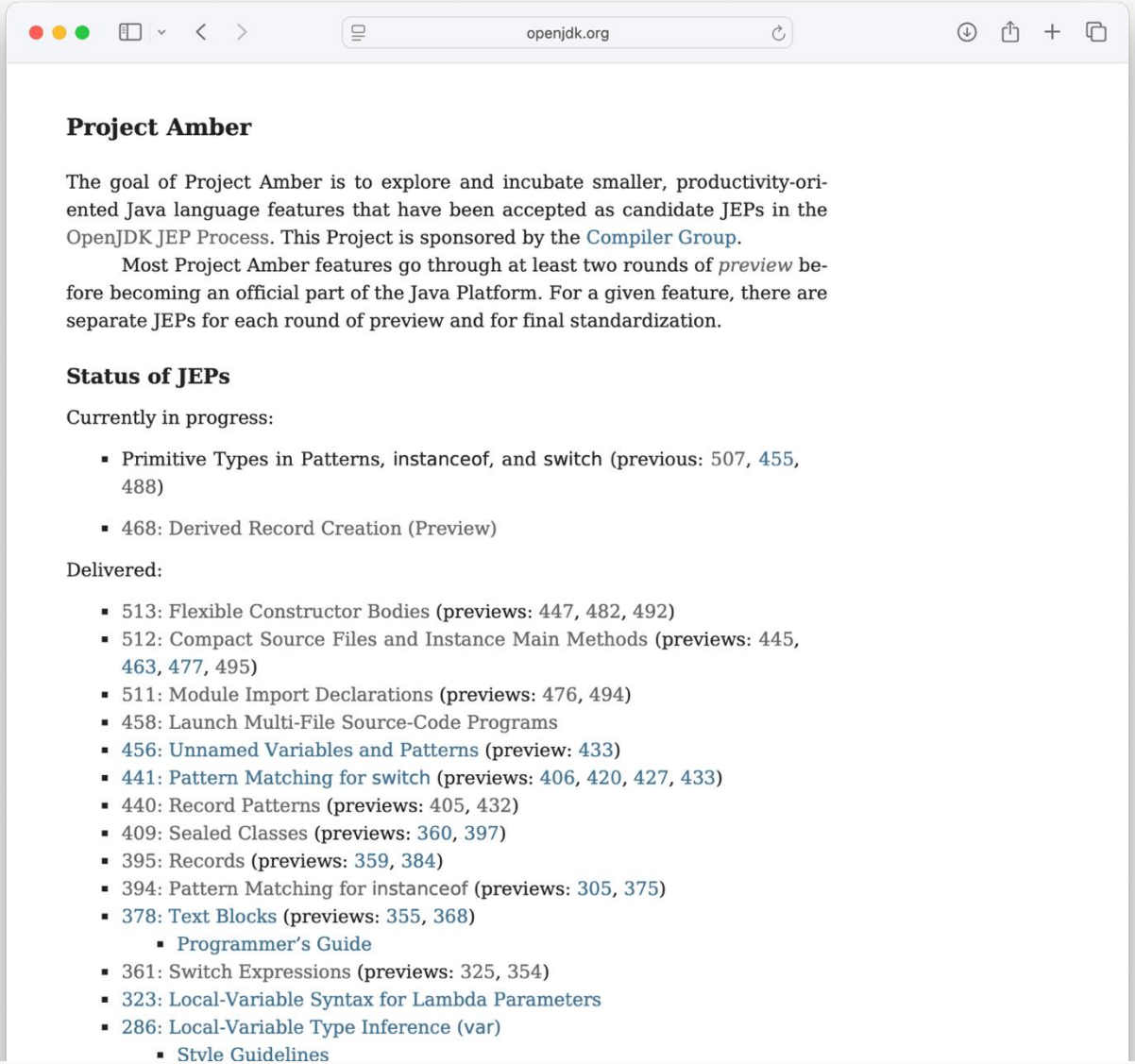
ORACLE

Project Amber Update

A review of progress from 2020 to 2025

Gavin Bierman
Java Platform Group
Oracle

Project Amber



"The goal of Project Amber is to explore and incubate smaller, productivity-oriented Java language features"

https://openjdk.java.net/projects/amber/

	2020	2021		2022		2023		2024		2025	
	15	16	17	18	19	20	21	22	23	24	25
Text Blocks	FINAL	→	→	→	→	→	→	→	→	→	→
Records	PRE	FINAL	→	→	→	→	→	→	→	→	→
Pattern matching for `instanceof`	PRE	FINAL	→	→	→	→	→	→	→	→	→
Sealed classes	PRE	PRE	FINAL	→	→	→	→	→	→	→	→
Pattern matching for `switch`			PRE	PRE	PRE	PRE	FINAL	→	→	→	→
Record Patterns					PRE	PRE	FINAL	→	→	→	→
Unnamed variables & patterns							PRE	FINAL	→	→	→
Compact source files & instance main methods							PRE	PRE	PRE	PRE	FINAL
Flexible constructor bodies								PRE	PRE	PRE	FINAL
Module Import Declarations									PRE	PRE	FINAL
Primitive Patterns									PRE	PRE	PRE

	2020	2021		2022		2023		2024		2025	
	15	16	17	18	19	20	21	22	23	24	25
Text Blocks	FINAL										
Records	PRE	FINAL									
Pattern matching for `instanceof`	PRE	FINAL									
Sealed classes	PRE	PRE	FINAL								
Pattern matching for `switch`			PRE	PRE	PRE	PRE	FINAL				
Record Patterns					PRE	PRE	FINAL				
Unnamed variables & patterns							PRE	FINAL			
Compact source files & instance main methods							PRE	PRE	PRE	PRE	FINAL
Flexible constructor bodies								PRE	PRE	PRE	FINAL
Module Import Declarations									PRE	PRE	FINAL
Primitive Patterns									PRE	PRE	PRE

Data-oriented programming

These individual features stand by themselves but form part of a broader feature arc concerning a better treatment of data.

OOP is *brilliant* at modeling complex entities and processes but *less good* at modeling plain old data

Many applications today are smaller services that consume and emit data (often untyped plain old data formats like JSON)

Balance of power has shifted ☹

Algebraic data types

Algebraic data types (ADTs) are a powerful paradigm from functional programming for describing plain old data

ADTs are built using **product** and **sum** types.

Records are nominal product types, e.g.

