OZON tech

Продвинутые системы типов. Чего ещё мне не хватает в Java из Rust

Пётр Портнов
Старший разработчик поисковой платформы

Привет!

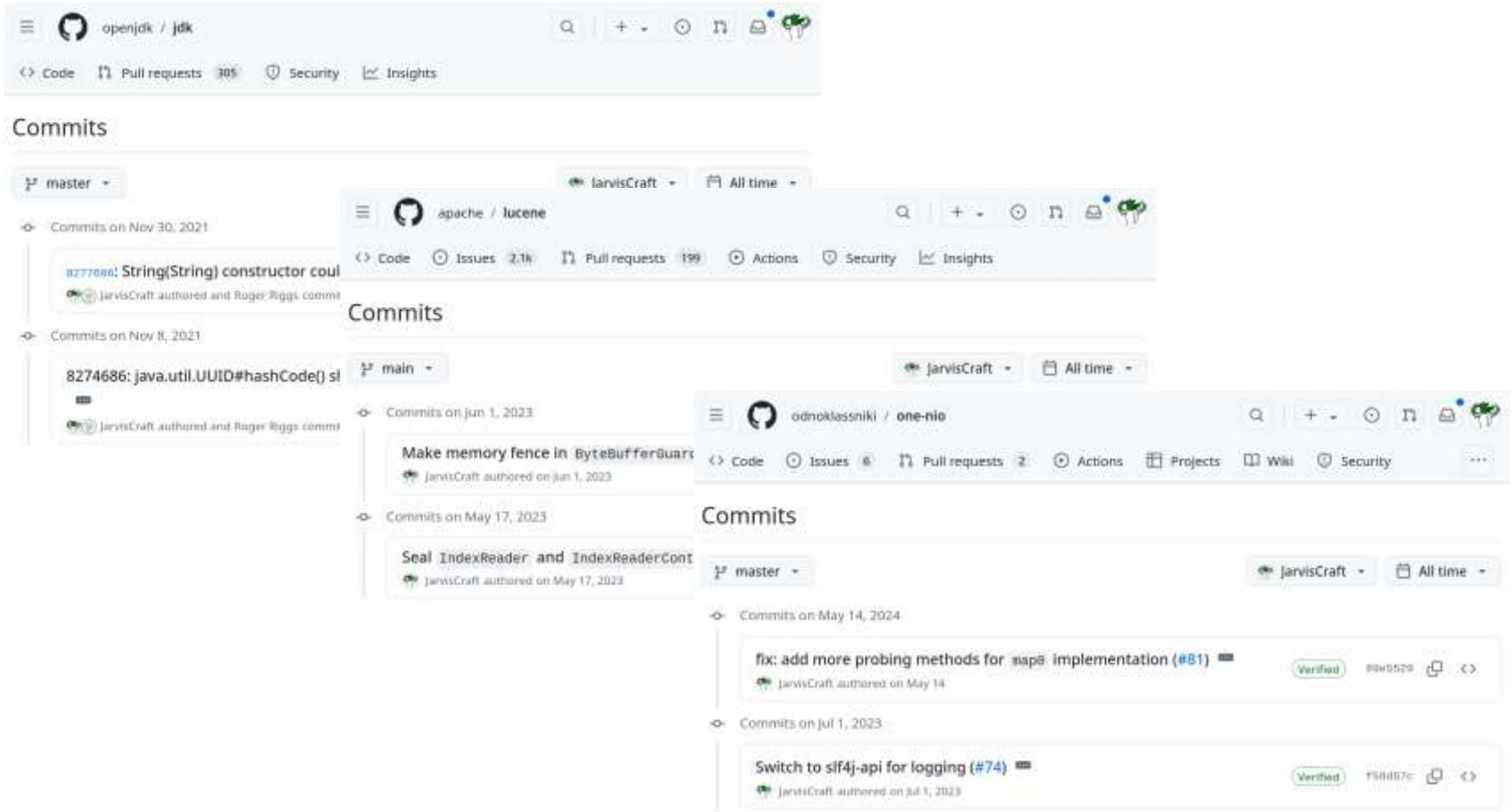
Я Петя Портнов

Старший разработчик в команде поисковой платформы Ozon

02:31

Опенсорс — наше всё

- OpenJDK
- lucene
- one-nio
- flipperzero & flipperzero-rs
- nixpkgs
- jrsonnet



AGENDA

01 Введение в теорию типов

02 ADT — алгебраические типы данных

03 Применение продвинутых типов

04 Экзотические типы

05 Заключение

01

~~Введение в теорию типов~~
Как мы усложняем себе жизнь!

Монада — это...


НЕТ!

Кто любит прекондишены?

```
public static List<Long> topK(
    Iterable<Long> values,
    int k
) {
    ...
}
```

Что может сделать плохой клиент?

```
topK(List.of(1, 2, 100, 2, 3, 5, 17), -1)
topK(null, 100)
```

Что нужно проверить?

```
public static List<Long> topK(
    Iterable<Long> values,
    int k
) {
    Preconditions.checkArgument(values != null,
        "values should be non-null"
    );
    Preconditions.checkArgument(k > 0,
        "k should be positive"
    );
    ...
}
```

Но ведь...

Java

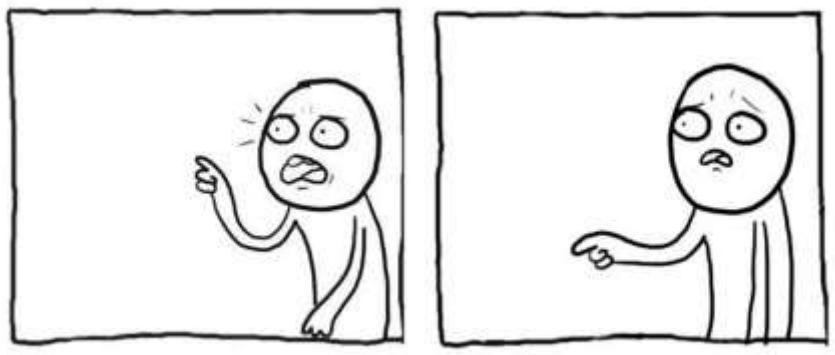
Paradigm	Multi-paradigm: generic, object-oriented (class-based), functional, imperative, reflective, concurrent
Designed by	James Gosling
Developer	Oracle Corporation
First appeared	May 23, 1995; 29 years ago[1]
Typing discipline	Static, strong, safe, nominative, manifest
Memory management	Automatic garbage collection
Filename extensions	.java, .class, .jar, .jmod

Воспользуемся этим

```
public static List<Long> topK(
    Iterable<Long> values,
    unsigned int k
) {
    Preconditions.checkArgument(k != 0,
        "k should be non-zero"
    );
    ...
}
```

Я сказал на Джаве!

```
public static List<Long> topK(
    @NotNull Iterable<Long> values,
    UnsignedInt k
) {
    Preconditions.checkArgument(k.nonZero(),
        "k should be non-zero"
    );
    ...
}
```



Свой unsigned

```
public record UnsignedInt(int value) {
    public UnsignedInt {
        Preconditions.checkArgument(value >= 0,
            "value %s should be non-negative", value
        );
    }
}
```


Always has been
Wait, it’s all Preconditions?

Как теперь выглядит клиент?

```
topK(
    List.of(1, 2, 100, 2, 3, 5, 17),
    new UnsignedInt(-1)
)
topK(
    null,
    new UnsignedInt(100)
)
```

Как теперь выглядит клиент?

```
topK(
    List.of(1, 2, 100, 2, 3, 5, 17),
    new UnsignedInt(-1) // ⚠️
)
topK(
    null,
    new UnsignedInt(100)
)
```

Как теперь выглядит клиент?

```
topK(
    List.of(1, 2, 100, 2, 3, 5, 17),
    new UnsignedInt(-1) // 🔥
)
topK(
    null,
    new UnsignedInt(100)
)
```

Обёртка UnsignedInt

Ну и зачем?

01 **Не может существовать с отрицательным значением**

Обёртка UnsignedInt

Ну и зачем?

01 **Не может существовать с отрицательным значением**

02 **Если существует, то гарантированно правильный**

На UnsignedInt могут быть свои операции

```
public record UnsignedInt(int value) {
    ...

    public UnsignedInt add(int other) {
        return new UnsignedInt(value + other);
    }
}
```

Клиент получит ошибку там, где её причина

```
UnsignedInt k = getMin();
k = k.add(Integer.MAX_VALUE); // 🔥

topK(myNums, k)
```

Некоторые операции всегда валидны

```
public record UnsignedInt(int value) {
    ...

    public UnsignedInt mul(UnsignedInt other) {
        return new UnsignedInt(value * other.value);
    }
}
```

Обёртка UnsignedInt

Ну и зачем?

01 **Не может существовать с отрицательным значением**

02 **Если существует, то гарантированно правильный**

03 **Проблемы локализуются**

Обёртка UnsignedInt

Ну и зачем?

01 **Не может существовать с отрицательным значением**

02 **Если существует, то гарантированно правильный**

03 **Проблемы локализуются**

Парсите

Не

Валидируйте

Вообще-то там остался ещё один Precondition

```
public static List<Long> topK(
    @NotNull Iterable<Long> values,
    UnsignedInt k
) {
    Preconditions.checkArgument(k.nonZero(),
        "k should be non-zero"
    );
    ...
}
```

Ноль проблем

```
public record PositiveInt(int value) {
    public PositiveInt {
        Preconditions.checkArgument(value > 0,
            "value %s should be positive", value
        );
    }
}
```

Ноль проблем

```
public record PositiveInt(int value) {
    public PositiveInt(UnsignedInt value) {
        Preconditions.checkArgument(value.nonZero(),
            "value %s should be positive", value
        );
        this(value);
    }
}
```

Один проблем

```
|  Error:
|  constructor is not canonical, so its first statement must invoke another
constructor of class PositiveInt
|      public PositiveInt(UnsignedInt value) {
|      ^-----------------------------------...
```

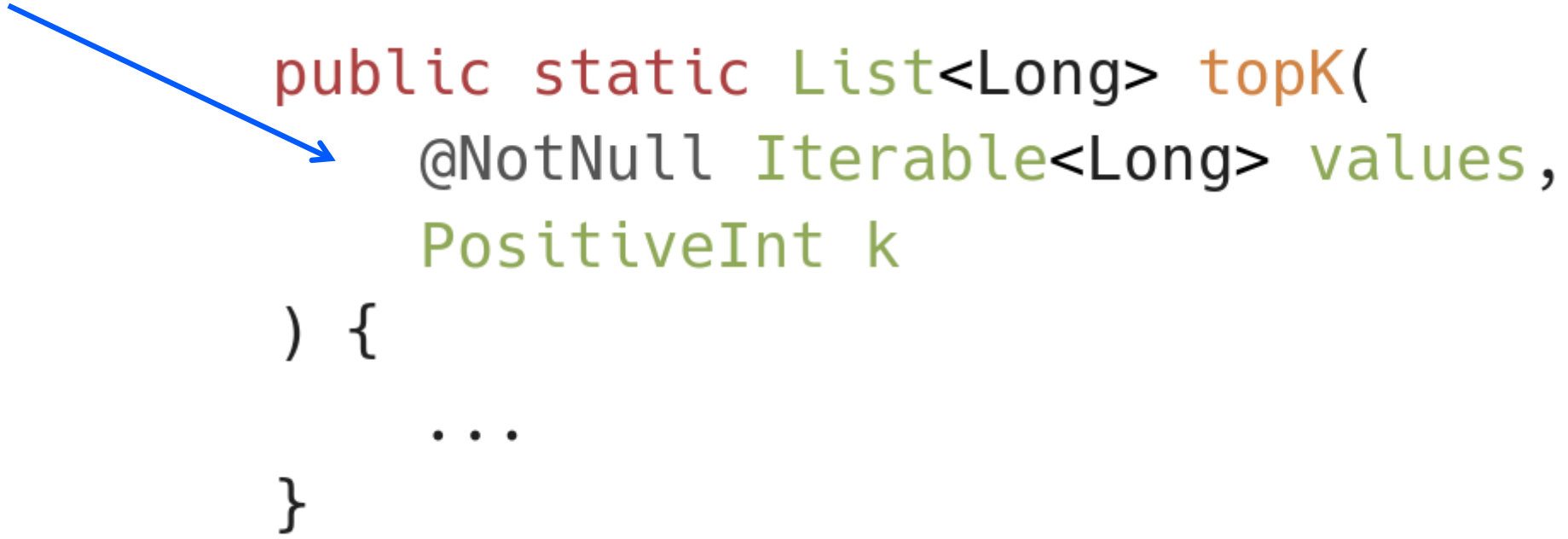
JEPы спешат на помощь!

JEP 447: Statements before super(...) (Preview)

https://openjdk.org/jeps/447

И мы снова живём без прекондишенов

Но остался ОН



```
public static List<Long> topK(
    @NotNull Iterable<Long> values,
    PositiveInt k
) {
    ...
}
```

Давайте сделаем как и раньше!

```
public record NonNull<T>(T value) {
    public NonNull {
        Preconditions.checkArgument(value != null
            "value should not be null"
        );
    }
}
```

Прекрасно!

