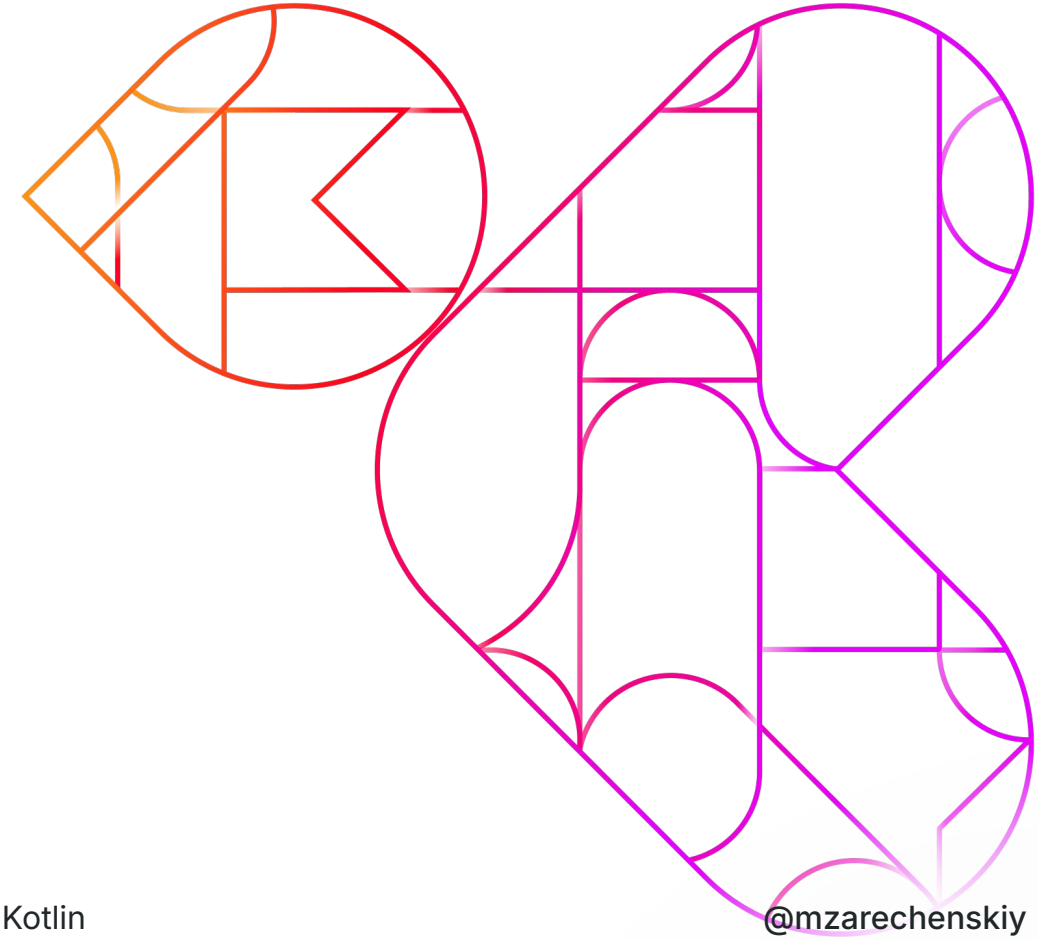
KotlinConf 2025



Rich Errors

Michail Zarečenskij, Lead Language Designer @ Kotlin

@mzarechenskiy

What are Rich Errors?

```
fun fetchUser(): User | FetchingError
fun User.charge(amount: Double): TransactionId | TransactionError

when (val transaction = fetchUser()?.charge(amount = 10.0)) {
    is TransactionId → println("Success, transaction id: $transaction")
    is FetchingError → println("Could load user data")
    is TransactionError →
        println("Transaction error: ${transaction.errorMessage}")
}
```

What we'll cover today

- Quick story about exceptions in Java
- We'll discuss errors using a toy example
- Rich Errors syntax and semantics
- Short-cuts and chain calls

Error model

- How can I, as a developer, express that a method might fail?
- If a method might fail, how can I handle errors?

Exceptions

Exceptions: the origin. C minimalism

```
int err = read_file("missing.txt");
if (err) {
    printf("Failed to read file: %s\n", strerror(err));
    return 1;
}
```

Exceptions: the origin. C limitations

```c
int err = read_file("missing.txt");
if (err) {
    printf("Failed to read file: %s\n", strerror(err));
    return 1;
}
```

No type safety, simple error codes

Easy to forget to check errors

No ergonomics for nested function calls

Checked exceptions in Java

```
void readFile(String name) throws IOException { ... }
```

Checked exceptions in Java

```
void readFile(String name) throws IOException { ... }
```

```
try {
    readFile("file.txt");
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Checked exceptions in Java

```
void readFile(String name) throws IOException { ... }
```

```
try {
    readFile("file.txt");
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Boilerplate? Should be unchecked one? Often, we can’t do anything meaningful with the exception