- A point is two integers
- A colored point is two integers and a color

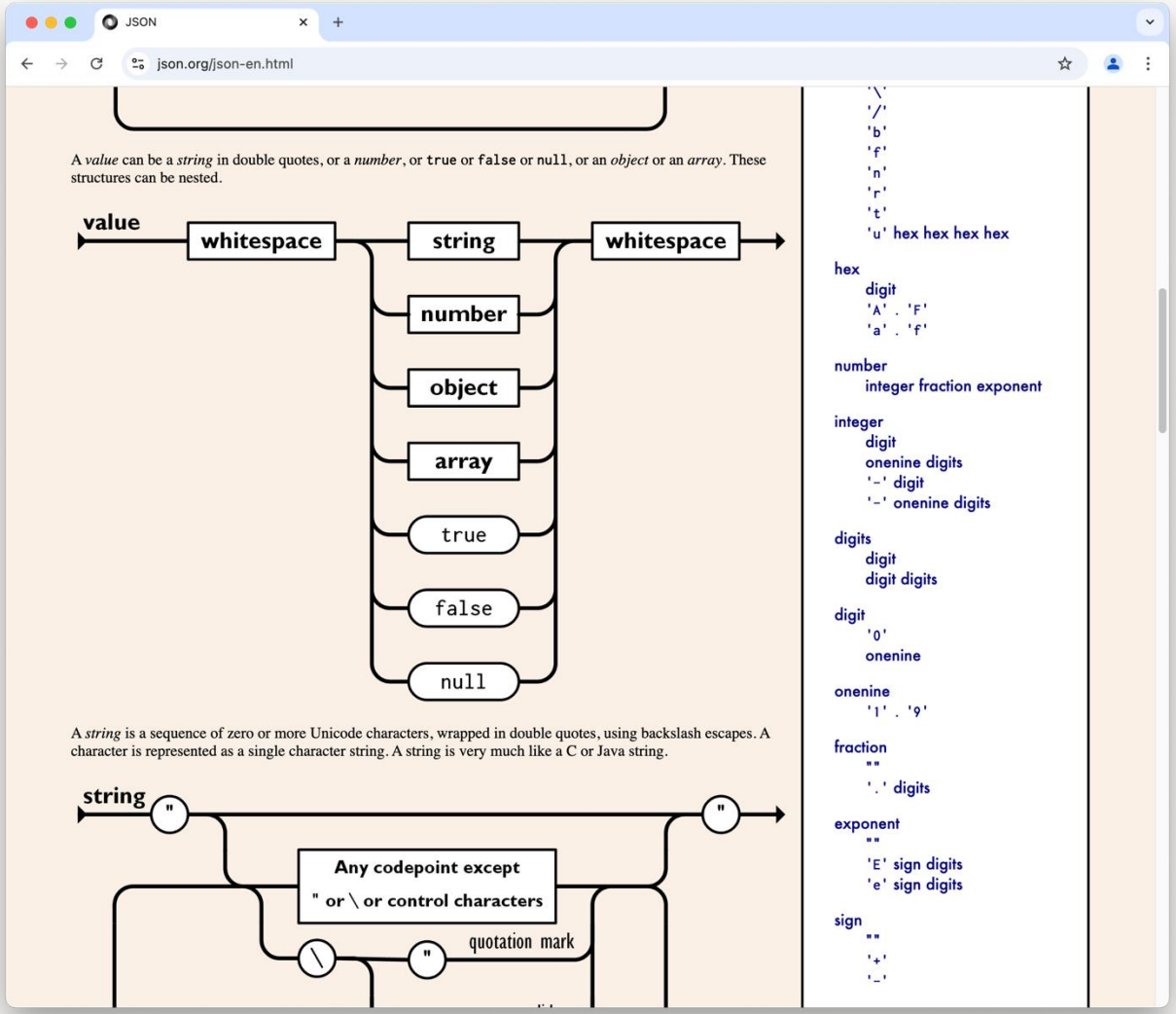
Sealed classes are nominal sum types, e.g.

- A shape is a square or a circle

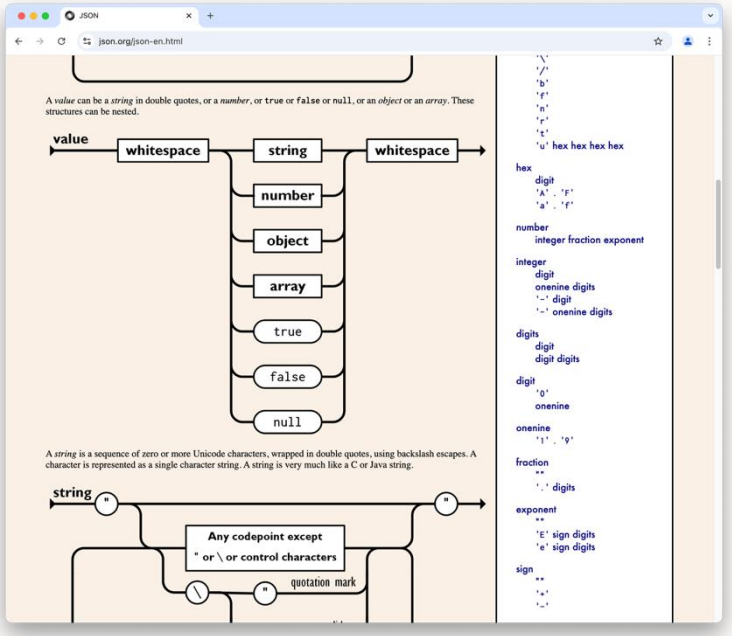
Use records and sealed classes to model the data

Use **pattern matching** to process this data

ADTs are everywhere



ADTs are everywhere

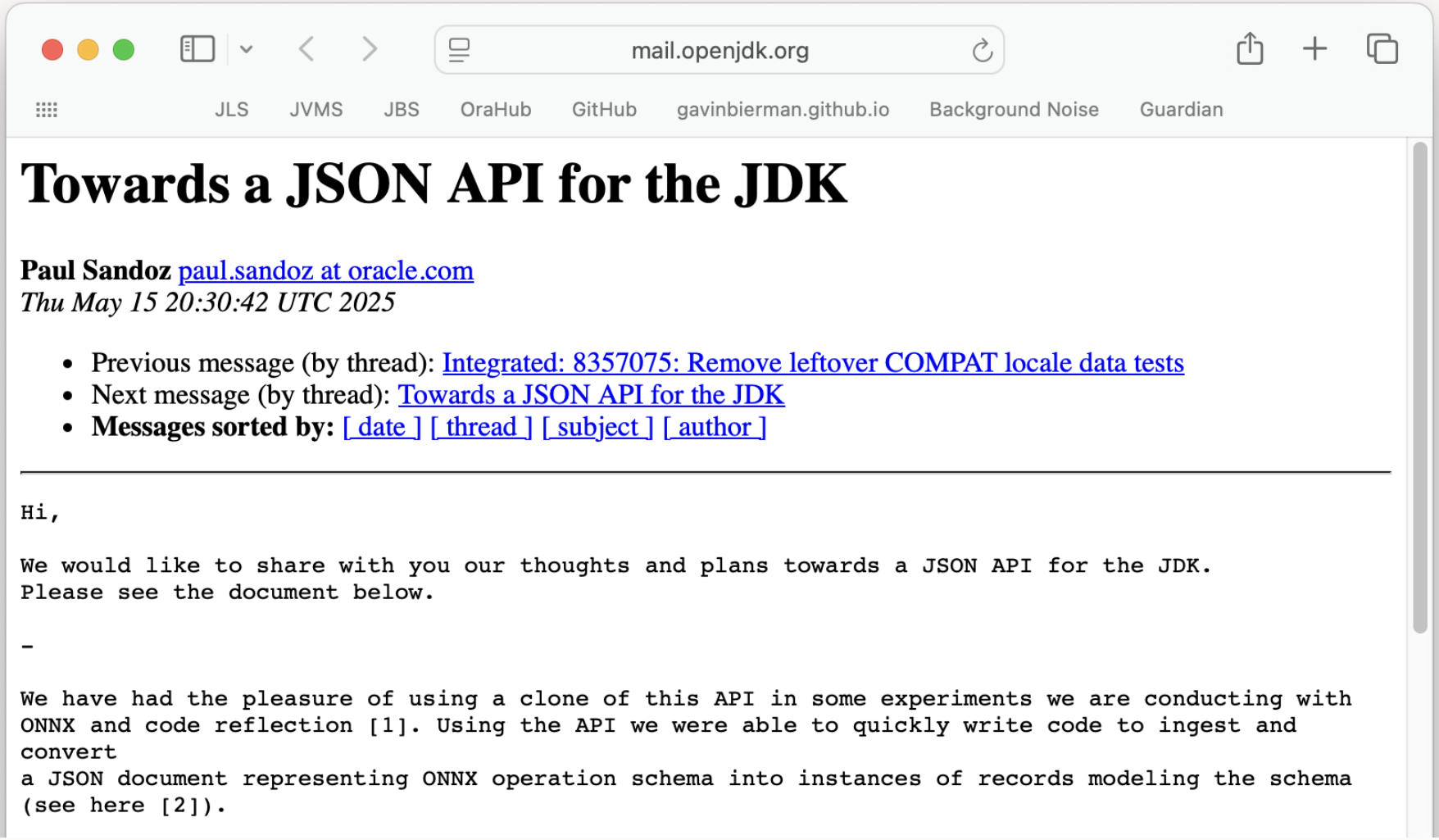



```
sealed interface JsonValue {
    record JsonString(String s)                        implements JsonValue { }
    record JsonNumber(double d)                        implements JsonValue { }
    record JsonNull()                                  implements JsonValue { }
    record JsonBoolean(boolean b)                      implements JsonValue { }
    record JsonArray(List<JsonValue> values)           implements JsonValue { }
    record JsonObject(Map<String, JsonValue> pairs)    implements JsonValue { }
}
```

