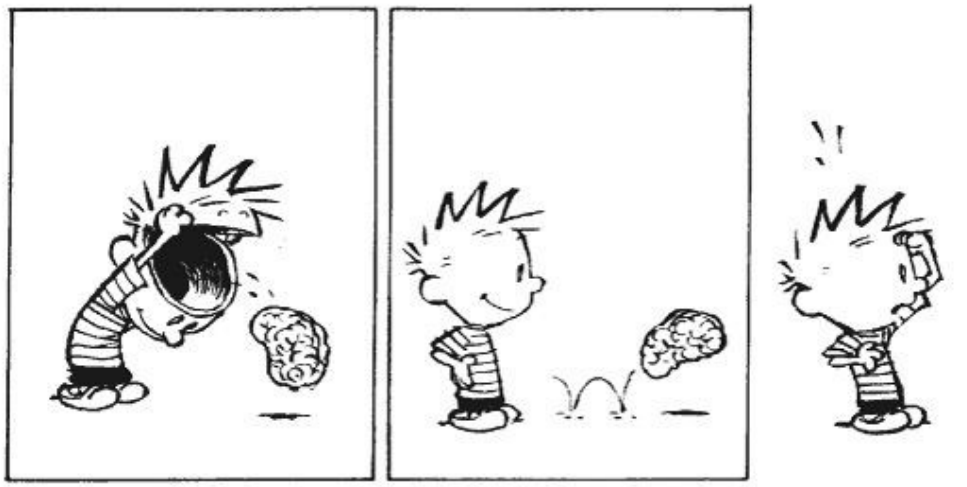
Data Oriented Programming in Java

Rémi Forax

Nantes JUG – December 2022



Warning head dump ahead !

Java ?

- 10 millions of users
- Open Source since 2006
  - https://github.com/openjdk/jdk
  - stewardship by Oracle
- A release every 6 months (actual Java 19)
  - a LTS every 2 years (support ~ 10 years)

Java Revolutions

Java is OOP (1995)

Java has type parameters (f-bounded polymorphism) (2004)

Java has lambdas (2014)

Java is/has … (2025 ??)

My work on Java

Mostly specs (+ some prototyping)

- Support for dynamic languages (invokedynamic + constant dynamic + java.lang.invoke)
- Lambdas (language + java.util)
- Module (language + java.lang.module)
- String concatenation / String interpolation (java.lang.template)
- ‘var’ et ‘_’ (local keyword / hyphenated keyword)
- Switch Expression, Record
- Virtual Threads (structured concurrency)
- Sealed Types, Pattern Matching
- Class without identity (not nullable?, tearable?)

Record (Java 17)

Named Tuples, defined by its components only

```
record Person(String name, int age) {}
```

Can be used to define type hierarchy

```
interface Vehicle {}
record Car(int seats) implements Vehicle {}
record TowTruck(Vehicle vehicle) implements Vehicle {}
```

Switch Expression (Java 17)

Allow to use switch in expression, clean the cruft inherited from C

```
String kind = ...
return switch(kind) {
  case "car", "sedan" -> new Car(…);
  case "towtruck" -> {
    System.out.println("this is a tow truck !");
    yield new TowTruck(…);
  };
  default - > throw new AssertionError("oops");
};
```

Virtual Threads

Platform threads (Java 1.0)

- Scheduled by the OS, 2M of stacks, starts in ms

Virtual threads (Java 19)

- Scheduled by the JDK, dynamic sizing, starts in µs
- Based on continuations, copy stack <--> heap

```
Thread thread = Thread.ofVirtual().starts(() - > { … });
```

String interpolation (Java 20)

```
var bob = new Person("Bob”, 32);
STR."hello \{bob.name} !";
```

STR is a template processor

- TemplateProcessor.apply(TemplateString)

  TemplateString <=> ("hello @ !" + ["Bob"])

- Avoid injection / escaping (SQL, HTML, JSON, etc)

Java is OOP ?

OOP according to Java

Encapsulation

```
class Car … { private int seats; … public void drive() {…} }
```