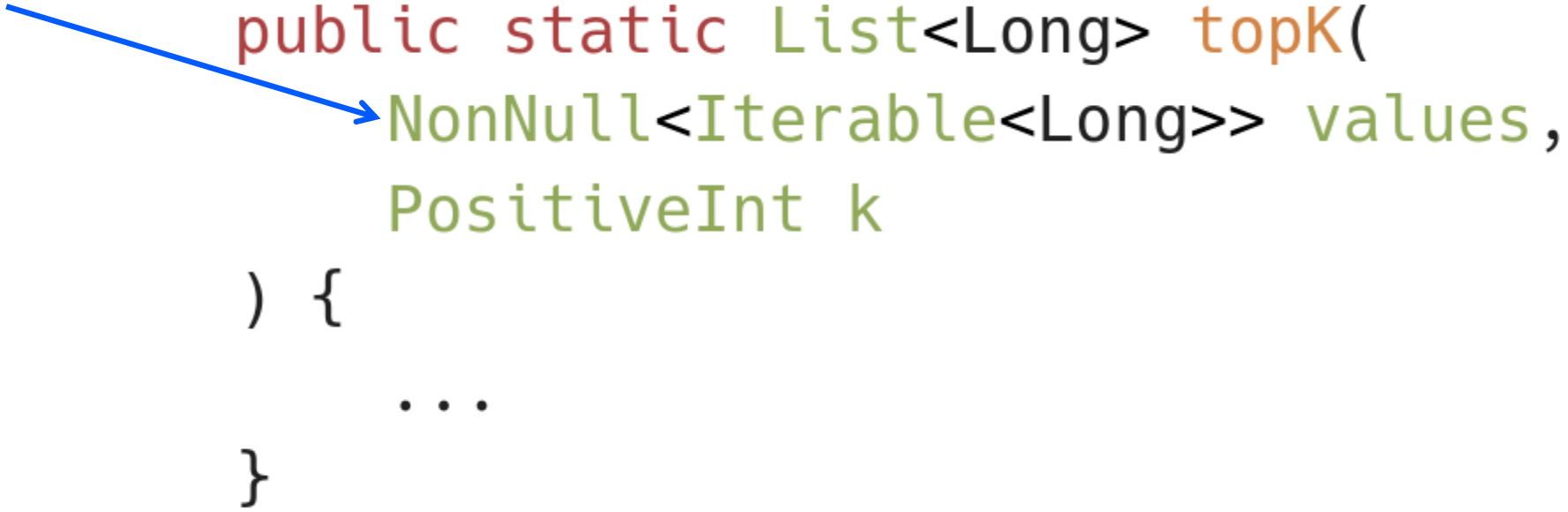
```
public static List<Long> topK(
    NonNull<Iterable<Long>> values,
    PositiveInt k
) {
    ...
}
```

Что скажет клиент?

```
topK(
    new NonNull(null), // 🔥
    new UnsignedInt(-1)
)
topK(
    null,
    new UnsignedInt(10)
)
```

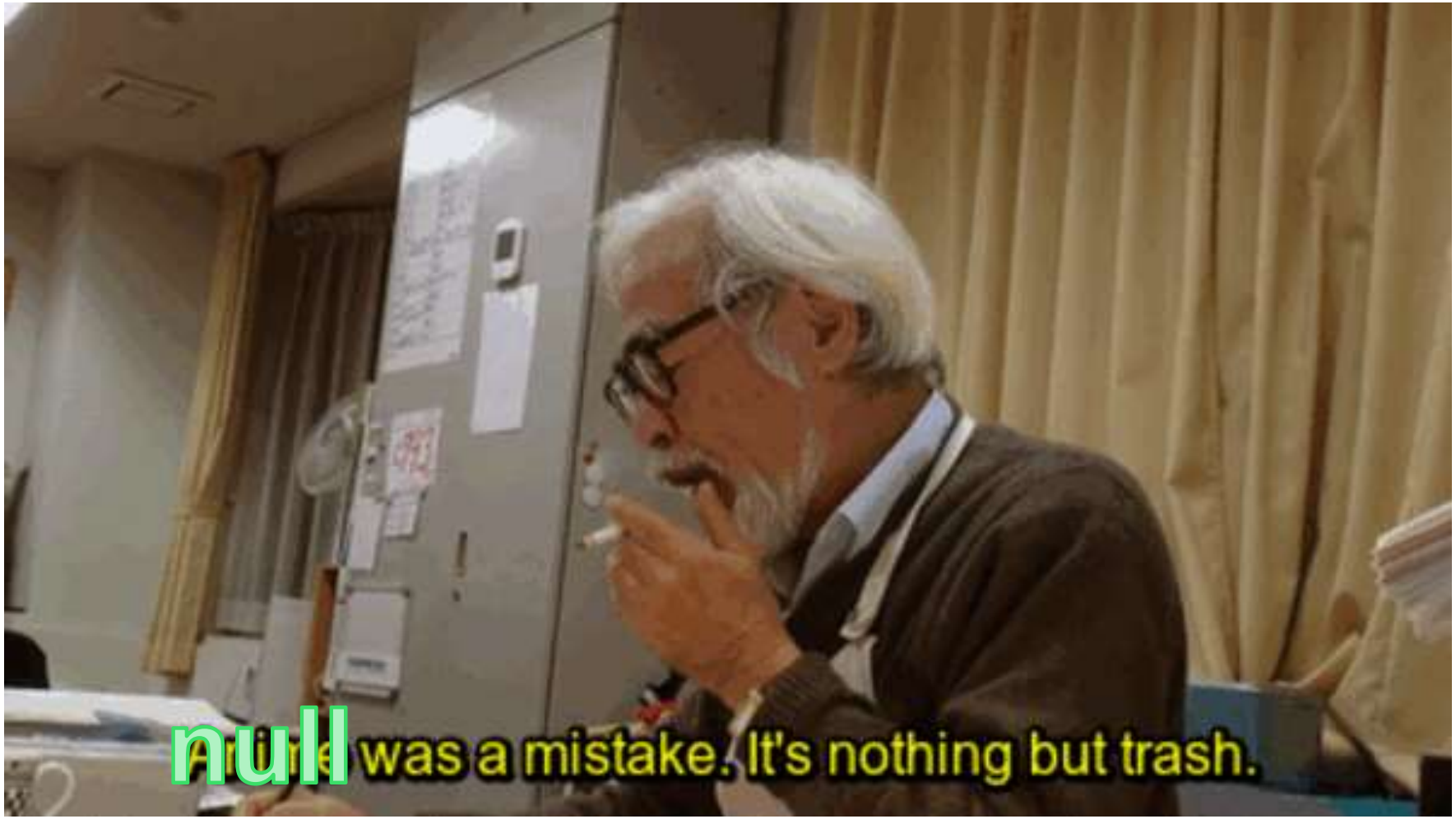
Есть нюанс

Обёртка тоже nullable



```
public static List<Long> topK(
    NonNull<Iterable<Long>> values,
    PositiveInt k
) {
    ...
}
```

null, повсюду null



Есть ограничения системы типов языка, поэтому действуем вопреки, а не благодаря

Как это делают другие языки?


```
fun topK(
    number: Iterable<Long>,
    k: UnsignedInt
): List<Long> {
    ...
}

...

topK(listOf(1), UnsignedInt(3u)
topK(null, UnsignedInt(10u)) // 🚫
```

Как это делают другие языки?

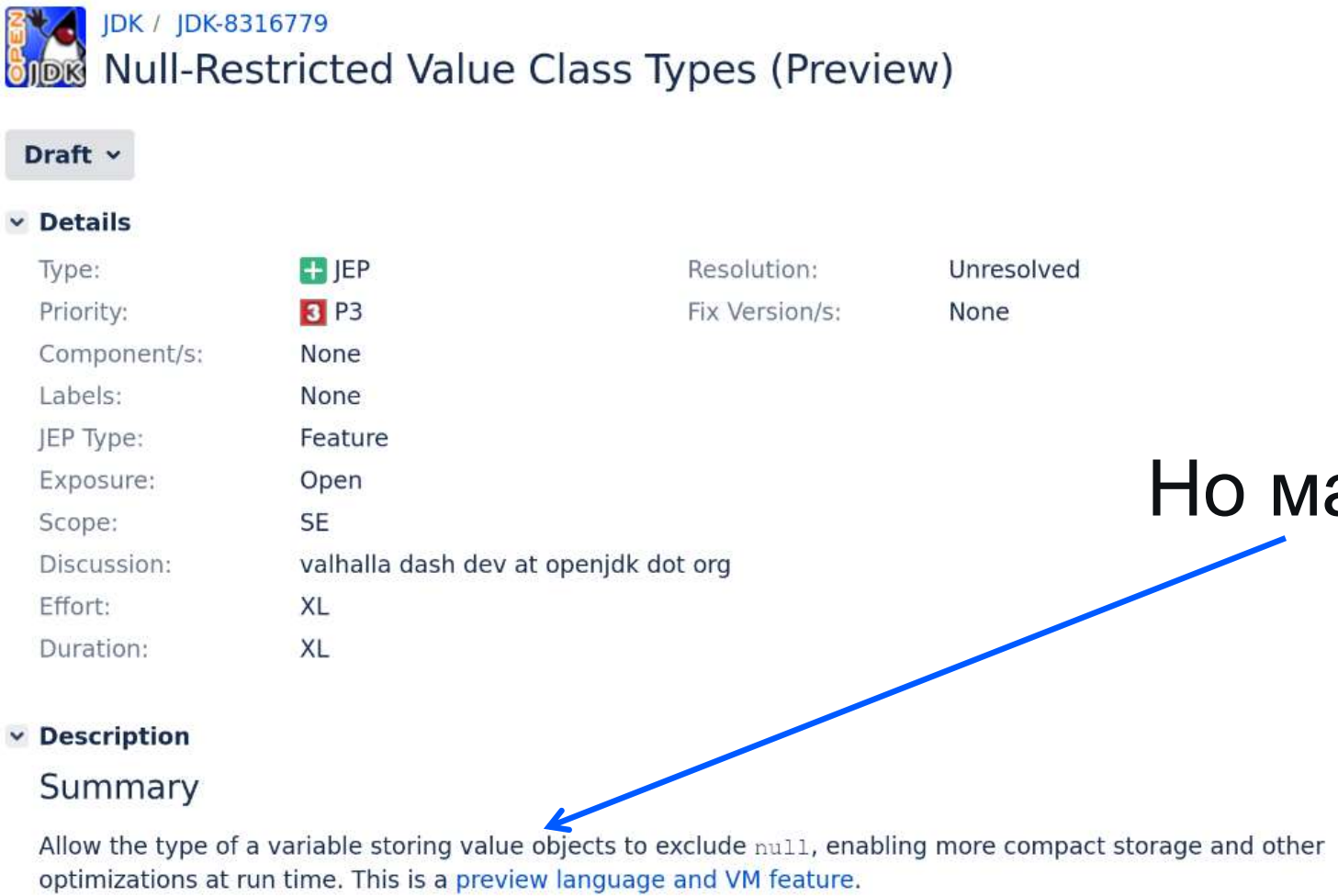

```
Null cannot be a value of a non-null type
'kotlin.collections.Iterable<kotlin.Long>'
```

Что делать?

Считаем

- Что в нашей кодовой базе (или её части) всё по умолчанию **не null**
- Явно «включаем» возможность null’а через @Nullable
- Для внешнего кода определяем дефолт по... доке
  - Если не уверены, то считаем nullable
- И да поможет нам IDE

Надежда есть



JDK / JDK-8316779

Null-Restricted Value Class Types (Preview)

Draft

Details

Type:	JEP	Resolution:	Unresolved
Priority:	P3	Fix Version/s:	None
Component/s:	None		
Labels:	None		
JEP Type:	Feature		
Exposure:	Open		
Scope:	SE		
Discussion:	valhalla dash dev at openjdk dot org		
Effort:	XL		
Duration:	XL		

Description

Summary

Allow the type of a variable storing value objects to exclude `null`, enabling more compact storage and other optimizations at run time. This is a preview language and VM feature.