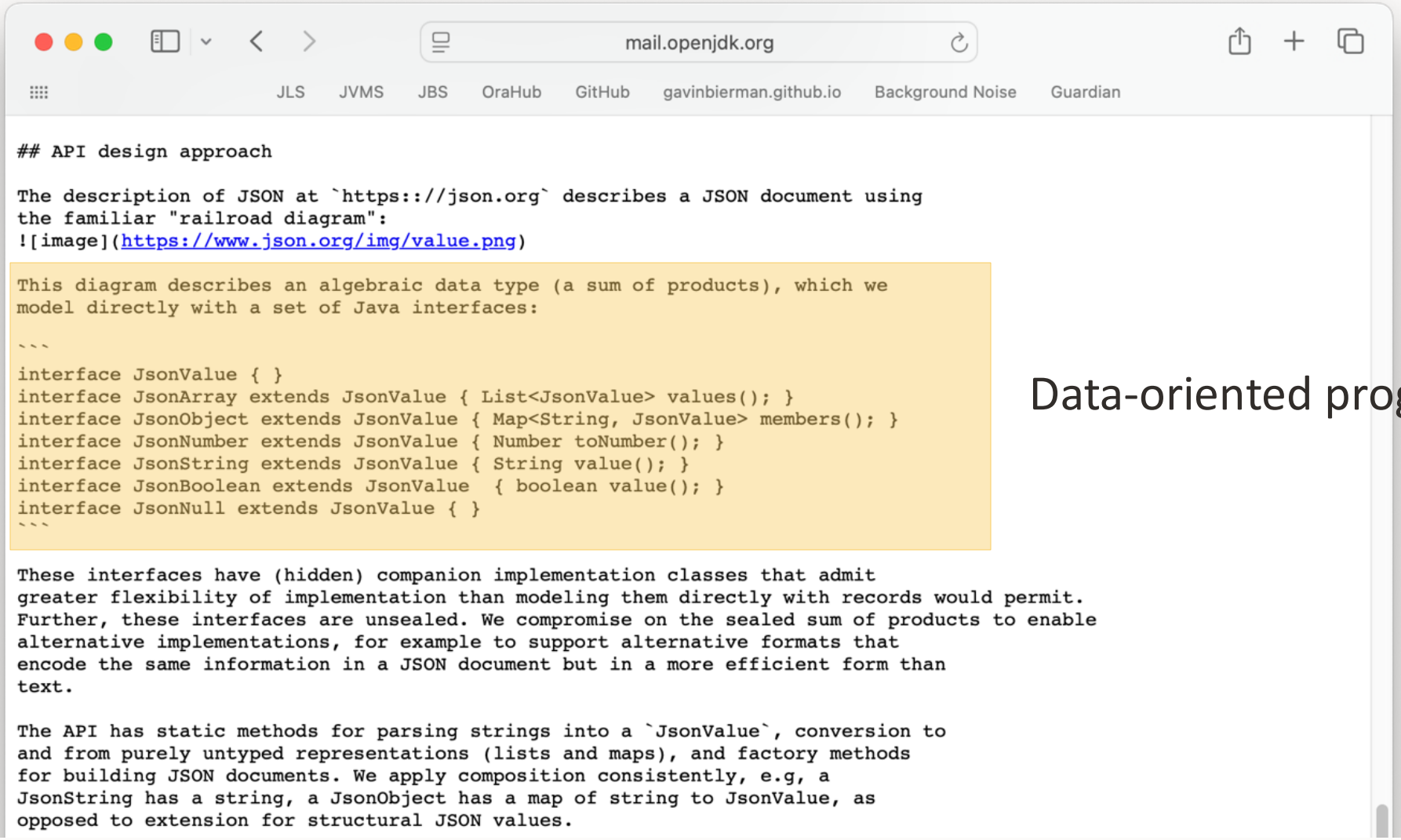
Processing JSON with pattern matching

```
JsonValue j;

if (j instanceof JsonObject(var pairs)
        && pairs.get("name") instanceof JsonString(String name)
        && pairs.get("age")  instanceof JsonNumber(int age)
        && pairs.get("city") instanceof JsonString(String city)) {

    // use name, age, city

}
```