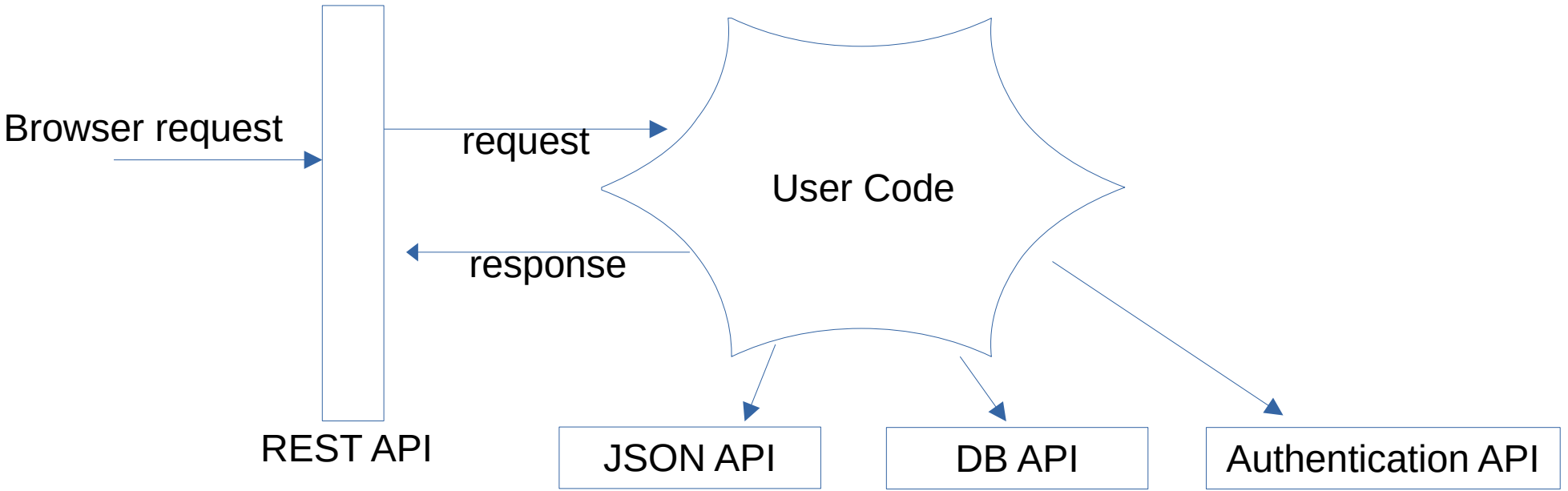
Interface and sub-typing

```
class Car implements Vehicle { … }
Vehicle vehicle = new Car(…);
```

Late binding (virtual polymorphism)

```
vehicle.toString()  // call Car.toString()
```

Anatomy of a web application



Success Story !

OOP in Java

APIs (interfaces) are more important than code

I’m a huge proponent of designing your code around the data, rather than the other way around […]
Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-- *Linus Torvalds*

Data Oriented Programming ?

Data Oriented Programming

Data First !

Data are more important than code

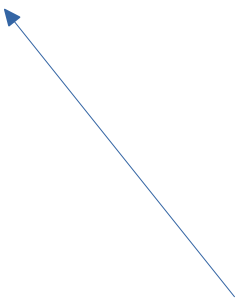
When Data change, the compiler helps

Example ?

```
interface Vehicle { int price(); }

record Car(int seats) implements Vehicle {
  public int price() { return 10 * seats; }
}

record TowTruck(Vehicle vehicle) implements Vehicle {
  public int price() { return 20 + vehicle.price(); }
}
```

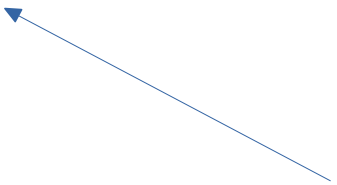


But not data first :(

Example ?

```
interface Vehicle { }
record Car(int seats) implements Vehicle {}
record TowTruck(Vehicle vehicle) implements Vehicle {}

static int price(Vehicle vehicle) {
  if (vehicle instanceof Car car) {
    return 10 * car.seats();
  }
  if (vehicle instanceof TwoTruck truck) {
    return 20 + price(truck.vehicle());
  }
  throw new AssertionError("oops");
}
```



But the compiler can not help :(

Sealed Type + Pattern Matching

But what if the definition of Car change :(

```
sealed interface Vehicle permits Car, TowTruck { }
record Car(int seats) implements Vehicle {}
record TowTruck(Vehicle vehicle) implements Vehicle {}

static int price(Vehicle vehicle) {
  return switch(vehicle) {
    case Car car -> 10 * car.seats();
    case TwoTruck truck -> 20 + price(truck.vehicle());
  };  // no default !
}
```

Record Pattern

But what if it’s not a record :(

```
sealed interface Vehicle permits Car, TowTruck { }
record Car(int seats, boolean premium) implements Vehicle {}
record TowTruck(Vehicle vehicle) implements Vehicle {}

static int price(Vehicle vehicle) {
  return switch(vehicle) {
    case Car(var seats, var premium) -> 10 * seats + premium? 100: 0;
    case TwoTruck(var vehicle) -> 20 + price(vehicle);
  };  // no default !
}
```

De-constructor


```
sealed interface Vehicle permits Car, TowTruck { }
final class Car implements Vehicle {
  private final int seats;
  private final boolean premium)
  ...
  (int,boolean) deconstuctor() {
    return match (seats, premium);
  }
}
record TowTruck(Vehicle vehicle) implements Vehicle {}

static int price(Vehicle vehicle) {
  return switch(vehicle) {
    case Car(var seats, var premium) -> 10 * seats + premium? 100: 0;
    case TwoTruck(var vehicle) -> 20 + price(vehicle);
  };  // no default !
}
```

Named Pattern ?


```
final class Optional<T> {
  …
  public static <T> Optional<T> of(T value) { … }
  public static <T> Optional<T> empty() { … }
}
```

```
Optional<String> optional = ...
return switch(optional) {
    case Optional.of(var value) -> ...;
    case Optional.empty() -> ...;
  };  // no default !
}
```

Named Pattern / Pattern Method


```
final class Optional<T> {
  private T value;
  public pattern (T) of() {
    if (value == null) return not-match;
    return match(value);
  }
  public pattern () empty() {
    if (value == null) return match();
    return no-match;
  }

  public static <T> Optional<T> of(T value) { … }
  public static <T> Optional<T> empty() { … }
}
```

```
Optional<String> optional = ...
return switch(optional) {
    case Optional.of(var value) -> ...;
    case Optional.empty() -> ...;
  };  // no default !
}
```

But this is not new !