Но малая

Промежуточные идеи

Описываем смысл нашего кода типами. Парсим, а не валидируем

Поддержка языка важна, как для удобства работы, так и для обеспечения корректности

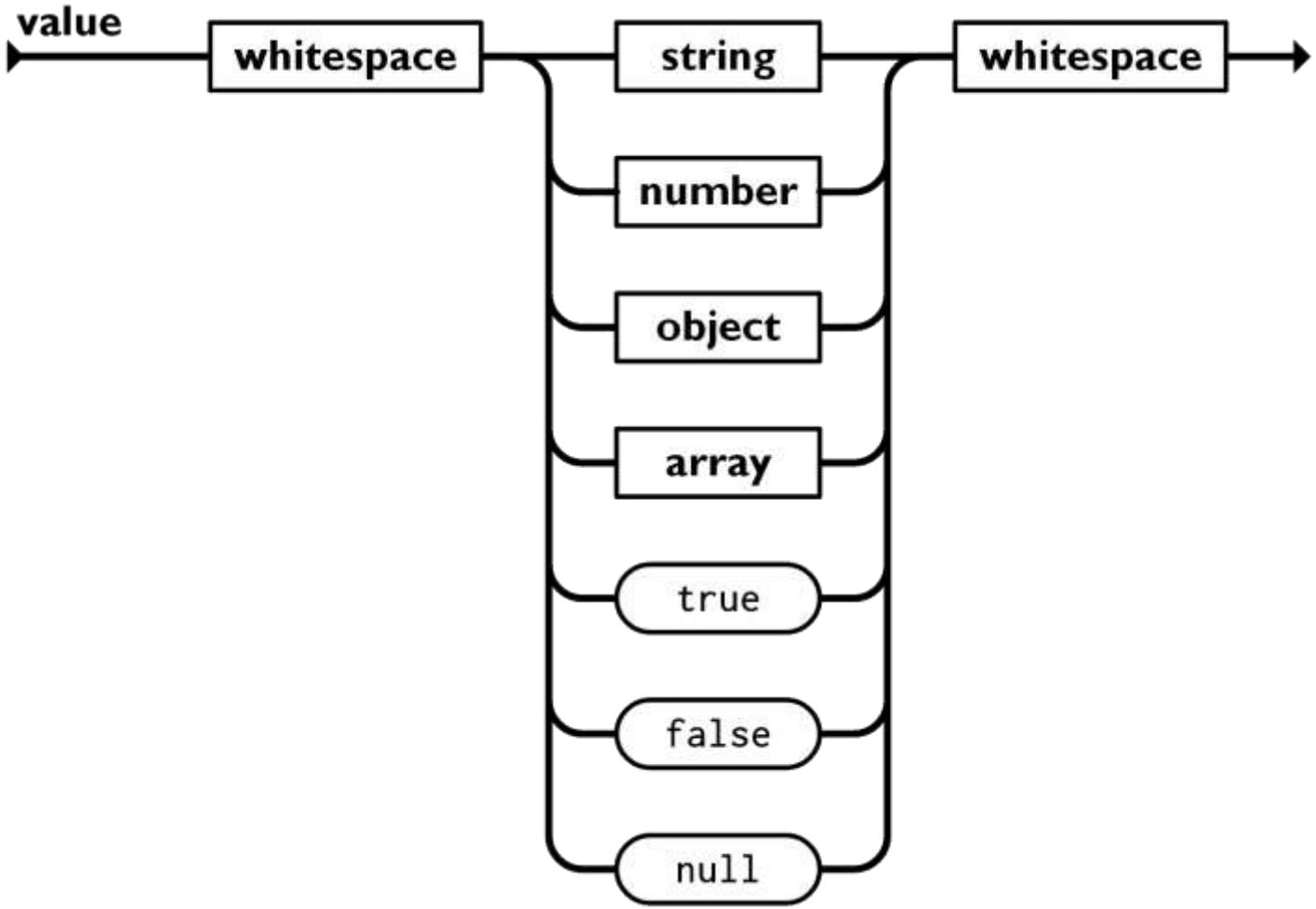
null — зло

02

ADT — алгебраические типы данных (от человека с ADHD)

А потом пришёл тимлид

Начнём с проблемы



Нужно написать анализатор JSON

```
Analyzed analyzeJson(JsonNode node) {
    // TODO
}
```

Путь 1. POJO

```
public final class JsonNode {
    // Boolean | Number | String | JsonNode[] | Map<String, JsonNode>
    private final @Nullable Object value;

    ...
    private final JsonNode(String value) {
        this.value = value;
    }
    ...
}
```

Путь 1. POJO

```
public boolean isNull() {
    return value == null;
}

...
public boolean isString() {
    return value instanceof String;
}

public Optional<String> asString() {
    return value instanceof String string
        ? Optional.of(string)
        : Optional.empty();
}
...
```

Путь 1. POJO

Минусы?

01 **Тайпкасты по Object**

https://youtu.be/QS8q3NKKfHw

Как этим пользоваться?

```
Analyzed analyzeJson(JsonNode node) {
    if (node.isNull()) {
        return onNull();
    }
    {
        var number = node.asNumber();
        if (number.isPresent()) {
            return onNumber(number.get());
        }
    }
    ...
    {
        var object = node.asObject();
        if (number.isPresent()) {
            return onObject(number.get());
        }
    }
    throw new AssertionError("Impossible node type");
}
```

Больно

Путь 1. POJO

Минусы?

01 **Тайпкасты по Object**

02 **Неудобно строить логику через кучу if’ов**

Путь 1. POJO

Минусы?

01 **Тайпкасты по Object**

02 **Неудобно строить логику через кучу if’ов**

03 **Возможное невозможное состояние**

Путь 2. POJO с тегом

```
public final class JsonNode {
    public enum Type { NULL, BOOLEAN, NUMBER, STRING, ARRAY, OBJECT };

    private final @Nullable Object value;
    private final Type type;

    ...
    public JsonNode(String value) {
        this.value = value;
        this.type = Type.STRING;
    }
    ...
}
```

Путь 2. POJO с тегом

```
public Type type() {
    return type;
}

...
public String asString() {
    Preconditions.checkArgument(type == Type.STRING,
        "This is not a `String` JsonNode"
    );
    return (String) value;
}
...
```

Как этим пользоваться?

```
Analyzed analyzeJson(JsonNode node) {
    return switch (node.type()) {
        case NULL -> onNull();
        ...
        case STRING -> onString(node.asString());
        ...
    };
}
```

Проще!

Минусы?

```
Analyzed analyzeJson(JsonNode node) {
    return switch (node.type()) {
        case NULL -> onNull();
        ...
        case STRING -> onNumber(node.asNumber());
        ...
    };
}
```

Путь 3. Visitor + Наследование

```
public interface JsonNodeVisitor<R> {
    R onNull();

    ...
    R onString(String value);
    ...
}
```

Путь 3. Visitor + Наследование

```
public interface JsonNode {
    <R> void visit(JsonNodeVisitor<R> visitor);

    ...
    record String(String value) {
        @Override
        public <R> void visit(JsonNodeVisitor<R> visitor) {
            return visitor.onString(value);
        }
    }
    ...
}
```

Как этим пользоваться?

```
Analyzed analyzeJson(JsonNode node) {
    var self = this;
    return node.visit(new JsonNodeVisitor<>() {
        @Override
        public Analyzed onNull() {
            return self.onNull();
        }

        ...
        @Override
        public Analyzed onString(String value) {
            return self.onString(value);
        }
        ...
    });
}
```

Ну такое

Путь 3. Visitor + Наследование

Минусы?

01 **Visitor’ы монструозны**

Путь 3. Visitor + Наследование

Минусы?

01 **Visitor’ы монструозны**

02 **Visitor’ы нужно аллоцировать**

Путь 3. Visitor + Наследование

Минусы?

01 **Visitor’ы монструозны**

02 **Visitor’ы нужно аллоцировать**

03 **Из Visitor’ов нельзя управлять control flow**

Путь 3. Visitor + Наследование

Минусы?

01 **Visitor’ы монструозны**

02 **Visitor’ы нужно аллоцировать**

03 **Из Visitor’ов нельзя управлять control flow**

Проблема в том...

Что гетерогенность JsonNode — это часть контракта: есть конечный набор возможных вариантов

Но мы любим пить смузи

И пользуемся свежими Джавами

Путь 4. Правильный

```
public sealed interface JsonNode {
    enum Null implements JsonNode { INSTANCE }
    record Boolean(boolean value) implements JsonNode {}
    record Number(Number value) implements JsonNode {}
    record String(String value) implements JsonNode {}
    record Array(JsonNode[] value) implements JsonNode {}
    record Object(Map<String, JsonNode> value) implements JsonNode {}
}
```

Как этим пользоваться?

```
Analyzed analyzeJson(JsonNode node) {
    return switch (node) {
        case JsonNode.Null typed -> onNull();
        case JsonNode.Boolean typed -> onBoolean(typed.value());
        case JsonNode.Number typed -> onNumber(typed.value());
        case JsonNode.String typed -> onString(typed.value());
        case JsonNode.Array typed -> onArray(typed.value());
        case JsonNode.Object typed -> onObject(typed.value());
    };
}
```

Сильно удобнее!

```
Analyzed analyzeJson(JsonNode node) {
    return switch (node) {
        case JsonNode.Null.INSTANCE -> onNull();
        case JsonNode.Boolean(var value) -> onBoolean(value);
        case JsonNode.Number(var value) -> onNumber(value);
        case JsonNode.String(var value) -> onString(value);
        case JsonNode.Array(var value) -> onArray(value);
        case JsonNode.Object(var value) -> onObject(value);
    };
}
```

Плюсы!

- Тип описывает ровно то, чем он является
- Никакого бойлерплейта
- Компилятор следит за руками

- Прикольное название

Тип-сумма — это тип, состоящий из взаимоисключающих вариантов

Звучит знакомо


```
message JsonNode {
  oneof kind {
    Empty null = 1;
    bool boolean = 2;
    double number = 3;
    string string = 4;
    Array array = 5;
    Object object = 6;
  }

  message Array {
    repeated JsonNode elements = 1;
  }
  message Object {
    map<string, JsonNode> children = 1;
  }
}
```

Nullable — тоже тип-сумма

```
public sealed interface Nullable<T> {
    enum Null implements Nullable { INSTANCE }
    record NonNull<T>(T value) Nullable<T> {}
}
```

И называется это Optional

```
public sealed interface Optional<T> {
    enum Empty implements Optional { INSTANCE }
    record Present<T>(T value) Optional<T> {}
}
```

В школе нас учили

А теперь я хочу div-mod

```
public record DivMod(long div, long mod) {}

@IntrinsicCandidate // hopefully
public static DivMod divMod(long a, long b) {
    var div = a / b;
    var mod = a % b;

    return new DivMod(div, mod);
}
```

А ещё я хочу arg-min

```
public record ArgMin(int index, long value) {}

public static ArgMin argMin(long[] nums) {
    Preconditions.checkArgument(nums.length > 0,
        "nums should not be empty"
    );

    var min = Long.MAX_VALUE;
    var minIndex = -1;
    for (var index = 0; index < nums.length; index++) {
        if (nums[index] < min) {
            min = value;
            minIndex = index;
        }
    }

    return new ArgMin(int index, long value);
}
```

Тип-произведение — это тип, являющийся декартовым произведением исходных типов

Проще говоря — кортеж

```
public record Tuple3<T1, T2, T3(
    T1 value1,
    T2 value2,
    T3 value3
) {}
```

Их тоже можно деконструировать

```
public record ArgMin(int index, long value) {}
public static ArgMin argMin(long[] nums) { ... }

...

var argMin = argMin(nums);
if (argMin instanceof ArgMin(var index, var value)) {
    System.out.printf("Index = %d, value = %d", index, value);
}
```

Деконструкция здорового человека


```scala
case class ArgMin(index: Int, value: Long)
def arg_min : ArgMin = ...

...

val (index, nums) = arg_min(nums)
println(f"Index = $index, value = $value")
```

Деконструкция здорового человека


```
def arg_min : (Int, Long) = ...

...

val (index, nums) = arg_min(nums)
println(f"Index = $index, value = $value")
```

Деконструкция здорового человека


```rust
fn arg_min(nums: &[u64]) -> (usize, long) { ... }

...

let (index, value) = arg_min(nums);
println!("Index = {index}, value = {value}")
```

В Джаве пока солёно и только для record’ов

Кстати! А как устроена обработка ошибок в Go?

```
file, err := os.Open("filename.ext")
if err != nil {
    return (nil, err)
}
```

Через как устроена обработка ошибок в Go?

Кстати! А как устроена обработка ошибок в Go?

Опять кортежик

```
func read(path string) (string, error) {
    file, err := os.Open("filename.ext")
    if err != nil {
        return (nil, err)
    }
    ...
}
```

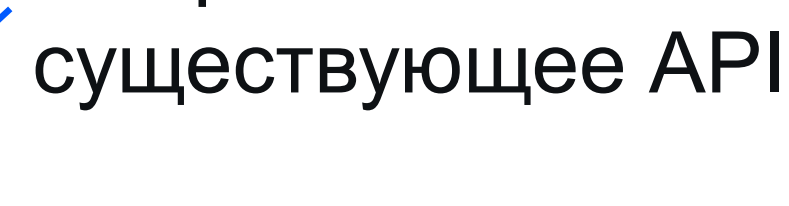
Успех и ошибка — это взаимоисключающие состояния

Что если в Optional сделать пустоту непустой? ...

```
public sealed interface Result<T, E> {
    record Ok<T, E>(T value) implements Result<T, E> {}
    record Err<T, E>(E error) implements Result<T, E> {}
}
```

… И использовать это вместо исключений?

```
public Result<InputStream, IOException> nothrowNewInputStream(
    Path path,
    OpenOption... options
) {
    try {
        return new Result.Ok<>(
            Files.newInputStream(path, options);
        );
    } catch (IOException e)
        return new Result.Error<>(e);
    }
}
```



И будем трактовать успех и ошибку равноправно

```
switch (nothrowNewInputStream(path)) {
    case Result.Ok(var stream) -> {
        log.debug("Opened input stream: {}", stream);
        ...
    };
    case Result.Error(var error) -> {
        log.error("Failed to open input stream", error);
        ...
    };
}
```

Код, получивший Result, обязан обработать оба случая, не делая один из них особенным

Как это делает Rust?


```rust
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Как это делает Rust?