Patterns & API design



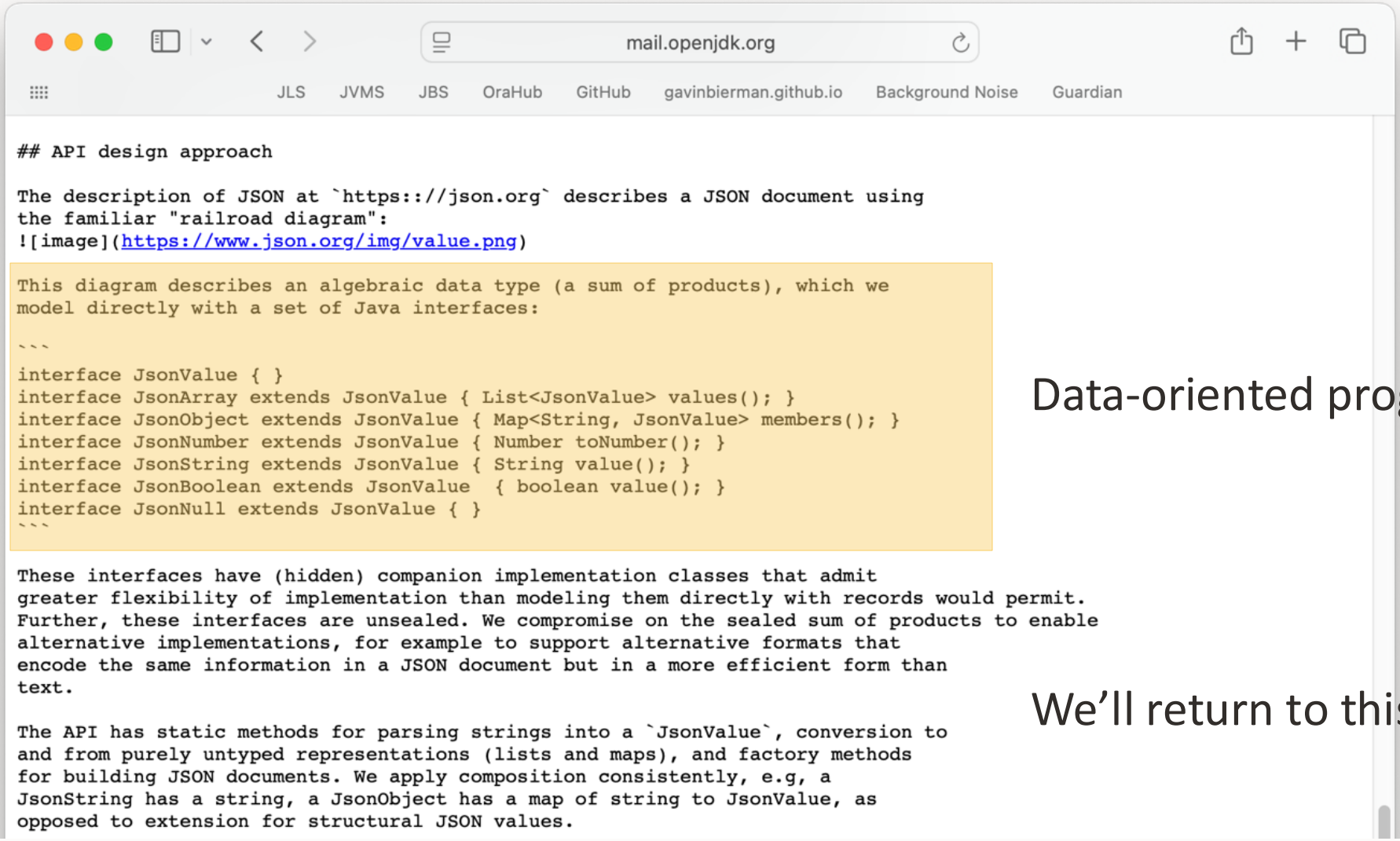
Patterns & API design




```
## API design approach

The description of JSON at `https:://json.org` describes a JSON document using
the familiar "railroad diagram":
![image](https://www.json.org/img/value.png)

This diagram describes an algebraic data type (a sum of products), which we
model directly with a set of Java interfaces:

```
interface JsonValue { }
interface JsonArray extends JsonValue { List<JsonValue> values(); }
interface JsonObject extends JsonValue { Map<String, JsonValue> members(); }
interface JsonNumber extends JsonValue { Number toNumber(); }
interface JsonString extends JsonValue { String value(); }
interface JsonBoolean extends JsonValue  { boolean value(); }
interface JsonNull extends JsonValue { }
```

These interfaces have (hidden) companion implementation classes that admit
greater flexibility of implementation than modeling them directly with records would permit.
Further, these interfaces are unsealed. We compromise on the sealed sum of products to enable
alternative implementations, for example to support alternative formats that
encode the same information in a JSON document but in a more efficient form than
text.

The API has static methods for parsing strings into a `JsonValue`, conversion to
and from purely untyped representations (lists and maps), and factory methods
for building JSON documents. We apply composition consistently, e.g, a
JsonString has a string, a JsonObject has a map of string to JsonValue, as
opposed to extension for structural JSON values.
```

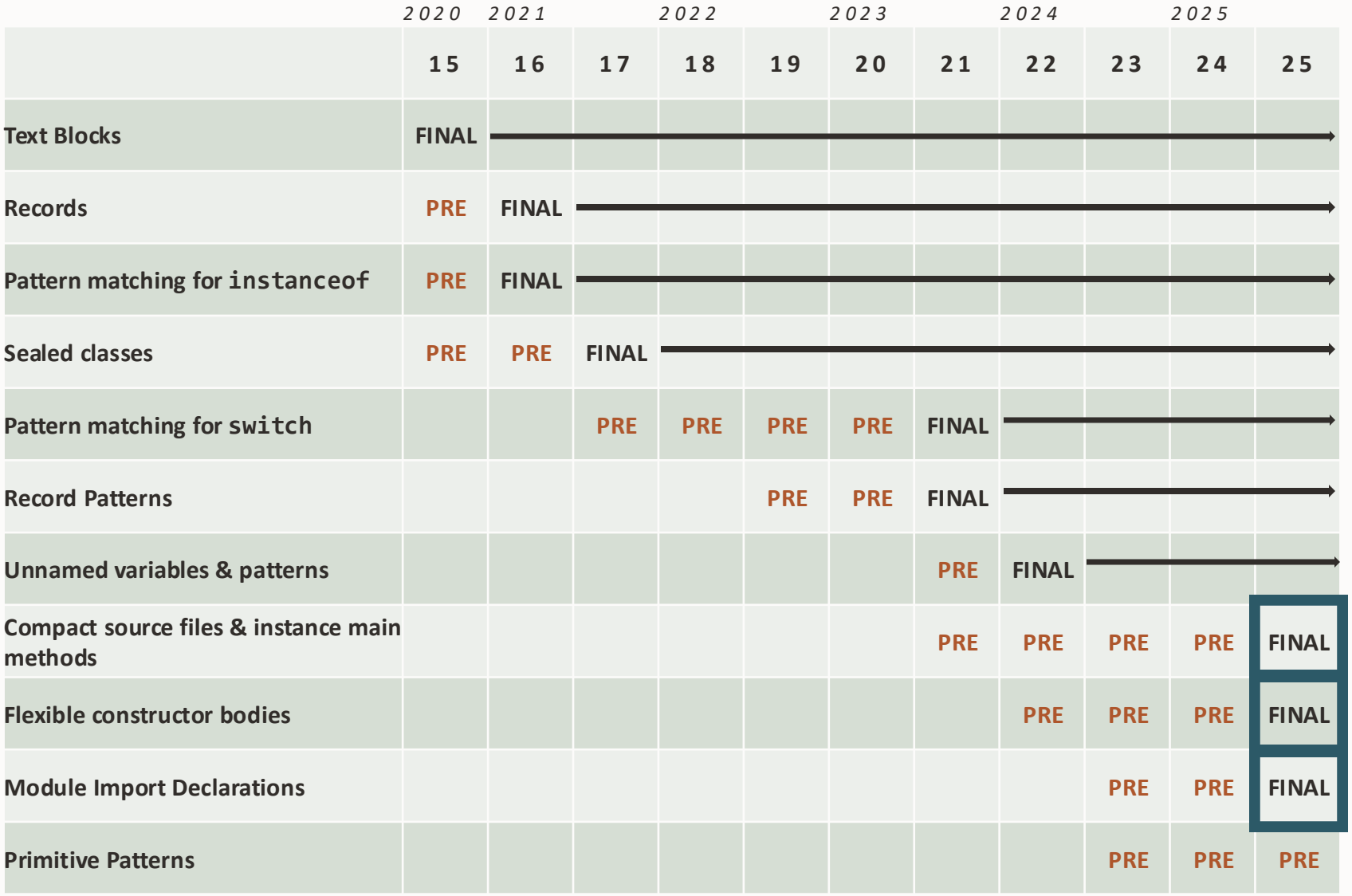
Data-oriented programming!

