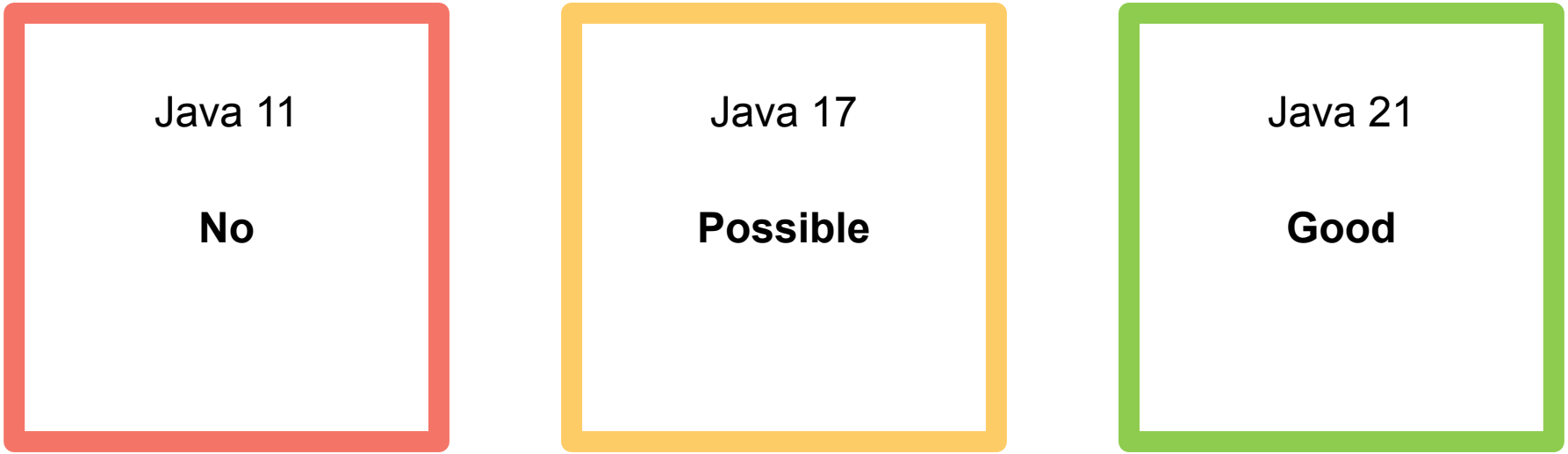
State manipulation with **fixed** data **structure**

- State machines
- CQRS
- Stream processing

Note - DOP usage in different Java LTS versions



Note - DOP usage in different Java LTS versions



Summary

1. We traced the **evolution** of Java 11 - Java 21+ in **context of DOP**
2. We reviewed how Brian Goetz's mental model of DOP **evolved from version 1.0** and the ideas it contained **to version 1.1**, as formulated by Nicolai Parlog
3. We touched on the **theoretical foundations** of DOP
4. **We confirmed in practice that DOP principles can help with making our code more laconic and secure**

Thank you!

Felix Desyatirikov, T1 Holding

@felix_des

felixdesyatirikov@gmail.com

Sources