```
match result {
    Ok(stream) => {
        debug!("Opened input stream: {stream}");
        ...
    }
    Err(error) => {
        error!("Failed to open input stream: {error}");
        ...
    }
}
```

Но если мы хотим ранний выход?

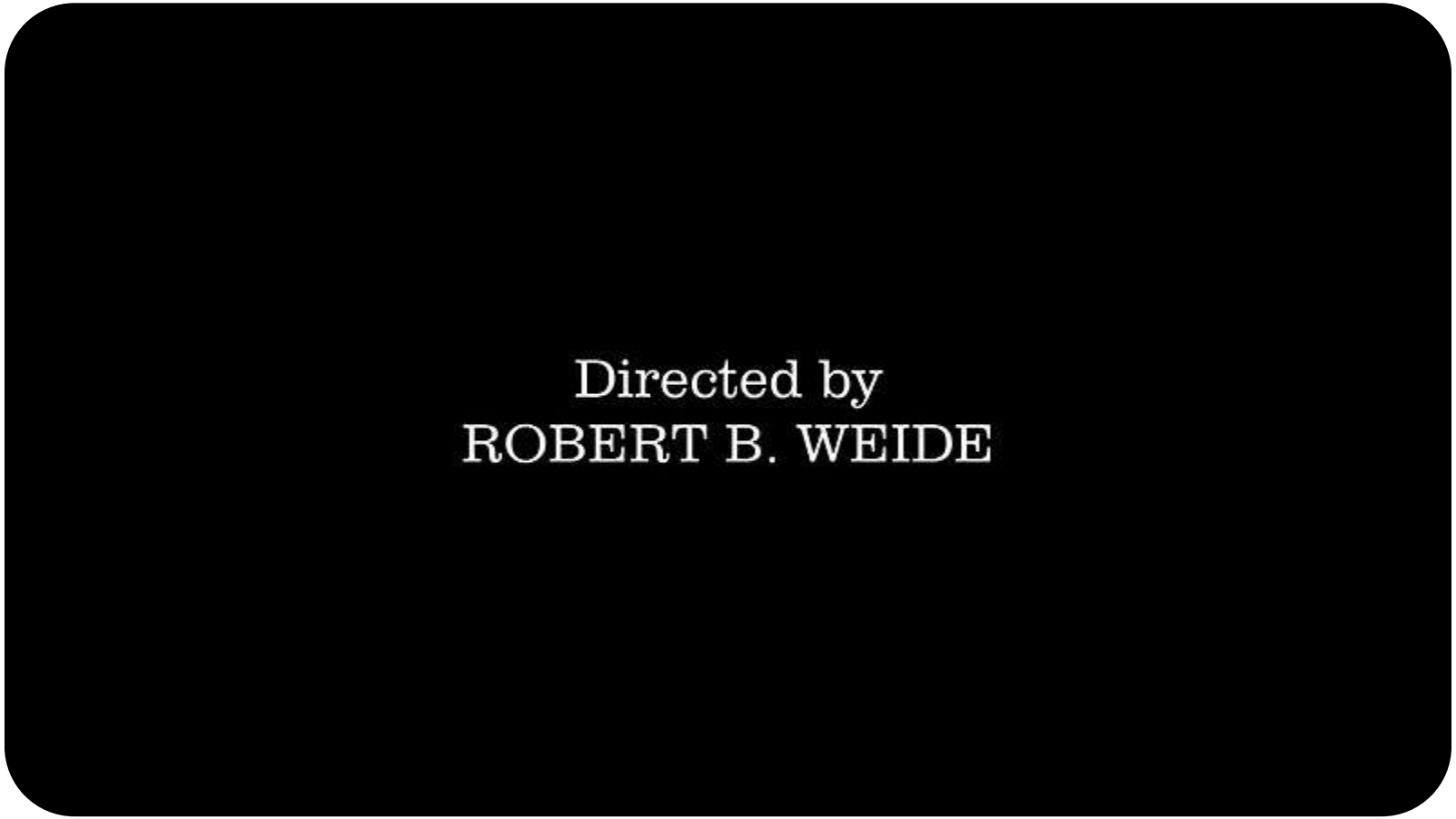
```
var stream = switch (result) {
    case Result.Ok(var it) -> it;
    case Result.Err(var e) -> {
        return new Result.Err(
            MyError.causedBy(e)
        );
    }
}

log.debug("Opened input stream: {}", stream);
```

Но если мы хотим ранний выход?

```
|  Error:
|  attempt to return out of a switch expression
|           return new Result.Err(
|           ^----------------------...
```

Но если мы хотим ранний выход?



(Ну и) как это делает Rust?


```rust
let stream = match result {
    Ok(stream) => stream,
    Err(error) => {
        return Err(
            MyError::caused_by(error)
        );
    }
}
```

(Ну и) как это делает Rust?


```
// При условии, что где-то описано
//  преобразование `From<io::Error> for MyError`
let stream = result?;
```

Нужна дополнительная... Функциональность...

```
public record Result<T, E> {
    <R> Result<R, E> map(
        Function<? super T, ? extends R> mapper
    );

    <R> Result<R, E> flatMap(
        Function<? super T, Result<R, E>> mapper
    );

    T orElse(T fallback);

    T orElseGet(
        Function<? super E, ? extends T> fallback
    );

    ...
}
```

Тривиальные реализации

```
record Ok<T, E>(T value) implements Result<T, E> {
    @Override
    public <R> Result<R, E> map(Function<? super T, ? extends R> mapper) {
        return new Ok<>(mapper.apply(value));
    }

    @Override
    public <R> Result<R, E> flatMap(Function<? super T, Result<R, E>> mapper) {
        return mapper.apply(value);
    }

    @Override
    public T orElse(T fallback) {
        return value;
    }

    @Override
    public T orElseGet(Function<? super E, ? extends T> fallback) {
        return value;
    }

    ...
}
```

Тривиальные реализации

```
record Err<T, E>(E error) implements Result<T, E> {
    @Override
    public <R> Result<R, E> map(Function<? super T, ? extends R> mapper) {
        return new Err<>(error);
    }

    @Override
    public <R> Result<R, E> flatMap(Function<? super T, Result<R, E>> mapper) {
        return new Err<>(error);
    }

    @Override
    public T orElse(T fallback) {
        return fallback;
    }

    @Override
    public T orElseGet(Function<? super E, ? extends T> fallback) {
        return fallback.apply(error);
    }

    ...
}
```

Тривиальные реализации

```
record Err<T, E>(E error) implements Result<T, E> {
    @SuppressWarnings("unchecked")
    public <R> Result<R, E> cast() {
        return (Result<R, E>) this;
    }

    @Override
    public <R> Result<R, E> map(Function<? super T, ? extends R> mapper) {
        return cast();
    }

    @Override
    public <R> Result<R, E> flatMap(Function<? super T, Result<R, E>> mapper) {
        return cast();
    }

    @Override
    public T orElse(T fallback) {
        return fallback;
    }

    @Override
    public T orElseGet(Function<? super E, ? extends T> fallback) {
        return fallback.apply(error);
    }

    ...
}
```

Функционально пользуемся нашим API

```
public Result<Document, BusinessError> readDocument(Path path) {
    return openInputStream(path)
        .map(BufferedInputStream::new)
        .flatMap(this::readAllLines)
        .mapErr(BusinessError.ReadFailed::new)
        .flatMap(lines -> parseDocument(lines)
            .onOk(doc -> log.debug(
                "Parsed doc from path {}: {}", path, doc
            ))
            .mapErr(BusinessError::InvalidDoc)
        );
}
```

Функционально пользуемся нашим API

```
public Result<Document, BusinessError> readDocument(Path path) {
    return openInputStream(path) // Result<InputStream, IoError>
        .map(BufferedInputStream::new) // Result<BufferedInputStream, IoError>
        .flatMap(this::readAllLines) // Result<List<String>, IoError>
        .mapErr(BusinessError.ReadFailed::new) // Result<List<String>, BusinessError>
        .flatMap(lines -> parseDocument(lines) // Result<Document, DocParseError>
            .onOk(doc -> log.debug(
                "Parsed doc from path {}: {}", path, doc
            )) // Result<Document, DocParseError>
            .mapErr(BusinessError::InvalidDoc)
        ); // Result<Document, BusinessError>
}

private Result<List<String>, IoError> readAllLines(BufferedInputStream stream) { ... }
private Result<Document, DocParseError> parseDocument(List<String> stream) { ... }
```

В дикой природе

https://www.javadoc.io/doc/io.vavr/vavr/latest/io/vavr/control/Try.html

https://www.javadoc.io/doc/ru.progrm-jarvis/java-commons/latest/ru/progrm_jarvis/javacommons/object/Result.html

Монада

1. Есть некий монадический тип
    1. **Optional**
    2. **Result**
2. Операция *unit* создаёт её из значения value
    - **Optional.of(value)**
    - **new Result.Ok(value)**
3. Есть операция *bind*, принимающее преобразование *f* из значения внутри монады в новую монаду
    1. **Optional.flatMap(f)**
    2. **Result.flatMap(f)**


php
C
java


Java
php

Это (чудесное) видео

https://youtu.be/NM3TU5VfEMM

Промежуточные идеи

«Один из» — тип-сумма

Кортеж — тип-произведение

С ними красиво работает сопоставление с образцом

Альтернатива исключениям — *Result*, но для его обработки нужна либо поддержка языка, либо функциональщина

Монада — это просто моноид в категории эндофункторов

А ещё *Result* и *Optional* — это монады

03 Применение продвинутых типов на практике

Protobuf — это ужасный формат

Но остальные ещё хуже

Protobuf-запрос

```
// Подзапрос, отдельно учитывающий распространённые (стоп-)слова.
// В отличии от CommonTermsQueryNode, данная версия не производит анализ текста,
// так как принимает текст в виде дерева токенов TextTreeNode.
message TokenizedCommonTermsQueryNode {
  // Поле, по которому осуществляется поиск.
  //
  // Обязательный параметр.
  string field = 1;

  // Текст в виде дерева токенов, по которому осуществляется поиск.
  //
  // Обязательный параметр.
  TextTreeNode text_tree = 2;

  // Частота обрезания распространённых слов.
  //
  // Принимает значение в интервале [0;1].
  //
  // Misc: https://ru.pinterest.com/pin/418131146660603959/
  float cutoff_frequency = 3;
}
```

Protobuf-запрос для непосвящённых

```
message SubQueryFoo {
  // Поле, по которому осуществляется поиск.
  //
  // Обязательный параметр.
  string field = 1;

  // Текст в виде дерева токенов, по которому осуществляется поиск.
  //
  // Обязательный параметр.
  TextTreeNode text_tree = 2;

  // Частота обрезания распространённых слов.
  //
  // Принимает значение в интервале [0;1].
  float freq = 3;
}
```

Как парсят Джависты?

```
1 public static FooQuery parse(FooQueryNode queryNode) {
2     var field = queryNode.getField();
3     var textTree = queryNode.getTextTree();
4     var freq = queryNode.getFreq();
5
6     return new FooQuery(field, textTree, freq);
7 }
```

Как парсят Джависты?

```
1 public static FooQuery parse(FooQueryNode queryNode) {
2     var field = queryNode.getField();
3     Preconditions.checkArgument(!field.isEmpty(),
4         "`field` should not be empty"
5     );
6     Preconditions.checkArgument(queryNode.hasTextTree(),
7         "`text_tree` should be set"
8     );
9     var textTree = queryNode.getTextTree();
10    var freq = queryNode.getFreq();
11    Preconditions.checkArgument(0 <= freq && freq <= 1,
12        "`freq` should be in range [0;1]"
13    );
14
15    return new FooQuery(field, textTree, freq);
16 }
```

Ну серьёзно!

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a `String` containing the `int` representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Проблемы?

Как парсят долгоживущие Джависты?

```
1 private final class ParseException extends Exception { ... }
2
3 public static void check(
4     boolean condition,
5     String message,
6     Objects... formatArgs
7 ) throws ParseException {
8     if (!condition) {
9         throw new ParseException(
10            message.formatted(formatArgs)
11        );
12    }
13 }
```

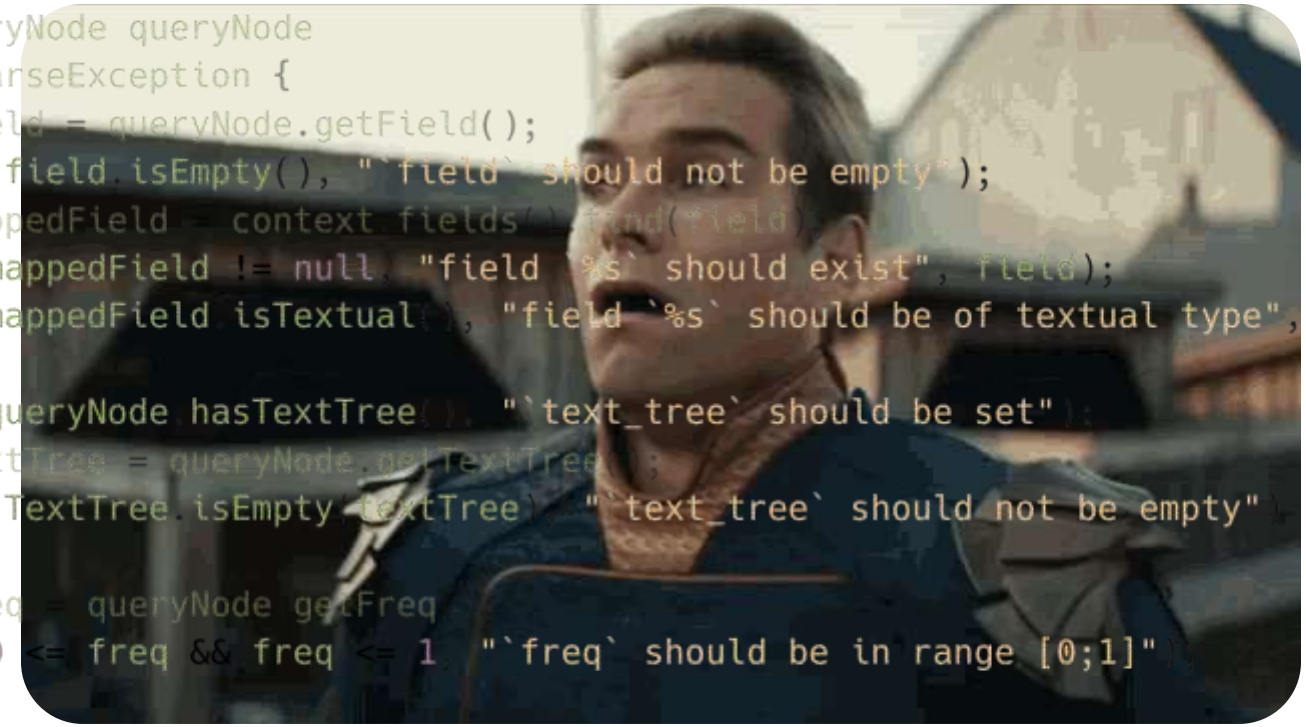
Как парсят долгоживущие Джависты?