Patterns & API design



Data-oriented programming!

We'll return to this topic later!



JEP 512: Compact Source Files and Instance main Methods
JEP 511: Module Import Declarations

Together, these JEPs form the “on–ramp”

Ron will discuss these later!

JEP 513: Flexible Constructor Bodies

The constructors of a class are responsible for creating valid instances of that class.

With subclassing, the constructors of superclasses and subclasses share responsibility for building valid instances. Consider:

```
class Employee extends Person
```

Observations:

- `Employee` constructor can refer to fields declared in `Person`
- To maintain safety, we can only allow the `Employee` constructor to access those fields after the `Person` constructor has finished assigning values to them.
- Integrity principle: Classes get first responsibility for their own fields.

JEP 513: Flexible Constructor Bodies

Java ensures this safety by requiring that constructors run **top down**:

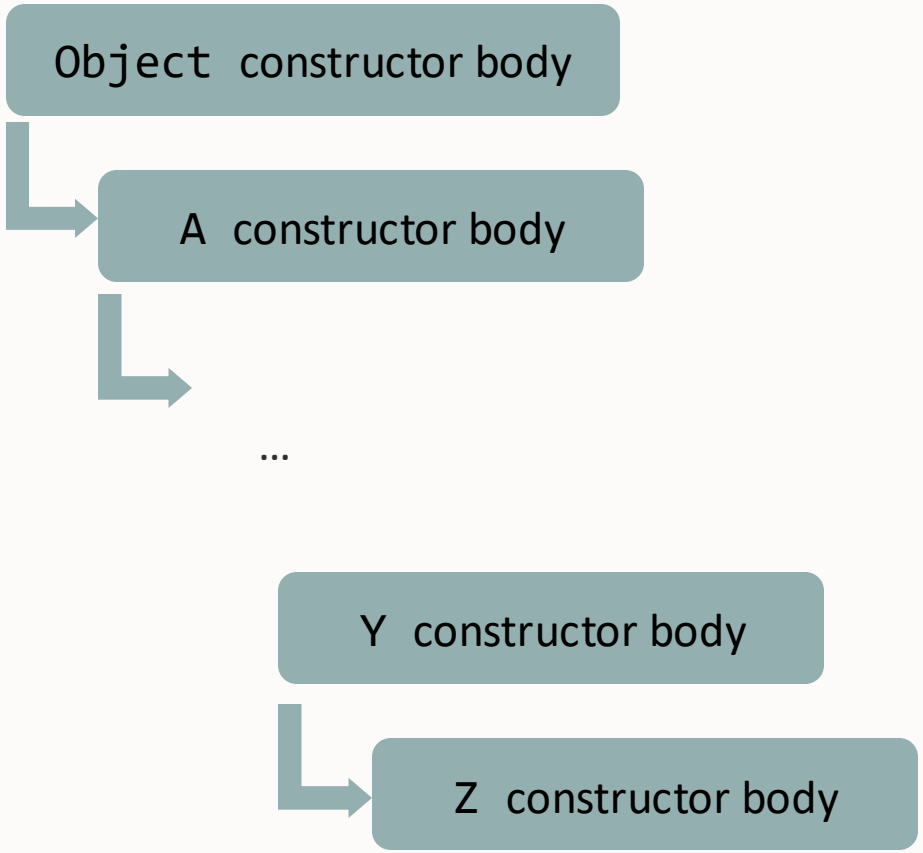
- A constructor in a superclass must run first, ensuring the validity of the fields declared in the superclass, before a constructor in a subclass runs.
- For example: `Person` constructor before `Employee` constructor

How?

- If you invoke a constructor (`super(…)` or `this(…)`) it must be the *first thing* in the constructor body
- If you don't, the compiler will insert `super()` as the first thing in a constructor body

JEP 513: Flexible Constructor Bodies

```
class Object { Object() { … } }
class A extends Object {
    A() { super();
        …
    }
}
…
class Z extends Y {
    Z() { super();
        …
    }
}
new Z();
```



Constructors are too restrictive

```
class Person {
    int age;
    Person(..., int age) {
        if (age < 0)
            throw new IllegalArgumentException(...);
        this.age = age;
    }
    ...
}
class Employee extends Person {
    Employee(..., int age) {
        super(..., age);        // Potentially unnecessary work ☹
        if (age < 18 || age > 67)
            throw new IllegalArgumentException(...);
    }
    ...
}
```


Constructors are too restrictive

```
class Person {
    int age;
    Person(..., int age) {
        if (age < 0)
            throw new IllegalArgumentException(...);
        this.age = age;
    }
    ...
}
class Employee extends Person {
    Employee(..., int age) {
        super(..., verifyAge(age));        // Inelegant hack ☹
    }
    ...
}
```


Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
jshell> new Employee(42, "CAM-FORA");
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

```
jshell> new Employee(42, "CAM-FORA");
Age: 42
Office: null
$3 ==> Employee@446cdf90
```

Superclass Constructors Can Violate Integrity

What happened?

- Integrity of `Employee` class has been violated!
- (Even if the field was `final`)

Item 19 of Effective Java says “Constructors must not invoke overrideable methods”

But nothing in the language to stop this ☹

Constructors

In conclusion:

- Constructor invocation must come first/Top-down execution limits expressiveness of constructors
- Superclasses can violate the integrity of their subclasses

Can we do better?

