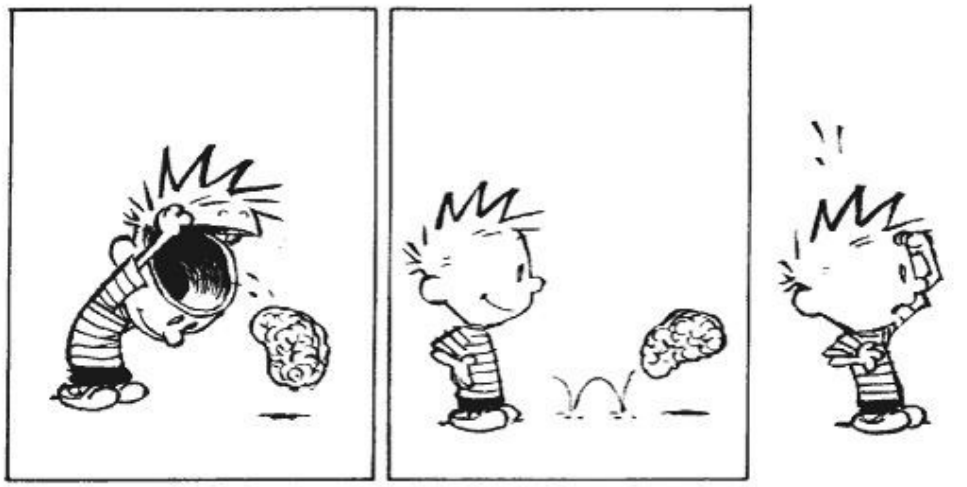
Pattern Matching in Java

Rémi Forax
Université Gustave Eiffel – May 2022



Don’t believe what I’m saying !

Pattern Matching ?

String Pattern Matching ?

~~String Pattern Matching ?~~

Tree Pattern Matching

Tree data structure

Given a recursive? data structure

```
Item = VideoGame(String name, PEGI rating)
     | ActionFigure(int id, String universe)
     | Box(List<Item> items)
```

we want to compute thing on it

Tree data structure in Java

```
interface Item {}

class VideoGame implements Item {
  final String name;
  final PEGI rating;
  ...
}

class ActionFigure implements Item {
  final String id;
  final String universe;
  ...
}

class Box implements Item {
  final List<Item> items;
  ...
}
```

Defining a computation

```
static int price(Item item) {
  if (item instanceof VideoGame) {
    var videoGame = (VideoGame) item;
    return videoGame.rating().year() * 50;
  }
  if (item instanceof ActionFigure) {
    var actionFigure = (ActionFigure) item;
    return actionFigure.universe.equals("Marvel")? 30: 20;
  }
  if (item instanceof Box) {
    var box = (Box) item;
    return 5 + box.items.stream().mapToInt(i -> price(i)).sum();
  }
  throw new AssertionError("oops");
}
```

In OOP

```
interface Item {
  abstract int price();
}

class VideoGame implements Item {
  ...
  int price() { return ratings.year() * 50; }
}

class ActionFigure implements Item {
  ...
  int price() { return universe.equals("Marvel")? 30: 20; }
}

class Box implements Item {
  ...
  int price() { return 5 + items.stream().mapToInt(Item::price).sum(); }
}
```

If … instanceof vs OOP

If … instanceof

- new computation: yes
- new subtype: not detected at compile time

OOP

- new computation: yes only if maintainer
- new subtype: yes even if not maintainer

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*

Designing code around data

data class Scala (or Kotlin)

```
sealed trait Item
case class VideoGame(name: String, rating:PEGI) extends Item
case class ActionFigure(id: int, universe: String) extends Item
case class Box(items: List<Item>) extends Item

def price(item: Item): int = item match {
  case VideoGame(name, rating) => ...
  case ActionFigure(id, universe) => ...
  case Box(items) => ...
}
```

Good Artists Copy, Great Artists Steal
-- Pablo Picasso

What we want in Java

```
sealed interface Item permits VideoGame, ActionFigure, Box {}
record VideoGame(String name, PEGI rating) implements Item {}
record ActionFigure(int id, String universe) implements Item {}
record Box(List<Item> items) implements Item {}

int price(Item item) {
  return switch(item) {
      case VideoGame(String name, PEGI rating) -> ...
      case ActionFigure(int id, String universe) -> ...
      case Box(List<Item> items) -> ...
  };
}
```

OpenJDK Project Amber

Java 17

- Arrow switch, switch expression
- Instanceof + Type pattern
- Record
- Sealed type

Java 19 (preview feature)

- Switch and null
- Switch on patterns
  - Type pattern
  - Record pattern
- Guard on switch case