```
1 public static FooQuery parse(FooQueryNode queryNode)
2     throws ParseException
3  {
4     var field = queryNode.getField();
5     check(!field.isEmpty(), "`field` should not be empty");
6     check(queryNode.hasTextTree(), "`text_tree` should be set");
7     var textTree = queryNode.getTextTree();
8     var freq = queryNode.getFreq();
9     check(0 <= freq && freq <= 1, "`freq` should be in range [0;1]");
10
11    return new FooQuery(field, textTree, freq);
12 }
```

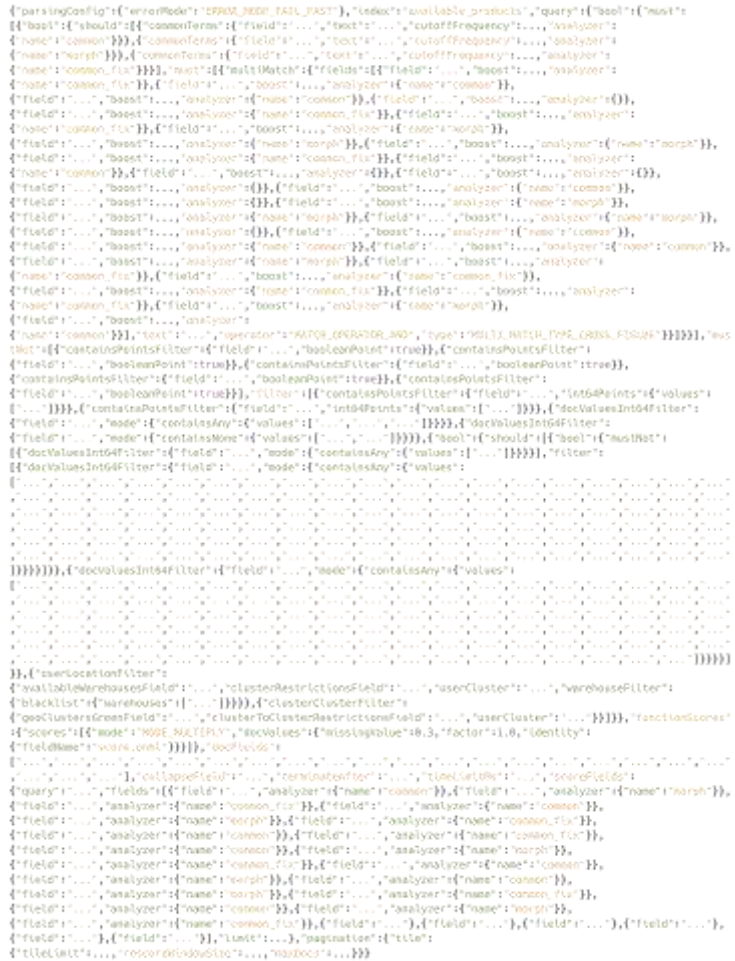
Как парсят долгоживущие Джависты?

Как парсят ответственные Джависты?

```
1 public static FooQuery parse(
2     ParsingContext context,
3     FooQueryNode queryNode
4 ) throws ParseException {
5     var field = queryNode.getField();
6     check(!field.isEmpty(), "`field` should not be empty");
7     var mappedField = context.fields().find(field);
8     check(mappedField != null, "field `%s` should exist", field);
9     check(mappedField.isTextual(), "field `%s` should be of textual type", mappedField);
10
11    check(queryNode.hasTextTree(), "`text_tree` should be set");
12    var textTree = queryNode.getTextTree();
13    check(!TextTree.isEmpty(textTree), "`text_tree` should not be empty");
14
15    var freq = queryNode.getFreq();
16    check(0 <= freq && freq <= 1, "`freq` should be in range [0;1]");
17
18    return new FooQuery(field, textTree, freq);
19 }
```



А теперь отправим маленький запросик



И получим ошибку

```
`field` should not be empty
```

Мы почти SQL

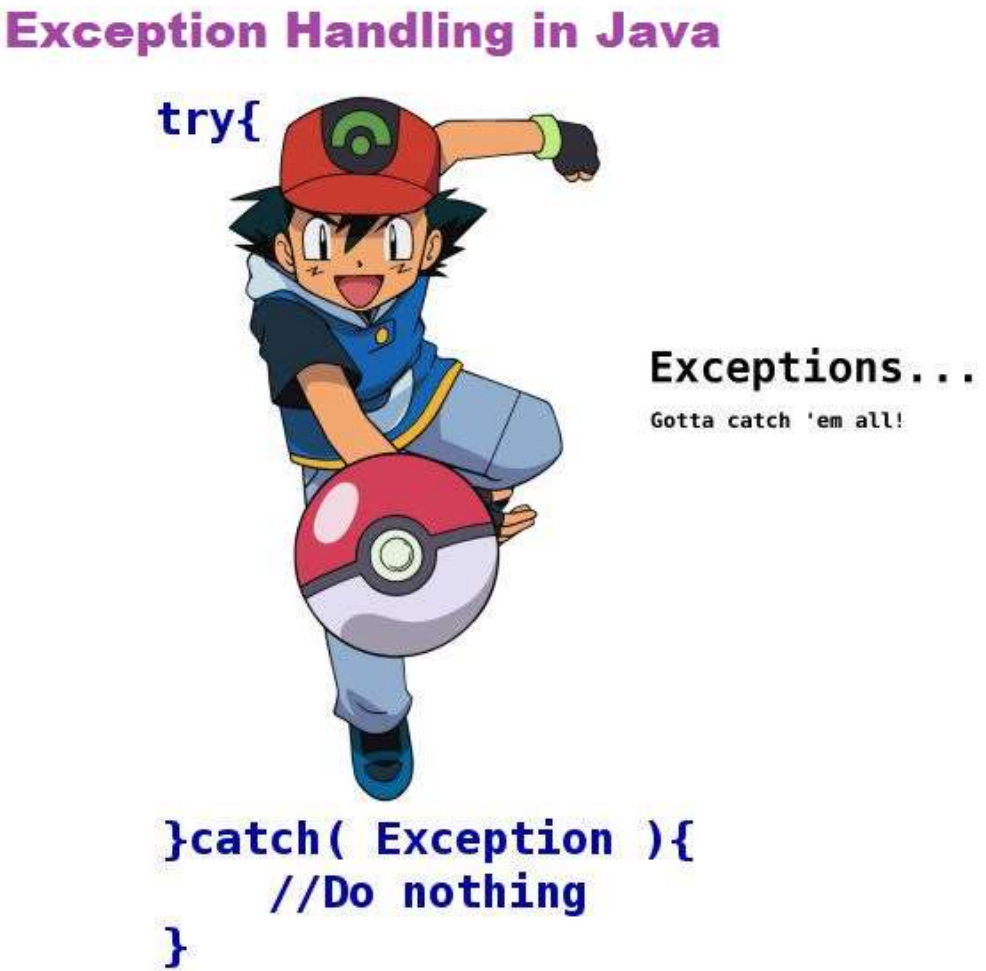

ERROR: SQL syntax error at or near ";"
Cool, an error, this might help.
Actually, Wow this is absolutely useless.

Мы хотим сохранять
путь до проблемных нод
Protobuf-запроса

Но у нас же есть эксепшены

```
1 public static FooQuery parse(
2     ParsingContext context,
3     FooQueryNode queryNode
4 ) throws ParseException {
5     try {
6         // Всё тело парсера.
7         ...
8         // Много строчек.
9     } catch (ParseException e) {
10        // Наращиваем скоуп
11        throw new ParseException("FooQuery", e)
12    }
13 }
```

Ах исключения, исключения

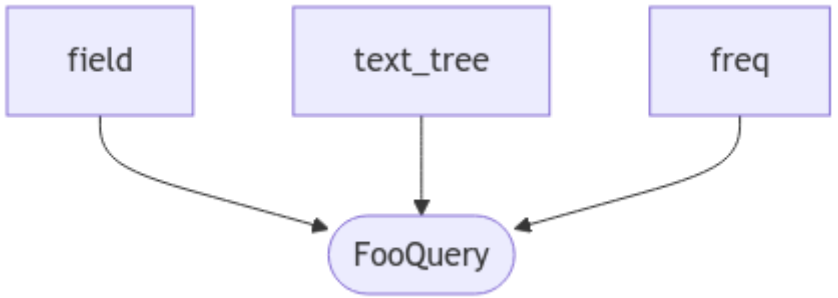


Можно слегка декомпозировать

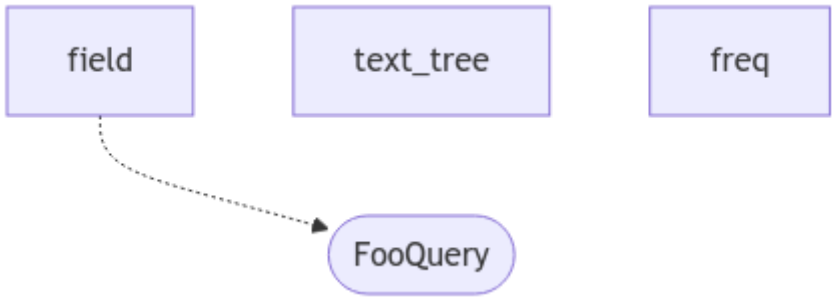
```
1 public static FooQuery parse(
2     ParsingContext context,
3     FooQueryNode queryNode
4 ) throws ParseException {
5     try {
6         return parseInScope(context, queryNode);
7     } catch (ParseException e) {
8         // Наращиваем скоуп
9         throw new ParseException("FooQuery", e)
10    }
11 }
12
13 private static FooQuery parseInScope(
14    ParsingContext context,
15    FooQueryNode queryNode
16 ) throws ParseException { ... }
```

Если мы выбросили исключение,
то другие ошибки мы уже не найдём

Представим всё как граф

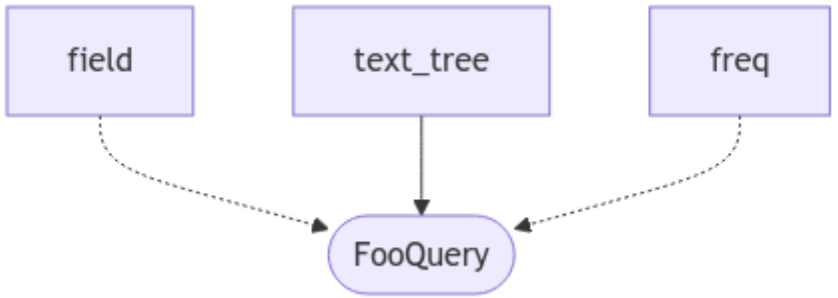


Одна ошибка — и остальное даже не проверили

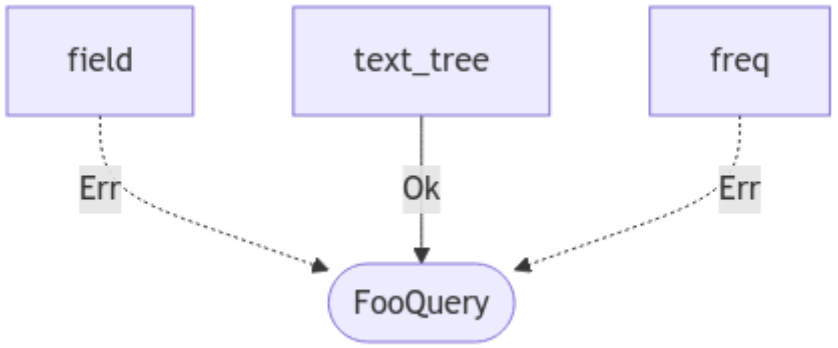


Давайте решать проблемы по мере поступления

Давайте решать проблемы по мере поступления



Давайте решать проблемы по мере поступления



Давайте просто формировать Result’ы и лениво их пробрасывать

Кастомный Result

```java
public sealed interface ParseResult<T> {
    record Ok<T>(
        T value
    ) implements ParseResult<T> {}

    record Error<T>(
        String message,
        List<Error<?>> errors
    ) implements ParseResult<T> {}
}
```

Кастомный Result

```
1 public sealed interface ParseResult<T> {
2     record Ok<T>(
3         T value
4     ) implements ParseResult<T> {}
5
6     record Error<T>(
7         String message,
8         List<Error<?>> errors
9     ) implements ParseResult<T> {}
10 }
```

Кастомный Result

```
1  public sealed interface ParseResult<T> {
2      record Ok<T>(
3          T value
4      ) implements ParseResult<T> {}
5
6      record Error<T>(
7          String message,
8          List<Error<?>> errors
9      ) implements ParseResult<T> {}
10 }
```

Ну либо так ☞☜


```rust
type ParseResult<T> = Result<T, ParseError>;
```

Атомарный парсер

```java
public static ParseResult<MappedField> parseField(
    ParsingContext context,
    String field,
    Set<FieldType> permittedTypes
) {
    if (field.isEmpty()) {
        return new Err("field is empty");
    }

    var mappedField = context.fields().find(field);
    if (mappedField == null) {
        return new Err("field `%s` is empty"
            .formatted(field)
        );
    }

    var fieldType = mappedField.type();
    if (!permittedTypes.contains(fieldType)) {
        return new Err("field `%s` is of type %s but should be of type %s"
            .formatted(field)
        );
    }

    return new Ok(mappedField);
}
```

Внутри fail-fast проверки

```
6  if (field.isEmpty()) {
7      return new Err("field is empty");
8  }
9
10 var mappedField = context.fields().find(field);
11 if (mappedField == null) {
12     return new Err("field `%s` is empty"
13         .formatted(field)
14     );
15 }
16
17 var fieldType = mappedField.type();
18 if (!permittedTypes.contains(fieldType)) {
19     return new Err("field `%s` is of type %s but should be of type %s"
20         .formatted(field)
21     );
22 }
23
24 return new Ok(mappedField);
```

Вышестоящий парсер