- More expressiveness
- Defend integrity
- No changes to the JVM 🤞

JEP 513: Flexible Constructor Bodies

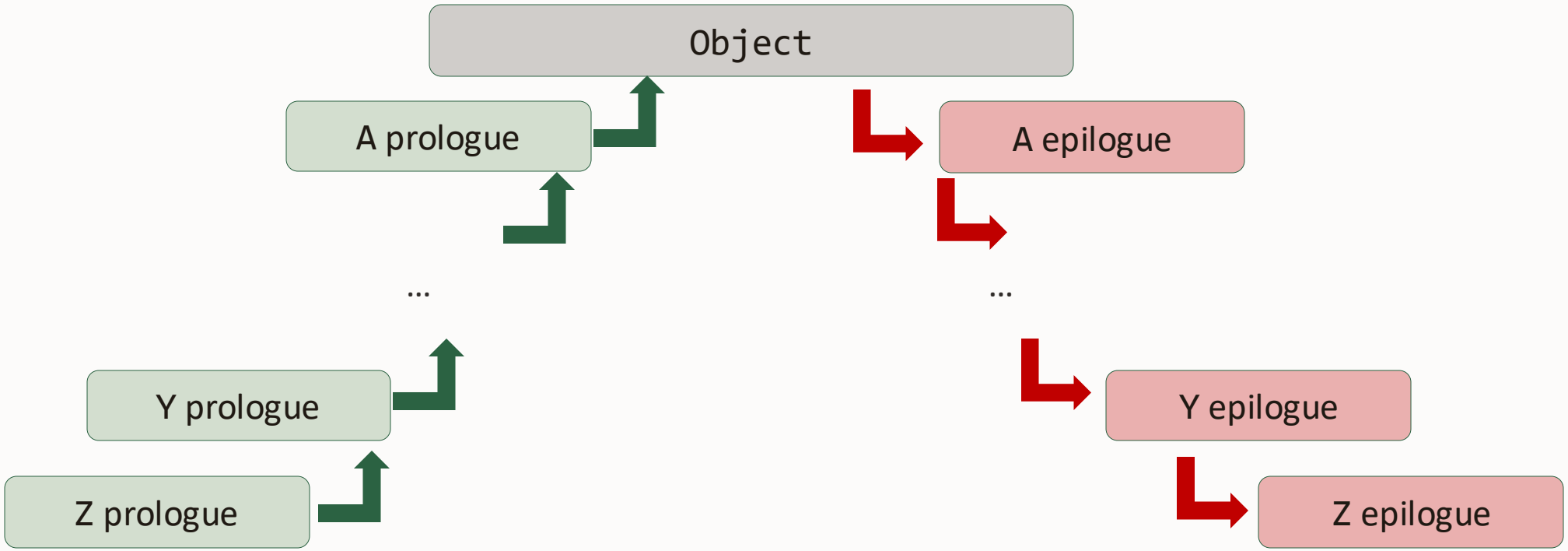
```
class Foo {

    Foo() {
        …
        super();
        …
    }
}
```

Prologue (limited)

Epilogue (unlimited)

JEP 482: Flexible Constructor Bodies

`new Z()` call-graph



Flexible constructor bodies

What’s allowed in the prologue?

Approximate answer:

- Any code that doesn’t use the current object being constructed

Recall this example?

```
class Person {
    int age;
    Person(..., int age) {
        if (age < 0)
            throw new IllegalArgumentException(...);
        this.age = age;
    }
    ...
}
class Employee extends Person {
    Employee(..., int age) {
        super(..., age);        // Potentially unnecessary work ☹
        if (age < 18 || age > 67)
            throw new IllegalArgumentException(...);
    }
    ...
}
```


Recall this example? Fixed!

```
class Person {
    int age;
    Person(..., int age) {
        if (age < 0)
            throw new IllegalArgumentException(...);
        this.age = age;
    }
    ...
}
class Employee extends Person {
    Employee(..., int age) {
        if (age < 18 || age > 67)
            throw new IllegalArgumentException(...);
        super(..., age); // ☺
    }
    ...
}
```


Flexible constructor bodies

What’s allowed in the prologue?

Approximate answer:

- Any code that doesn’t use the current object being constructed
  AND Allowed to write to fields declared in our class

Recall this example?

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        super(..., age);        // ☹
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
    }
    ...
}
```

Recall this example?

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        if (age < 18  || age > 67)
            throw new IAException(...);
        super(..., age);        // ☺
        this.officeID = officeID;
    }
    ...
}
```

Recall this example?

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        if (age < 18  || age > 67)
            throw new IAException(...);
        super(..., age);        // ☺
        this.officeID = officeID;
    }
    ...
}
```

Recall this example? Fixed!

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
        super(..., age);        // ☺
    }
    ...
}
```

Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
jshell> new Employee(42, "CAM-FORA");
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
        super(..., age);        // ☺
    }
    ...
}
```

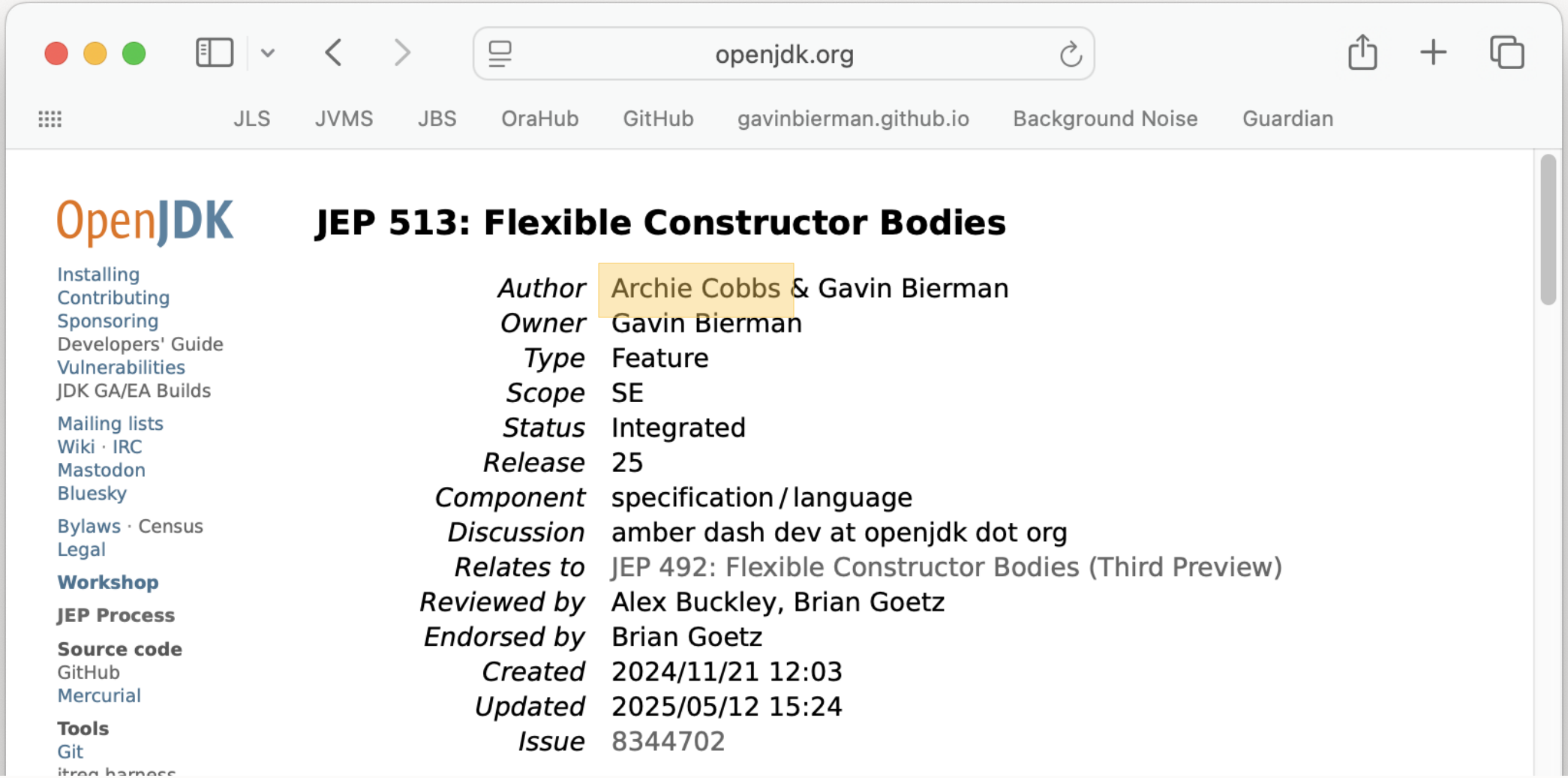
Superclass Constructors Can Violate Integrity