From Scala (2004) / Kotlin (2011)

- record instead “case class” / “data class”
- sealed keyword
- deconstructor instead of unapply()

From Standard ML (1983) – Algebraic Data Type

```
datatype Article
    =  Drawing of String
    |  Toy of Maker
```

Patterns

Where to use patterns ?

In switch

```
case Point(int x, int y) -> ...
```

In instanceof

```
if (o instanceof Point(int x, int y)) { ...
```

In for(:)

```
for(Point(int x, int y) : points) { ...
```

In assignment

```
Point(int x, int y) = point;
...
```

Patterns

Type pattern

```
case String s
```

Destructor pattern / Record pattern

```
case Person(String name, int age)
```

Any pattern

```
_
```

Var pattern (inference)

```
var name
```

Constant pattern

```
case Person(eq "Bob", int age)
```

Named Pattern (pattern method)

```
case Optional.of(String s)
```

Array Pattern / Rest Pattern / Collection Literal Pattern

```
case String[] { String first, … }
```

DOP vs OOP ?

Wadler’s Expression Problem

Polymorphism

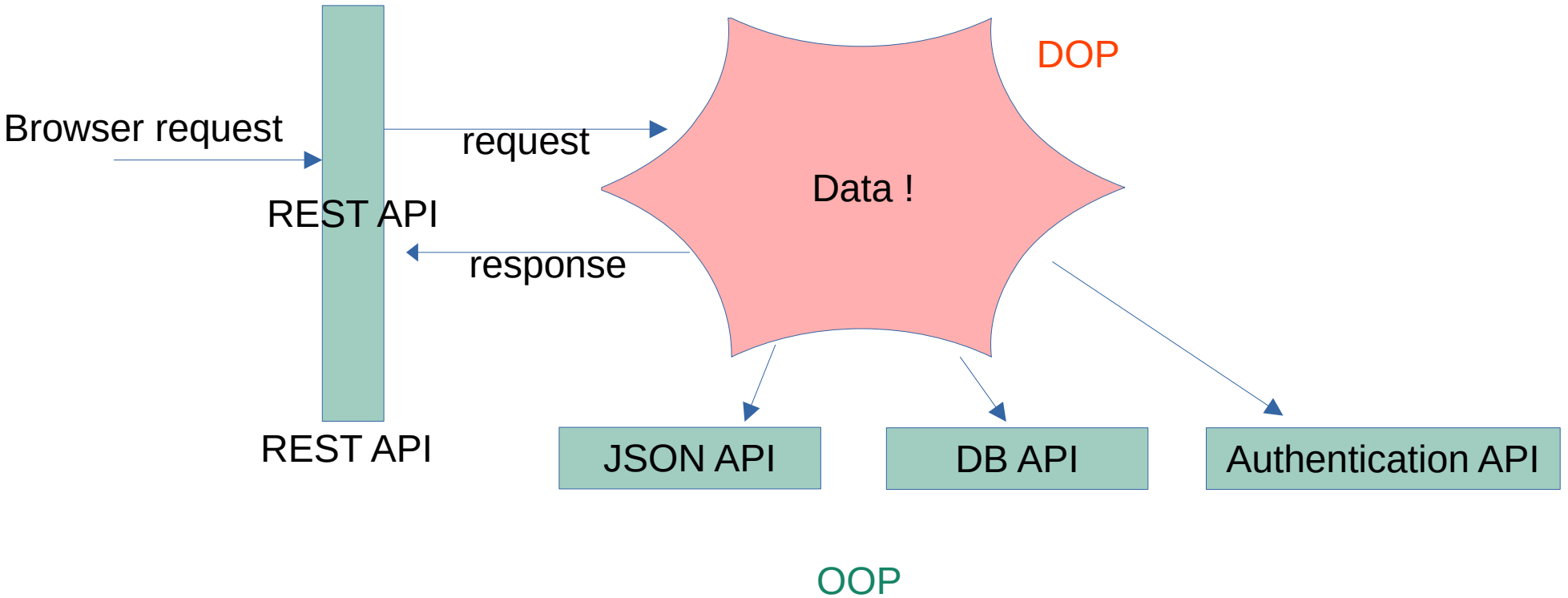
- add new subtypes
- No new operation

Pattern Matching

- Add new operations
- No new subtype

We can not get both :(

Work at different scales



Questions ?

https://github.com/forax/dop-examples/tree/master/data-first/src/main/java/com/github/forax/dop