```
1 public static ParseResult<FooQuery> parse(
2     ParsingContext context,
3     FooQueryNode queryNode
4 ) {
5     var parsedField = CommonParsers
6         .parseField(context, queryNode.getField(), FieldTypes.TEXTUAL)
7         .inScope("field");
8     var parsedTextTree = queryNode.hasTextTree()
9         ? TextTreeParser
10            .parse(context, queryNode.getTextTree())
11            .inScope("text_tree")
12        : new Err("`text_tree` is unset");
13    var parsedFreq = CommonParsers.require0to1(context, queryNode.getFreq())
14        .inScope("freq");
15
16    return ???;
17 }
```

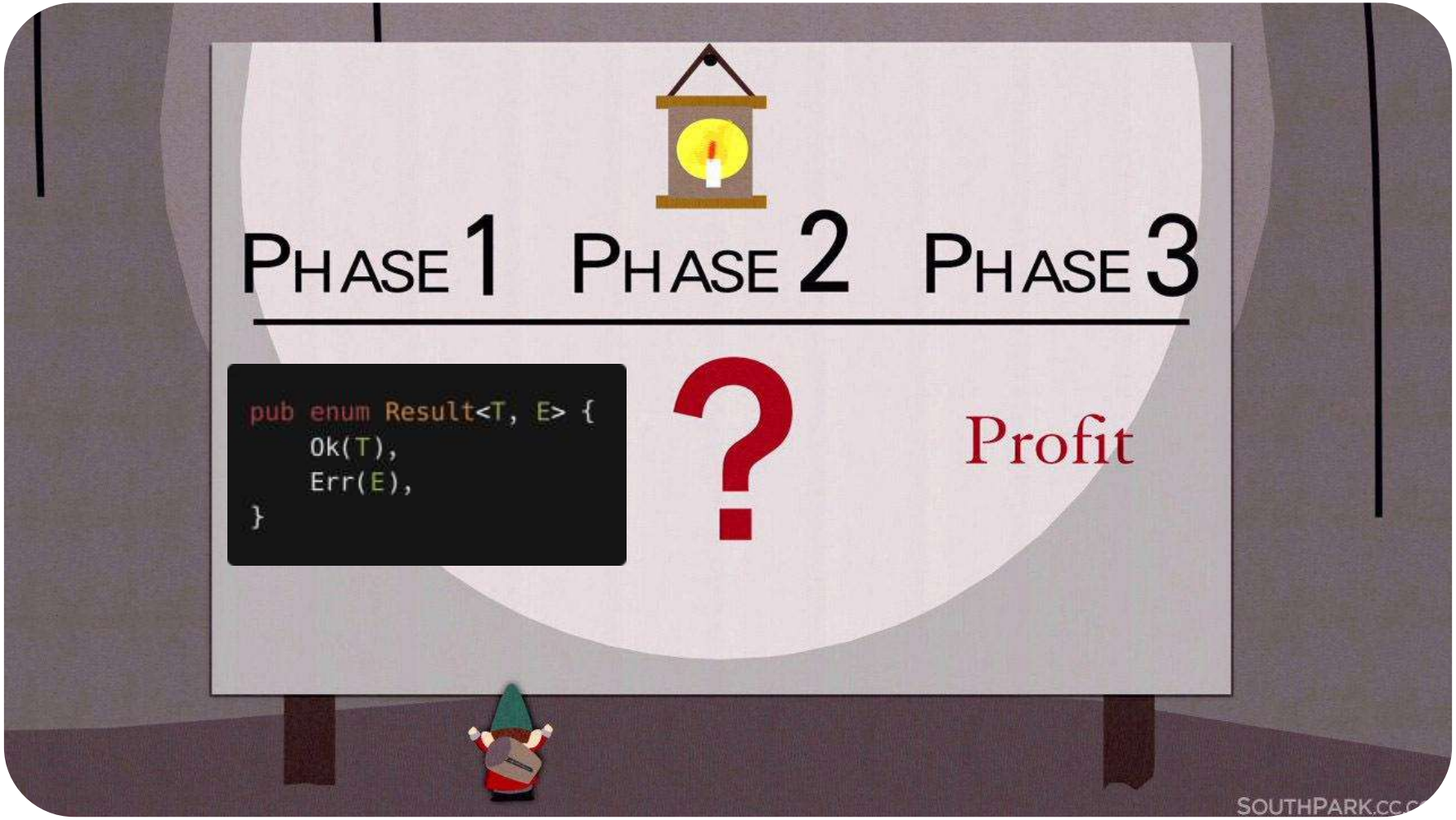
Вышестоящий парсер

```
1 public static ParseResult<FooQuery> parse(
2     ParsingContext context,
3     FooQueryNode queryNode
4 ) {
5     var parsedField = CommonParsers
6         .parseField(context, queryNode.getField(), FieldTypes.TEXTUAL)
7         .inScope("field");  // ParseResult<MappedField>
8     var parsedTextTree = queryNode.hasTextTree()
9         ? TextTreeParser
10            .parse(context, queryNode.getTextTree())
11            .inScope("text_tree")
12        : new Err("`text_tree` is unset");  // ParseResult<TextTree>
13    var parsedFreq = CommonParsers.require0to1(context, queryNode.getFreq())
14        .inScope("freq"); // ParseResult<Float>
15
16    return ???;
17 }
```

???

```
1  public static ParseResult<FooQuery> parse(
2      ParsingContext context,
3      FooQueryNode queryNode
4  ) {
5      var parsedField = CommonParsers
6          .parseField(context, queryNode.getField(), FieldTypes.TEXTUAL)
7          .inScope("field");  // ParseResult<MappedField>
8      var parsedTextTree = queryNode.hasTextTree()
9          ? TextTreeParser
10             .parse(context, queryNode.getTextTree())
11             .inScope("text_tree")
12         : new Err("`text_tree` is unset");  // ParseResult<TextTree>
13     var parsedFreq = CommonParsers.require0to1(context, queryNode.getFreq())
14         .inScope("freq"); // ParseResult<Float>
15
16     return ???;
17 }
```

PROFIT!


PHASE 1
PHASE 2
PHASE 3
pub enum Result<T, E> {
Ok(T),
Err(E),
}
?
Profit
SOUTHPARK.CC.C

Как собрать результаты воедино?

В духе старого доброго

```java
static <T1, T2, T3, R> ParseResult<R> allOk(
    String scope,
    ParseResult<T1> result1,
    ParseResult<T2> result2,
    ParseResult<T3> result3,
    Function3<? super T1, ? super T2, ? super T3, ? extends R> mapper
) {
    if (result1 instanceof Ok(value1)
        && result2 instanceof Ok(value2)
        && result3 instanceof Ok(value3)) {
        return new Ok<>(mapper.apply(value1, value2, value3));
    }

    var errors = new ArrayList<Error<?>>();
    result1.addErrorTo(errors);
    result2.addErrorTo(errors);
    result3.addErrorTo(errors);

    return new Error<>(scope, Collections.unmodifiableList(errors));
}
```

Монадического

```java
static <T1, T2, T3, R> ParseResult<R> allOk(
    String scope,
    ParseResult<T1> result1,
    ParseResult<T2> result2,
    ParseResult<T3> result3,
    Function3<? super T1, ? super T2, ? super T3, ? extends R> mapper
) {
    if (result1 instanceof Ok(value1)
        && result2 instanceof Ok(value2)
        && result3 instanceof Ok(value3)) {
        return new Ok<>(mapper.apply(value1, value2, value3));
    }

    var errors = new ArrayList<Error<?>>();
    result1.addErrorTo(errors);
    result2.addErrorTo(errors);
    result3.addErrorTo(errors);

    return new Error<>(scope, Collections.unmodifiableList(errors));
}
```

Если всё хорошо, то всё хорошо

```java
static <T1, T2, T3, R> ParseResult<R> allOk(
    String scope,
    ParseResult<T1> result1,
    ParseResult<T2> result2,
    ParseResult<T3> result3,
    Function3<? super T1, ? super T2, ? super T3, ? extends R> mapper
) {
    if (result1 instanceof Ok(value1)
        && result2 instanceof Ok(value2)
        && result3 instanceof Ok(value3)) {
        return new Ok<>(mapper.apply(value1, value2, value3));
    }

    var errors = new ArrayList<Error<?>>();
    result1.addErrorTo(errors);
    result2.addErrorTo(errors);
    result3.addErrorTo(errors);

    return new Error<>(scope, Collections.unmodifiableList(errors));
}
```

Если не всё хорошо, то всё плохо

```java
static <T1, T2, T3, R> ParseResult<R> allOk(
    String scope,
    ParseResult<T1> result1,
    ParseResult<T2> result2,
    ParseResult<T3> result3,
    Function3<? super T1, ? super T2, ? super T3, ? extends R> mapper
) {
    if (result1 instanceof Ok(value1)
        && result2 instanceof Ok(value2)
        && result3 instanceof Ok(value3)) {
        return new Ok<>(mapper.apply(value1, value2, value3));
    }

    var errors = new ArrayList<Error<?>>();
    result1.addErrorTo(errors);
    result2.addErrorTo(errors);
    result3.addErrorTo(errors);

    return new Error<>(scope, Collections.unmodifiableList(errors));
}
```

Но есть нюанс

```
allOk(String, ParseResult<T1>, ThrowingFunction1<? super T1, ? extends R, X>): ParseResul
tryAllOk(String, ParseResult<T1>, ThrowingFunction1<? super T1, ? extends ParseResult<R>
allOk(String, ParseResult<T1>, ParseResult<T2>, ThrowingFunction2<? super T1, ? super T2
allOk(String, List<ParseResult<T>>, ThrowingFunction1<List<T>, ? extends R, X>): ParseRe
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ThrowingFunction2<? super T1, ? supe
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ThrowingFunction3<? s
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ThrowingFunction3<
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Throw
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Th
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Parse
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Pa
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Parse
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Pa
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Parse
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Parse
allOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Parse
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Pa
tryAllOk(String, ParseResult<T1>, ParseResult<T2>, ParseResult<T3>, ParseResult<T4>, Pa
```

Ну миленькое же!

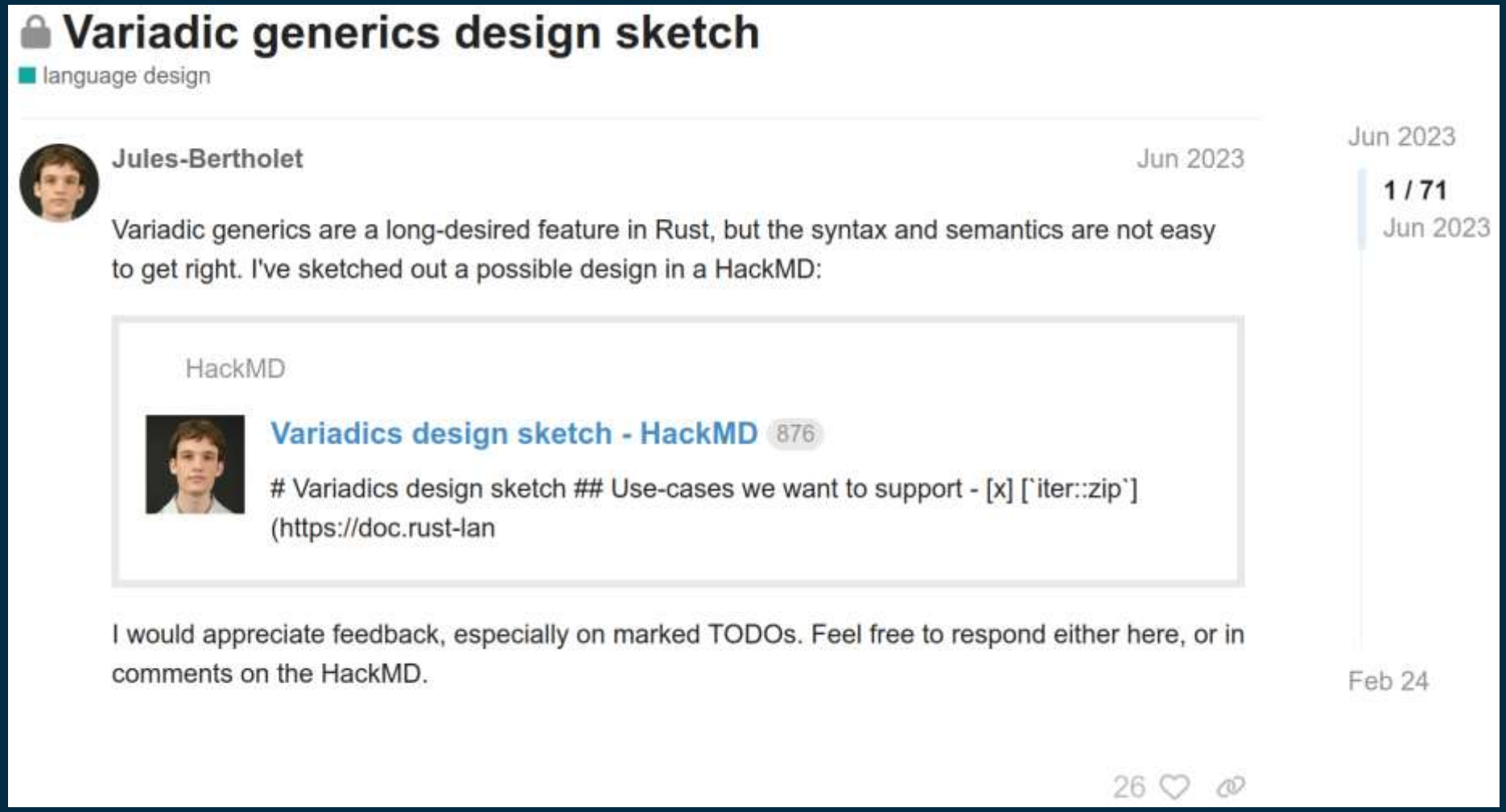
```
public static ParseResult<FooQuery> parse(
    ParsingContext context,
    FooQueryNode queryNode
) {
    var parsedField = CommonParsers
        .parseField(context, queryNode.getField(), FieldTypes.TEXTUAL)
        .inScope("field");
    var parsedTextTree = queryNode.hasTextTree()
        ? TextTreeParser
            .parse(context, queryNode.getTextTree())
            .inScope("text_tree")
        : new Err("`text_tree` is unset");
    var parsedFreq = CommonParsers.require0to1(context, queryNode.getFreq())
        .inScope("freq");

    return ParseResult.allOk(
        "FooQuery",
        parsedField, parsedTextTree, parsedFreq,
        FooQuery::new
    );
}
```

Ну миленькое же!