```
class Person {
    int age;
    void show() {
        System.out.println("Age: " + this.age);
    }
    Person(..., int age) {
        if (age < 0)
            throw new IAException(...);
        this.age = age;
        show();
    }
    ...
}
```

```
class Employee extends Person {
    String officeID;
    @Override
    void show() {
        System.out.println("Age: " + this.age);
        System.out.println("Office: " + this.officeID);
    }
    Employee(..., int age, String officeID) {
        if (age < 18  || age > 67)
            throw new IAException(...);
        this.officeID = officeID;
        super(..., age);        // ☹
    }
    ...
}
```

```
jshell> new Employee(42, "CAM-FORA");
Age: 42
Office: CAM-FORA
$3 ==> Employee@446cdf90
```

JEP 513: Thank you Archie!



	2020	2021	2022			2023		2024		2025	
	15	16	17	18	19	20	21	22	23	24	25
Text Blocks	FINAL	→	→	→	→	→	→	→	→	→	→
Records	PRE	FINAL	→	→	→	→	→	→	→	→	→
Pattern matching for `instanceof`	PRE	FINAL	→	→	→	→	→	→	→	→	→
Sealed classes	PRE	PRE	FINAL	→	→	→	→	→	→	→	→
Pattern matching for `switch`			PRE	PRE	PRE	PRE	FINAL	→	→	→	→
Record Patterns					PRE	PRE	FINAL	→	→	→	→
Unnamed variables & patterns							PRE	FINAL	→	→	→
Compact source files & instance main methods							PRE	PRE	PRE	PRE	FINAL
Flexible constructor bodies								PRE	PRE	PRE	FINAL
Module Import Declarations									PRE	PRE	FINAL
Primitive Patterns									PRE	PRE	PRE

	2020	2021		2022		2023		2024		2025	
	15	16	17	18	19	20	21	22	23	24	25
Text Blocks	FINAL	→	→	→	→	→	→	→	→	→	→
Records	PRE	FINAL	→	→	→	→	→	→	→	→	→
Pattern matching for `instanceof`	PRE	FINAL	→	→	→	→	→	→	→	→	→
Sealed classes	PRE	PRE	FINAL	→	→	→	→	→	→	→	→
Pattern matching for `switch`			PRE	PRE	PRE	PRE	FINAL	→	→	→	→
Record Patterns					PRE	PRE	FINAL	→	→	→	→
Unnamed variables & patterns							PRE	FINAL	→	→	→
Compact source files & instance main methods							PRE	PRE	PRE	PRE	FINAL
Flexible constructor bodies								PRE	PRE	PRE	FINAL
Module Import Declarations									PRE	PRE	FINAL
Primitive Patterns									PRE	PRE	PRE
String Templates							PRE	PRE	X		

String Templates Background

A common (mis-guided) developer lament:

Q: Why make me write this?

```
String statement = "SELECT * FROM users WHERE name = '" + username + "'";
```

When I want to write this?

```
String statement = "SELECT * FROM users WHERE name = '$username'";
```

“All I want is simple string interpolation!”

String Templates Background

A common (mis-guided) developer lament:

Q: Why make me write this?

```
String statement = "SELECT * FROM users WHERE name = '" + username + "'";
```

When I want to write this?

```
String statement = "SELECT * FROM users WHERE name = '$username'";
```

A: SQL Injection attacks!

When `username` is: `a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't`

ian.sh

ian carroll / Bypassing airport security via SQL injection

Bypassing airport security via SQL injection

08/29/2024

Introduction

Like many, Sam Curry and I spend a lot of time waiting in airport security lines. If you do this enough, you might sometimes see a special lane at airport security called **Known Crewmember** (KCM). KCM is a TSA program that allows pilots and flight attendants to bypass security screening, even when flying on domestic personal trips.

The KCM process is fairly simple: the employee uses the dedicated lane and presents their KCM barcode or provides the TSA agent their employee number and airline. Various forms of ID need to be presented while the TSA agent's laptop verifies the employment status with the airline. If successful, the employee can access the sterile area without any screening at all.

A similar system also exists for cockpit access, called the **Cockpit Access Security System** (CASS). Most aircraft have at least one jumpseat inside the cockpit sitting behind the flying pilots. When pilots need to commute or travel, it is not always possible for them to occupy a revenue seat, so a jumpseat can be used instead. CASS allows the gate agent of a flight to verify that the jumpseater is an authorized pilot. The gate agent can then inform the crew of the flight that the jumpseater was authenticated by CASS.

The employment status check is the most critical component of these processes. If the individual doesn't currently work for an airline, they have not had a background check and should not be permitted to bypass security screening or access the cockpit. This process is also responsible for returning the photo of the crewmember to ensure the right person is being authorized for access. So how does this work, when every airline presumably uses a different system to store their employee information? That is what we were wondering, and where it gets interesting...

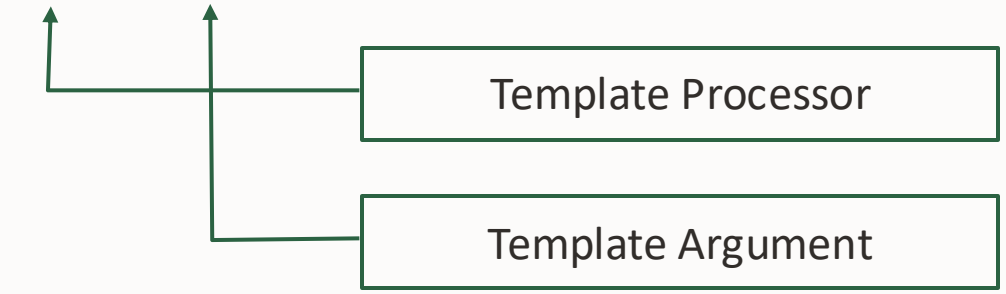
ARINC

ARINC (a subsidiary of Collins Aerospace) appears to be contracted by the TSA to operate the Known Crewmember system. ARINC operates a few central components, including an online website for pilots and flight attendants to check their KCM status, and an API to route authorization requests between different airlines. Each airline appears to operate their own authorization system to participate in KCM and CASS, and it interacts with the "hub" of ARINC.

String Templates

A new expression form, the **template expression**, e.g.

```
STR."Hello \{firstName}!"
```



A **template processor** is an expression of new type `StringTemplate.Processor`

A **template argument** is either a string literal, a text block, or a **template**.

A **template** looks like a string or a text block but it has \{**embedded expressions**}.

String Templates

```
String firstName = "Bill";
String lastName = "Duck";

String fullName = STR."\{firstName} \{lastName}";                    "Bill Duck”

String sortName = STR."\{lastName}, \{firstName}";                   "Duck, Bill"

int x = 10, y = 20;
String s = STR."\{x} + \{y} = \{x + y}";                             "10 + 20 = 30"

String s = STR."You have a \{getOfferType()} waiting for you!";
                                                  "You have a gift waiting for you!"