Demo Java 17

Problems of the Old / C switch

Problems: Fall-through, Declaration scoped to the whole switch, Not expression oriented

```
enum PEGI {
  PEGI12, PEGI16, PEGI18;
  int year() {
    int year;
    switch(this) {
      case PEGI12:
        year = 12;
        break;
      case PEGI16:
        year = 16;
        break;
      case PEGI18:
        System.out.println("DEBUG");
        year = 18;
        break;
      default:
        throw new AssertionError(...);
    }
    return year;
  }
}
```

Enhanced switch

An arrow is followed by a statement or a block (if there are several instructions)

```
enum PEGI {
  PEGI12, PEGI16, PEGI18;

  int year() {
    int year;
    switch(this) {
      case PEGI12 -> year = 12;
      case PEGI16 -> year = 16;
      case PEGI18 -> {
          System.out.println("DEBUG");
          year = 18;
        }
      default:
        throw new AssertionError(...);
    }
    return year;
  }
}
```

Switch Expression

Switch is extended to also work as an expression (and be exhaustive)

```
enum PEGI {
  PEGI12, PEGI16, PEGI18;

  int year() {
    return switch(this) {
      case PEGI12 - > 12;
      case PEGI16 - > 16;
      case PEGI18 - > {
          System.out.println("DEBUG");
          yield 18;   // “returns” from the switch not the method
        }
      default:
        throw new AssertionError(...);
    };
  }
}
```

Instanceof + Type pattern

instanceof is enhanced to specify a local variable as last parameter

```
int price(Item item) {
  if (item instanceof VideoGame videoGame) {
    return videoGame.rating().year() * 50;
  }
  if (item instanceof ActionFigure actionFigure) {
    return actionFigure.universe().equals("Marvel")? 30: 20;
  }
  if (item instanceof Box box) {
    return 5 + box.items().stream().mapToInt(item -> price(item)).sum();
  }
  throw new AssertionError("oops");
}
```

instanceof and equals

The binding (local variable) is also accessible after a &&

```
class VideoGame {
  final String name;
  final PEGI rating;
  …
  public boolean equals(Object o) {
    return o instance VideoGame videoGame
      && rating == videoGame.rating
      && name.equals(videoGame.name);
  }
}
```

Sealed type and permits

A sealed types list all its direct subtypes

```
sealed interface Item permits VideoGame, Box {}
final class VideoGame implements Item { … }
final class ActionFigure implements Item { … }
final class Box implements Item { … }
```

A subtype must be either **final**, **sealed** or **non-sealed**

Sealed type

All subtypes must have a *stable* name and declared in the same package (or module)

- A lambda is not a valid subtype
- A local class is not a valid subtype (may be relaxed)

“permits” is not necessary if the sealed type and all its subtypes are in the same file

Record

Unmodifiable Named Tuple

```
record VideoGame(String name, PEGI rating) {}
```

“name” and “rating” are components of the record

- No more fields are allowed

Accessors name() and rating() are generated
(they do not follow the getter convention)

Canonical Constructor

A record has an overridable constructor that takes all the components

```
record VideoGame(String name, PEGI rating) {
  VideoGame(String name, PEGI rating) {
    Objects.requireNonNull(name);
    Objects.requireNonNull(rating);
    this.name = name;
    this.rating = rating;
  }
}
```

Canonical Compact constructor

There is a syntactic sugar version of the canonical constructor

```
record VideoGame(String name, PEGI rating) {
  VideoGame {  // the compiler insert the parameters
    Objects.requireNonNull(name);
    Objects.requireNonNull(rating);

    // all the lines “this.foo = foo;” are added by the compiler
  }
}
```

Equals/hashCode/toString()

equals/hashCode and toString are generated

Technically the compiler generates the header and the JDK implement them (less bytecodes)

```
assertEquals(
  Set.of(new VideoGame("naruto", PEGI16)),
  Set.of(new VideoGame("naruto", PEGI16)));
```

More on records

Final class that extends java.lang.Record

- It can implement interfaces

Auto-serializable (if it implements Serializable)

- Do not bypass the constructor when deserializing

Reflection: Class.getRecordComponents() returns a RecordComponent(String name, Class<?> type)

- GetName(), getType(), getAccessor()

Demo Java 18
(with --enable-preview)

Switch and null

By default a switch does not accept null

```
String s = ...
switch(s) {
    case null -> …
    case "foo", "bar" -> …
    default -> ...
}
```

If a “case null” is present, switch accepts null

Switch and null (2)

“default” by default does not accept null

```
String s = ...
switch(s) {
    case "foo", "bar" -> …
    case null, default -> ...
}
```

but “case null” can be combined with “default”

Switch statement + enum

A switch statement on an enum does not ask for exhaustiveness (backward compatibility issue)