```
return ParseResult.allOk(
    "FooQuery",
    parsedField, parsedTextTree, parsedFreq,
    FooQuery::new
);
```

Справедливости ради



Variadic generics design sketch

language design

Jun 2023

Jules-Bertholet Jun 2023

Variadic generics are a long-desired feature in Rust, but the syntax and semantics are not easy to get right. I've sketched out a possible design in a HackMD:

HackMD

Variadics design sketch - HackMD 876

Variadics design sketch ## Use-cases we want to support - [x] [`iter::zip`] (https://doc.rust-lan

I would appreciate feedback, especially on marked TODOs. Feel free to respond either here, or in comments on the HackMD.

26

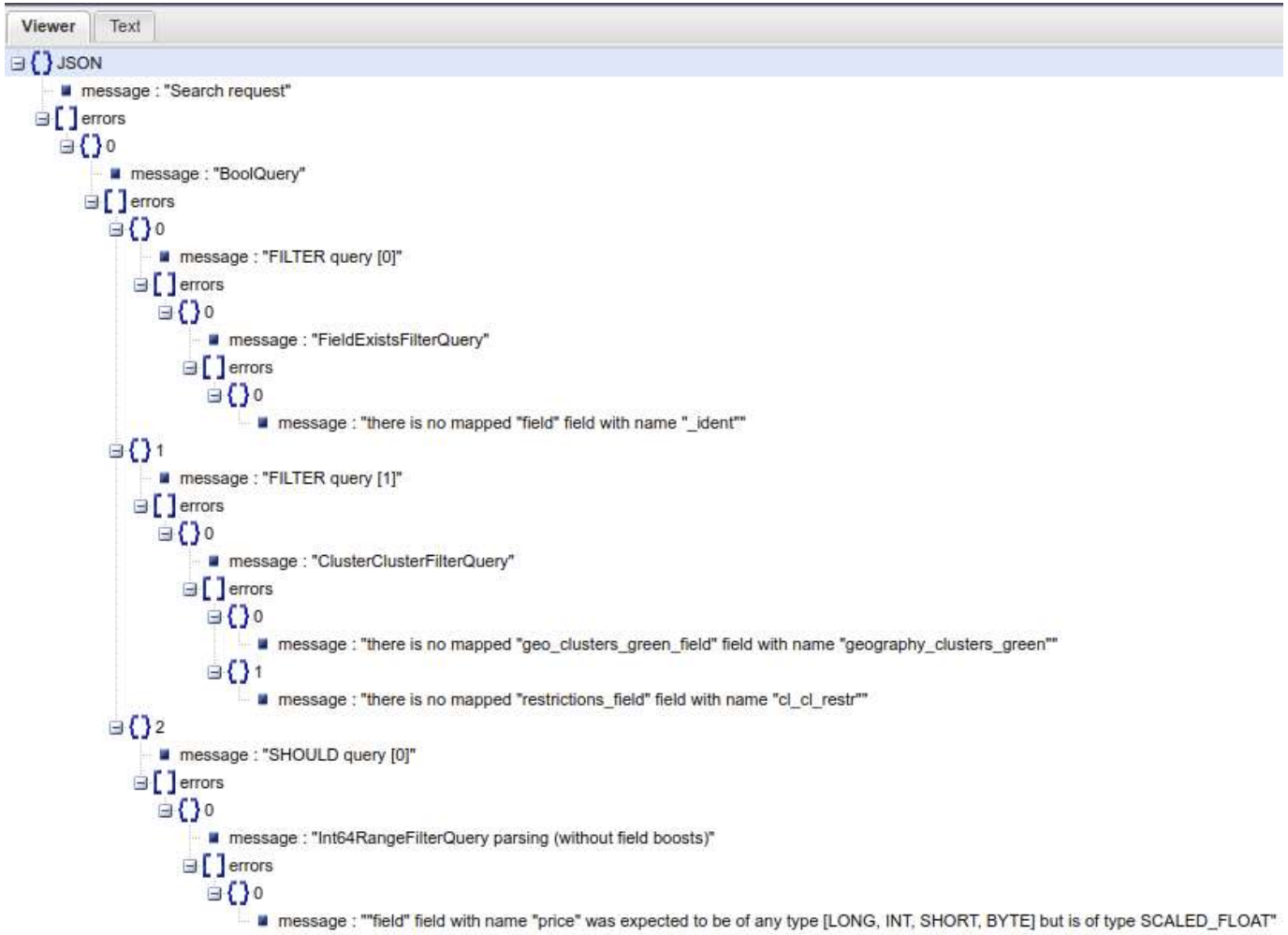
Jun 2023

1 / 71

Jun 2023

Feb 24

И получили мы красивое



И получили мы красивое

Промежуточные идеи

ADT помогают не только в моделировании данных, но и в вычислениях над ними

Result можно использовать для ленивых вычислений с контекстом

Variadic generics — это сложно, но нужно

04 {

Экзотические типы

А какой тип у exit(0)?

```
var what = System.exit(0);
```

Или у бесконечного цикла

```
var what = while (true) {
    ...
};
```

Или у ошибки в всегда успешном Result

```
public interface IntParser<T, E> {
    Result<Integer, E> parse(T value);
}

public class StringToIntParser implements IntParser<String, NumberFormatException> {
    @Override
    public Result<Integer, NumberFormatException> parse(String value) {
        try { return new Ok(Integer.parseInt(value)); }
        catch (NumberFormatException e) { return new Err(e); }
    }
}

public class IntToIntParser implements IntParser<Integer, <???>> {
    @Override
    public Result<Integer, <???>> parse(Integer value) {
        return new Ok(value);
    }
}
```

Или у ошибки в всегда успешном Result

```
public interface IntParser<T, E> {
    Result<Integer, E> parse(T value);
}

public class StringToIntParser implements IntParser<String, NumberFormatException> {
    @Override
    public Result<Integer, NumberFormatException> parse(String value) {
        try { return new Ok(Integer.parseInt(value)); }
        catch (NumberFormatException e) { return new Err(e); }
    }
}

public class IntToIntParser implements IntParser<Integer, <???>> {
    @Override
    public Result<Integer, <???>> parse(Integer value) {
        return new Ok(value);
    }
}
```

Никогда не говори никогда

```
public sealed interface Never {}
```

Или говори :(

```
|  Error:
|  sealed class must have subclasses
|  public sealed interface Never {}
|  ^-----------------------------^
```

Как там у людей?

Сюда не смотрите

Primitive Type never [-]

This is a nightly-only experimental API. (never_type #35121)

[-] The `!` type, also called "never".

`!` represents the type of computations which never resolve to any value at all. For example, the `exit` function `fn exit(code: i32) -> !` exits the process without ever returning, and so returns `!`.

`break`, `continue` and `return` expressions also have type `!`. For example we are allowed to write:

```
#![feature(never_type)]
let x: ! = {
    return 123
};
```

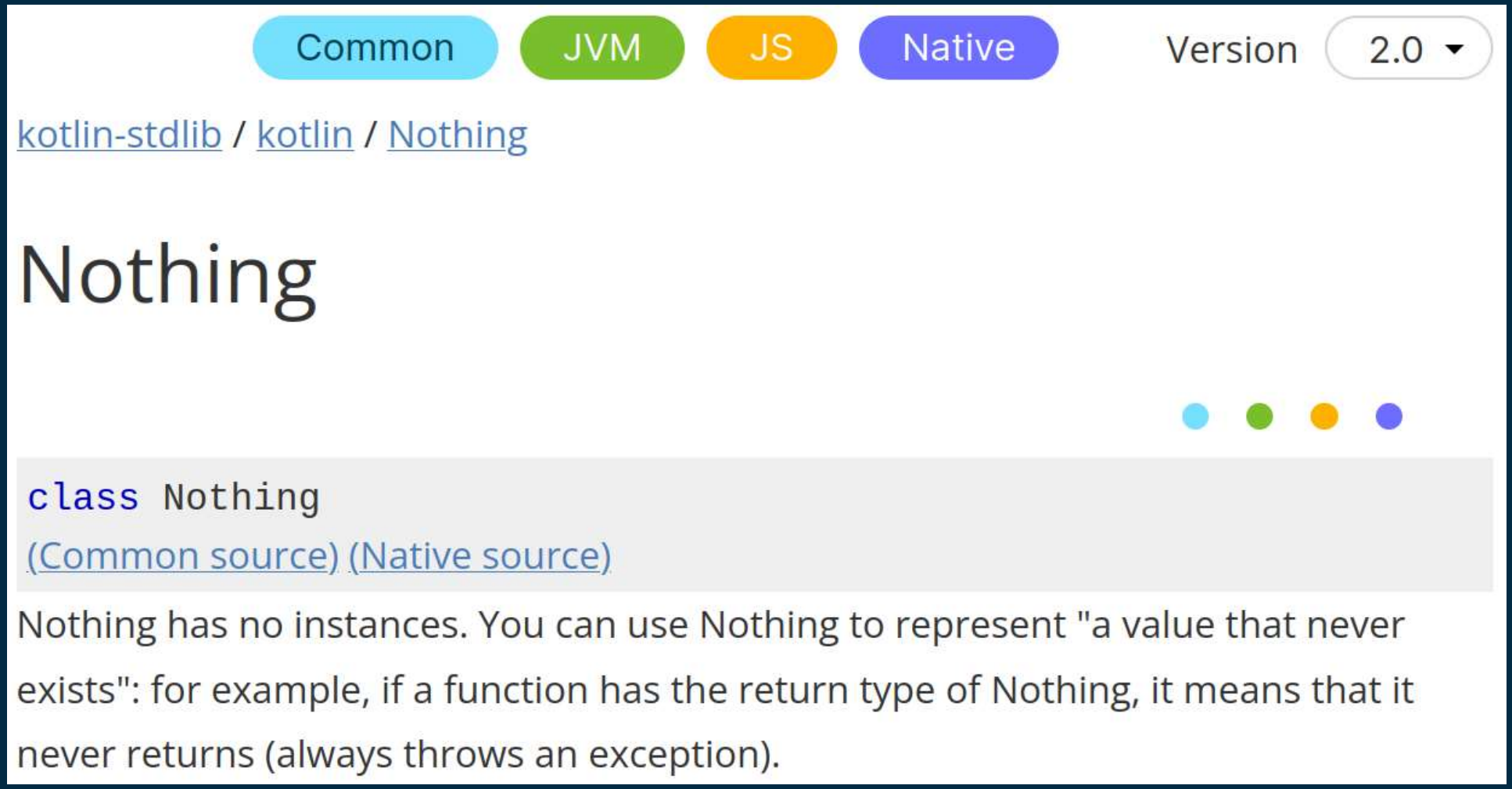
Although the `let` is pointless here, it illustrates the meaning of `!`. Since `x` is never assigned a value (because `return` returns from the entire function), `x` can be given type `!`. We could also replace `return 123` with a `panic!` or a never-ending `loop` and this code would still be valid.

A more realistic usage of `!` is in this code:

```
let num: u32 = match get_a_number() {
    Some(num) => num,
    None => break,
};
```

Both match arms must produce values of type `u32`, but since `break` never produces a value at all we know it can never produce a value which isn't a `u32`. This illustrates another behaviour of the `!` type - expressions with type `!` will coerce into any other type.

Как там у людей?



А зачем?

```
public static Never unreachable() {
    throw new AssertionError("This should be unreachable");
}
```

А зачем?

```
enum Coin { HEAD, TAIL, SIDE }

public static Coin flipCoin() {
    return switch (ThreadLocalRandom.current().nextInt(3)) {
        case 0 -> Coin.HEAD;
        case 1 -> Coin.TAIL;
        case 2 -> Coin.SIDE;
    };
}
```

А зачем?

```
|  Error:
|  the switch expression does not cover all possible input values
|      return switch (ThreadLocalRandom.current().nextInt(3)) {
|             ^-----------------------------------------...
```

Затем!