String t = STR."Access at \{req.date} \{req.time} from \{req.ipAddress}";
                                                "Access at 2022-03-25 15:34 from 8.8.8.8"
```


String Templates

```
String name = "Joan Smith";
String phone = "555-123-4567";
String address = "1 Maple Drive, Anytown";
String json = STR."""
    {
        "name": "\{name}",
        "phone": "\{phone}",
        "address": "\{address}"
    }
""";
```

```
"""
{
    "name": "Joan Smith",
    "phone": "555-123-4567",
    "address": "1 Maple Drive, Anytown"
}
"""
```


String Templates

```
var JSON = StringTemplate.Processor.of( (StringTemplate st) -> new
    JSONObject(st.interpolate()) );

String name = "Joan Smith";
String phone = "555-123-4567";
String address = "1 Maple Drive, Anytown";

JSONObject json = JSON.""" {
    "name": "\{name}",
    "phone": "\{phone}",
    "address": "\{address}"
} """;
```


String Templates

Key idea of a template:

- It denotes a pair of *string fragments* and *values*

Template processors then process these two lists (not a string) to build a result.

These always go hand-in-hand.

This is core to our design; we don’t want to be responsible for the next thirty years of string injection attacks in Java!

- cf. PEP 501 – General purpose template literal strings in Python
- cf. Safe templating library for building HTML in Go

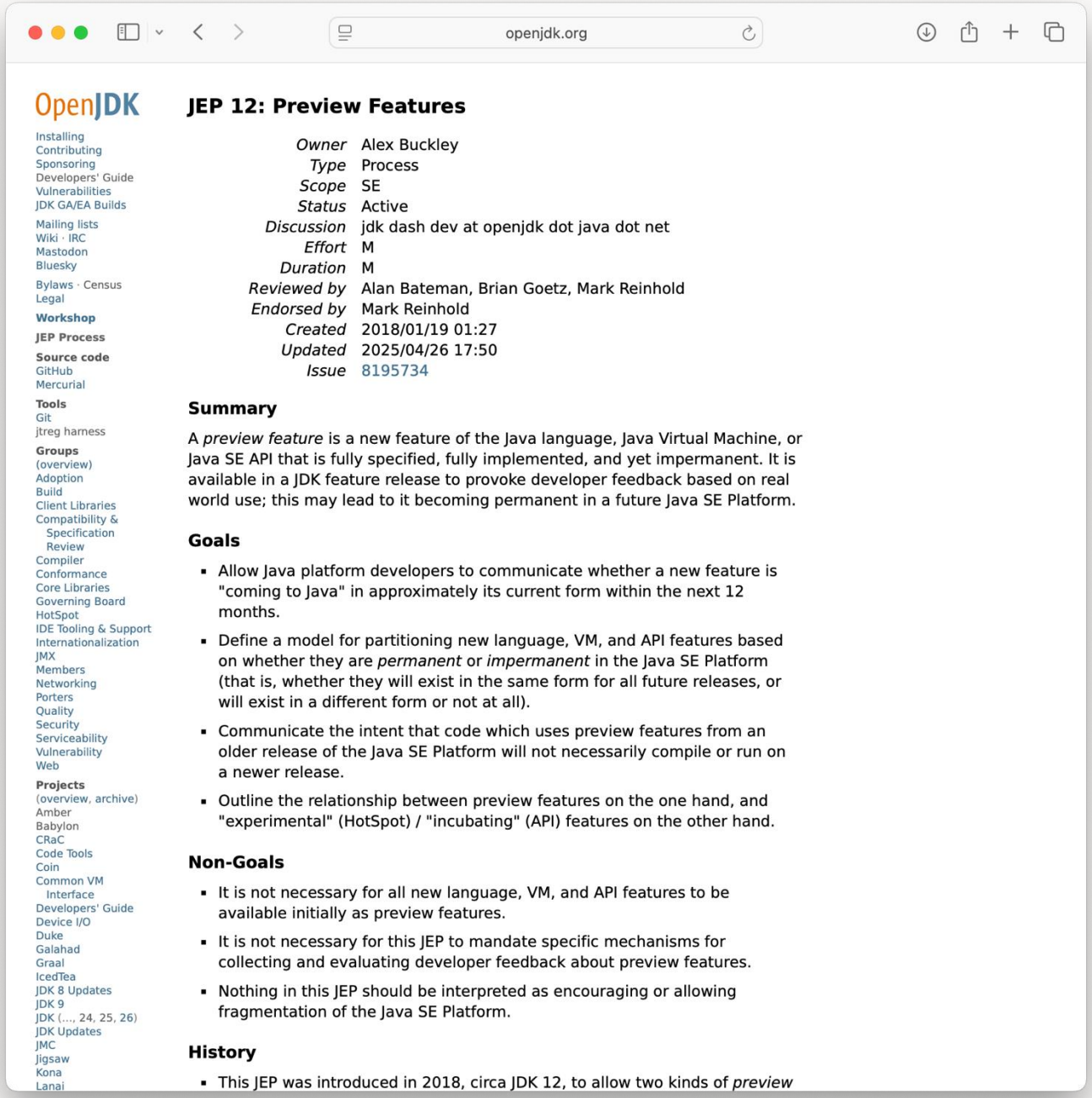
String Templates

But more extensive (internal) experimentation raised concerns:

- Writing non-functional template processors with side-effects was impossible ☹
- Nested templates required explicit application, which made code ugly ☹
- API designers faced a new ambiguity: processors or methods? ☹
- 3rd party processors couldn't play the performance game in the same way JDK processors could ☹

Conclusions:

- Serious doubt about the design ☹



Eventually, the JEP owner must decide the preview feature's ultimate fate. If the decision is to remove the preview feature, then the owner must file an issue in JBS to remove the feature in the next JDK feature release; no new JEP is needed. On the other hand, if the decision is to finalize, then the owner must file a new JEP, noting refinements informed by developer feedback. The title of this JEP should be the feature's name, omitting the earlier suffix of `(Preview)` / `(Second Preview)`, and without adding any new suffix such as `(Standard)` or `(Final)`. This JEP will ultimately reach Targeted status for the next JDK feature release.

Eventually, the JEP owner must decide the preview feature's ultimate fate. If the decision is to remove the preview feature, then the owner must file an issue in JBS to remove the feature in the next JDK feature release; no new JEP is needed. On the other hand, if the decision is to finalize, then the owner must file a new JEP, noting refinements informed by developer feedback. The title of this JEP should be the feature's name, omitting the earlier suffix of `(Preview)` / `(Second Preview)`, and without adding any new suffix such as `(Standard)` or `(Final)`. This JEP will ultimately reach Targeted status for the next JDK feature release.

- We removed the feature in JDK 23
- This is the preview/JEP process **working perfectly**!
- We are actively working on a replacement to this feature!

What's coming in the future?

1. More pattern matching
   - Pattern matching for more classes
   - Pattern-matching aware APIs
   - Pattern matching assignment
   - Constants in patterns
2. Type classes
3. With expressions
4. *Project Valhalla language features*
   - *Value classes*
   - *Nullity markers on reference types*

There are many others but… 🤫

Thank you!