Adding a “case null” makes it exhaustive

```
switch(rating) {
  case null - > throw null;  // make the switch exhaustive
  case PEGI12 -> ...
  case PEGI16 -> ...
  case PEGI18 -> ...
}  // can not fall-through anymore
```

Switch on type pattern

Even the switch statement must be exhaustive

```
void foo(Item item) {
  switch(item) {
      case VideoGame videoGame -> ...
      case ActionFigure actionFigure -> ...
      case Box box -> ...
  }
}
```

Switch on type pattern (2)

“case” on a subtype should appear first (like try/catch)

```
int price(Item item) {
  return switch(item) {
      case Item i -> ...
      case VideoGame videoGame -> …  // not reachable
  };
}
```

Java 19

Switch on record pattern

Guard on switch case

Record Pattern

Match and de-structure

```
int price(Item item) {
  return switch(item) {
      case VideoGame(String name, PEGI rating) -> ...
      case ActionFigure(int id, String universe) -> ...
      case Box(List<Item> items) -> ...
  };
}
```

The type patterns inside a record pattern match null

Record Pattern syntax

A record pattern has an optional binding for the record itself

RecordType(c_0*Type* c_0*, …) name?*

```
int price(Item item) {
  return switch(item) {
      case VideoGame(String name, PEGI rating) videoGame -> ...
      ...
  };
}
```

Record Pattern (2)

Also works with instanceof

```
class VideoGame {
  final String name;
  final PEGI rating;
  …
  public boolean equals(Object o) {
    return o instance VideoGame(String name2, PEGI rating2)
      && rating == rating2
      && name.equals(name2);
  }
}
```

Guarded case

A case can add an additional boolean expression using “when”

```
void foo(Item item) {
  switch(item) {
      case VideoGame(String name, PEGI rating)
          when rating == PEGI18 -> …
      case VideoGame videoGame -> ...
      ...
  }
}
```

A case with a guard must appear before the case without a guard

Future

Future of pattern matching

De-constructor on class

- Allow class to export values without breaking encapsulation

Pattern on List, Map, Set, etc

- Requires a real syntax for List, Set, Map first

User defined Pattern methods

- Users can create their own patterns

Pattern assignment

- de-structuring while assigning

De-constructor on class

Export values for pattern matching without getters *(experimental syntax)*

```
class Point {
  final int x;
  final int y;
  ...

  public (int, int) deconstructor {
    return match x, y;  // always match
  }
}

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

Concise syntax for collections

Add new syntax for usual collections *(experimental syntax)*

- Non mutable (equivalent to List.of(), Set.of(), Map.of())
- Use target typing (may require a new kind of cast)

```
List<String> list = [ "foo", "bar" ];                    // non mutable
var list2 = new ArrayList<>([ "foo", "bar" ]);           // mutable
var list3 = (List<>) [ "foo", "bar" ];
Set<String> set = [ "foo", "bar" ];
Map<String, Integer> map = { "foo": 3, "bar": 4 };
```

Pattern on collections

Pattern on collection has two variants

- Type [value$_0$, value$_1$, …] // size >= 2
- Type [value$_0$, value$_1$] // size == 2

```
Collection<String> c = ...
switch(c) {
    case List ["foo", ...] - > …
    case ArrayList [1, 2] - >  …  // does not compile int != String
    ...
}
```

+ a syntax for pattern on arrays

User defined pattern method

Use instance methods to define pattern methods *(experimental syntax)*

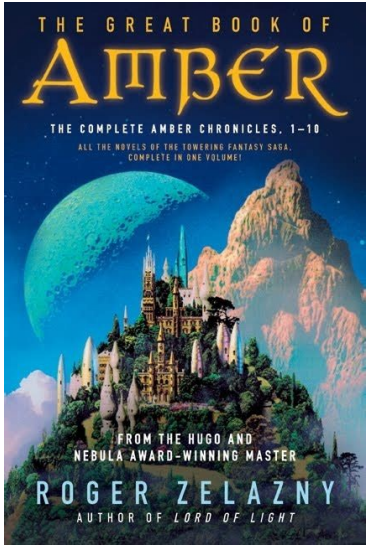
```
class Optional<T> {
  final T value;
  final boolean exist;
  …
  pattern (T) of() { if (exists) match value; else not-match; }
  pattern () empty() { if (exist) not-match; else match; }
}

switch(opt) {
  case Optional.of(T t) - > …
  case Optional.empty() - > ...
}  // how to say that this is exhaustive ??
```

Pattern assignment

Use record pattern on the left hand side of an assignment *(experimental syntax)*

- Assignment conversions should apply

```
record Box<T>(T value) {}
Box<Integer> box = …
Box<>(int v) = p;     // auto-unboxing conversion
System.out.println(v);
```



Questions ?

https://github.com/forax/pattern-matching