```
enum Coin { HEAD, TAIL, SIDE }

public static Coin flipCoin() {
    return switch (ThreadLocalRandom.current().nextInt(3)) {
        case 0 -> Coin.HEAD;
        case 1 -> Coin.TAIL;
        case 2 -> Coin.SIDE;
        default -> unreachable();
    };
}
```

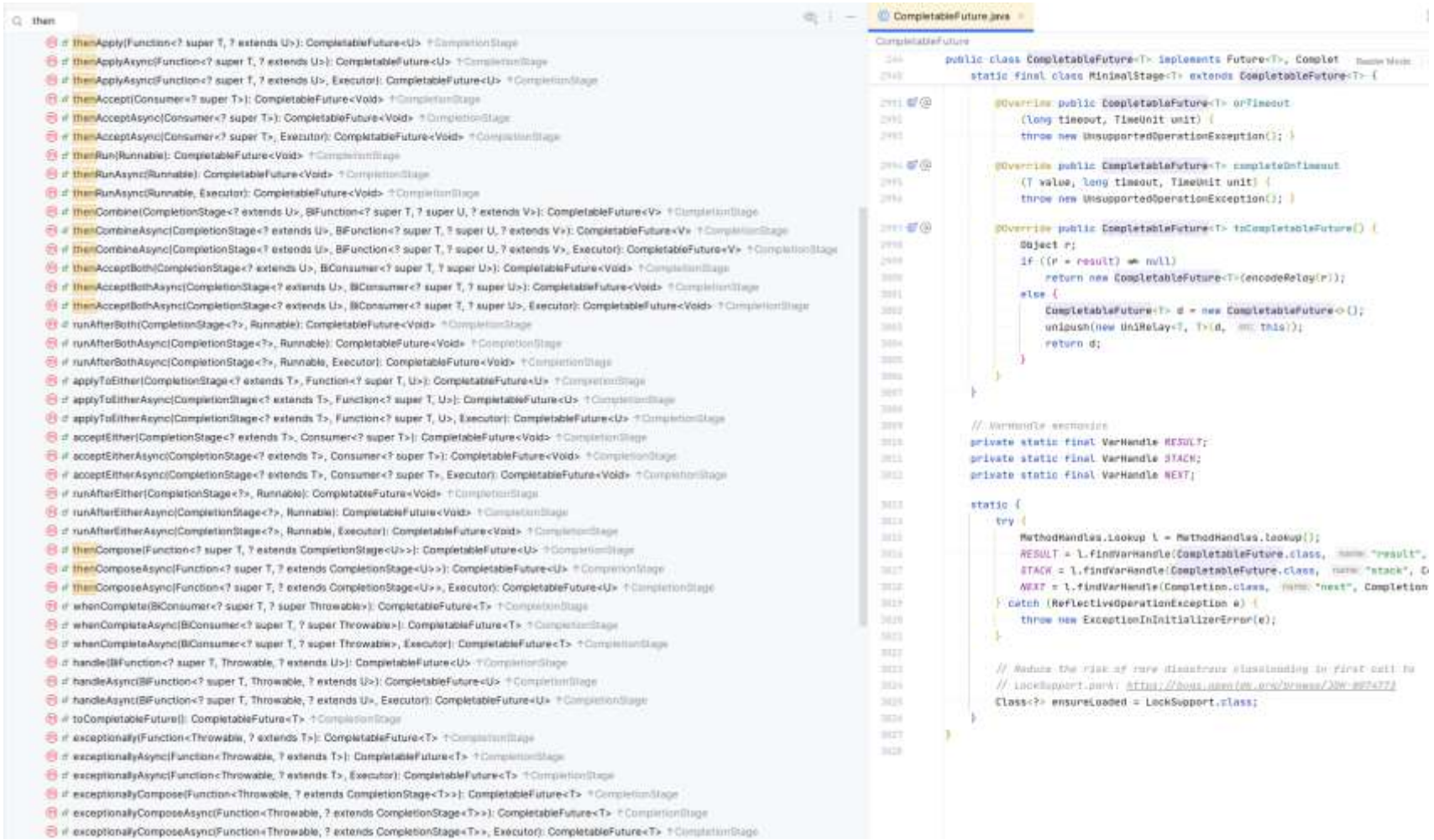
Могло быть красиво

```
|  Error:
|  incompatible types: bad type in switch expression
|      Never cannot be converted to Coin
|          default -> unreachable();
|                     ^----------^
```

Ну и для полноты картины

```
public enum Unit { INSTANCE }
```

Зачем?


then
thenApply(Function<? super T, ? extends U>): CompletableFuture<U>
thenApplyAsync(Function<? super T, ? extends U>): CompletableFuture<U>
thenApplyAsync(Function<? super T, ? extends U>, Executor): CompletableFuture<U>
thenAccept(Consumer<? super T>): CompletableFuture<Void>
thenAcceptAsync(Consumer<? super T>): CompletableFuture<Void>
thenAcceptAsync(Consumer<? super T>, Executor): CompletableFuture<Void>
thenRun(Runnable): CompletableFuture<Void>
thenRunAsync(Runnable): CompletableFuture<Void>
thenRunAsync(Runnable, Executor): CompletableFuture<Void>
thenCombine(CompletionStage<? extends U>, BiFunction<? super T, ? super U, ? extends V>): CompletableFuture<V>
thenCombineAsync(CompletionStage<? extends U>, BiFunction<? super T, ? super U, ? extends V>): CompletableFuture<V>
thenCombineAsync(CompletionStage<? extends U>, BiFunction<? super T, ? super U, ? extends V>, Executor): CompletableFuture<V>
thenAcceptBoth(CompletionStage<? extends U>, BiConsumer<? super T, ? super U>): CompletableFuture<Void>
thenAcceptBothAsync(CompletionStage<? extends U>, BiConsumer<? super T, ? super U>): CompletableFuture<Void>
thenAcceptBothAsync(CompletionStage<? extends U>, BiConsumer<? super T, ? super U>, Executor): CompletableFuture<Void>
runAfterBoth(CompletionStage<?>, Runnable): CompletableFuture<Void>
runAfterBothAsync(CompletionStage<?>, Runnable): CompletableFuture<Void>
runAfterBothAsync(CompletionStage<?>, Runnable, Executor): CompletableFuture<Void>
applyToEither(CompletionStage<? extends T>, Function<? super T, U>): CompletableFuture<U>
applyToEitherAsync(CompletionStage<? extends T>, Function<? super T, U>): CompletableFuture<U>
applyToEitherAsync(CompletionStage<? extends T>, Function<? super T, U>, Executor): CompletableFuture<U>
acceptEither(CompletionStage<? extends T>, Consumer<? super T>): CompletableFuture<Void>
acceptEitherAsync(CompletionStage<? extends T>, Consumer<? super T>): CompletableFuture<Void>
acceptEitherAsync(CompletionStage<? extends T>, Consumer<? super T>, Executor): CompletableFuture<Void>
runAfterEither(CompletionStage<?>, Runnable): CompletableFuture<Void>
runAfterEitherAsync(CompletionStage<?>, Runnable): CompletableFuture<Void>
runAfterEitherAsync(CompletionStage<?>, Runnable, Executor): CompletableFuture<Void>
thenCompose(Function<? super T, ? extends CompletionStage<U>>): CompletableFuture<U>
thenComposeAsync(Function<? super T, ? extends CompletionStage<U>>): CompletableFuture<U>
thenComposeAsync(Function<? super T, ? extends CompletionStage<U>>, Executor): CompletableFuture<U>
whenComplete(BiConsumer<? super T, ? super Throwable>): CompletableFuture<T>
whenCompleteAsync(BiConsumer<? super T, ? super Throwable>): CompletableFuture<T>
whenCompleteAsync(BiConsumer<? super T, ? super Throwable>, Executor): CompletableFuture<T>
handle(BiFunction<? super T, Throwable, ? extends U>): CompletableFuture<U>
handleAsync(BiFunction<? super T, Throwable, ? extends U>): CompletableFuture<U>
handleAsync(BiFunction<? super T, Throwable, ? extends U>, Executor): CompletableFuture<U>
toCompletableFuture(): CompletableFuture<T>
exceptionally(Function<Throwable, ? extends T>): CompletableFuture<T>
exceptionallyAsync(Function<Throwable, ? extends T>): CompletableFuture<T>
exceptionallyAsync(Function<Throwable, ? extends T>, Executor): CompletableFuture<T>
exceptionallyCompose(Function<Throwable, ? extends CompletionStage<T>>): CompletableFuture<T>
exceptionallyComposeAsync(Function<Throwable, ? extends CompletionStage<T>>): CompletableFuture<T>
exceptionallyComposeAsync(Function<Throwable, ? extends CompletionStage<T>>, Executor): CompletableFuture<T>
CompletableFuture.java
CompletableFuture
public class CompletableFuture<T> implements Future<T>, Complet
static final class MinimalStage<T> extends CompletableFuture<T> {
@Override public CompletableFuture<T> orTimeout
(long timeout, TimeUnit unit) {
throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> completeOnTimeout
(T value, long timeout, TimeUnit unit) {
throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> toCompletableFuture() {
Object r;
if ((r = result) == null)
return new CompletableFuture<T>(encodeRelay(r));
else {
CompletableFuture<T> d = new CompletableFuture<>();
unipush(new UniRelay<T, T>(d, this));
return d;
}
}
}
// VarHandle mechanics
private static final VarHandle RESULT;
private static final VarHandle STACK;
private static final VarHandle NEXT;
static {
try {
MethodHandles.Lookup l = MethodHandles.lookup();
RESULT = l.findVarHandle(CompletableFuture.class, "result",
STACK = l.findVarHandle(CompletableFuture.class, "stack",
NEXT = l.findVarHandle(Completion.class, "next", Completion
} catch (ReflectiveOperationException e) {
throw new ExceptionInInitializerError(e);
}
// Reduce the risk of rare disastrous classloading in first call to
// LockSupport.park: https://bugs.openjdk.org/browse/JDK-8074773
Class<?> ensureLoaded = LockSupport.class;
}
}

Есть два пути

```
public <T, R> R foo(Function<? super T, ? extends R> function) { ... }

...

var baz = foo(input -> return new Baz(bar(input)));
foo(input -> {
    qux(input);
    return null; // Как бы `Void`, но не `Void`.
});
```

Есть два пути

```
public <T, R> R foo(Function<? super T, ? extends R> function) { ... }
public <T, R> void foo(Consumer<? super T> consumer) {
    foo(input -> {
        consumer.accept(input);
        return null; // Пофиг, абсолютно.
    });
}

...

var baz = foo(input -> return new Baz(bar(input)));
foo(input -> qux(input));
```

Промежуточные идеи

Экзотические типы интересны и дают возможность описывать некоторые интересные вырожденные случаи

Перечисленные экзотические типы являются частными случаями ADT

Компилятор всё ещё вставляет палки в колёса там, где это не нужно

05

Заключение

Итого

01 Описывай домен типами

02 Парси, а не валидируй

03 Помогай компилятору типами

04 Тип-сумма — наше всё

05 Кортежи полезны с сахаром

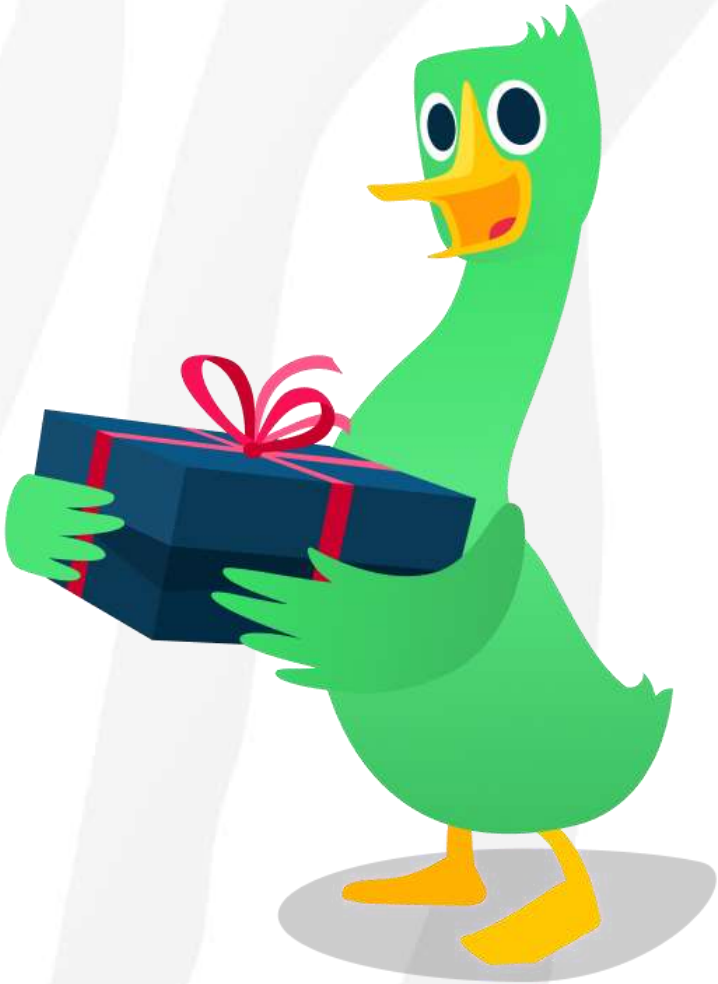
06 Пустота тоже полезна

07 Иногда говори «Никогда»

08 `<T...>` сложны

Истинное зло

- `null`
- исключения
- неправильная магия



Темы для дискуссионки

Цена вопроса: сколько это стоит во время исполнения

ОВЁС (с) Шипилёв: а оно нам точно так надо?

Альтернативы: какие ещё можно было бы решить перечисленные проблемы?

А также мерч за лучшие вопросы

OZON tech

Спасибо за внимание!

Пётр Портнов, старший разработчик

pportnov@ozon.ru

github.com/JarvisCraft