Checked exceptions in Java

```
void readFile(String name) throws IOException { ... }


void usage() throws IOException, … {
   readFile("file.txt");
}
```

Checked exceptions in Java: lambdas

```
void readFile(String name) throws IOException { ... }


void usage(List<String> files) {
   files.forEach(file → readFile(file));
}
```

Checked exceptions don’t compose with higher-order functions

Checked exceptions in Java: lambdas

```
void readFile(String name) throws IOException { ... }


void usage(List<String> files) {
    files.forEach(file → {
        try {
            readFile(file);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    });
}
```

Checked exceptions in Java

- Force developers to handle errors explicitly 👍
- Make code more robust by preventing silent failures 👍

However, in practice:

- Don’t compose with higher-order functions
- Checked exceptions are part of regular exceptions:
  - Uses non-trivial control-flow
  - Works fine for sequential, fail-fast behavior, but not in the async world
  - As part of the unchecked exception, checked ones were often highly misused

Kotlin and unchecked exceptions

- Kotlin inherited the concept of exceptions but only *unchecked* once
- Unchecked exceptions have become a distinct feature
  - Mostly, for unrecoverable cases and fail-first behavior
  - If I want to signal about a recoverable case, how should I do it?
- The lack of language support for errors has led to various organic approaches to error handling…

Toy example

Toy example: nulls as errors

```
// null indicates of a Network error
fun fetchUser(): Json? { … }

// null indicates of a parsing error
fun Json.parseUser(): User? { … }
```

Toy example: nulls as errors

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

fun getUser(): User? = fetchUser()?.parseUser()

fun usage() {
    val user = getUser()
    println(user.name) // error
    if (user == null) {
        println("Failed to get user")
        return
    }

    println(user.name)
}
```

Toy example

Ergonomics for chain-calls 👍

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

fun getUser(): User? = fetchUser()?.parseUser()

fun usage() {
    val user = getUser()
    println(user.name) // error
    if (user == null) {
        println("Failed to get user")
        return
    }

    println(user.name)
}
```

Type-safety 👍

Smartcasts 👍

Toy example: nulls as errors

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

fun getUser(): User? = fetchUser()?.parseUser()

fun usage() {
    val user = getUser()
    println(user.name) // error
    if (user == null) {
        println("Failed to get user")
        return
    }

    println(user.name)
}
```

Unclear error:
Parsing or network error? 🤔

Toy example: sealed hierarchies

```
// null indicates of a Network error
fun fetchUser(): Json? { … }

// null indicates of a parsing error
fun Json.parseUser(): User? { … }
```

Toy example: sealed hierarchies

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

sealed interface UserResult {
    data class Success(val user: User) : UserResult
    data object NetworkError : UserResult
    data object ParsingError : UserResult
}
```

Toy example: sealed hierarchies

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

sealed interface UserResult { … }

fun getUserSealed(): UserResult {
    val user = fetchUser() ?: return NetworkError
    val parsedUser = user.parseUser() ?: return ParsingError
    return Success(parsedUser)
}

fun usageSealed() {
    when (val result = getUserSealed()) {
        is Success → {
            ...
            println(result.user.name)
        }
        NetworkError → println("Failed to fetch user")
        ParsingError → println("Failed to parse user")
    }
}
```

Toy example: sealed hierarchies

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

sealed interface UserResult { … }

fun getUserSealed(): UserResult {
    val user = fetchUser() ?: return NetworkError
    val parsedUser = user.parseUser() ?: return ParsingError
    return Success(parsedUser)
}

fun usageSealed() {
    when (val result = getUserSealed()) {
        is Success → {
            ...
            println(result.user.name)
        }
        NetworkError → println("Failed to fetch user")
        ParsingError → println("Failed to parse user")
    }
}
```

Clear errors 👍

Toy example: sealed hierarchies

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

sealed interface UserResult { … }

fun getUserSealed(): UserResult {
    val user = fetchUser() ?: return NetworkError
    val parsedUser = user.parseUser() ?: return ParsingError
    return Success(parsedUser)
}

fun usageSealed() {
    when (val result = getUserSealed()) {
        is Success → {
            ...
            println(result.user.name)
        }
        NetworkError → println("Failed to fetch user")
        ParsingError → println("Failed to parse user")
    }
}
```

Extensibility 

Verbosity 👎

Nesting 50 50

Toy example: more extensibility

```
sealed interface FetchingResult {
    data class Success(val user: Json) : FetchingResult
🆕  data object NotFound : FetchingResult
    data object NetworkError : FetchingResult
}

sealed interface ParsingResult {
    data class Success(val user: User) : ParsingResult
    data object ParsingError : ParsingResult
}

fun fetchUserSealed(): FetchingResult { ... }
fun Json.parseUserSealed(): ParsingResult { ... }
```

Toy example: more extensibility

```
sealed interface FetchingResult {
    data class Success(val user: Json) : FetchingResult
    data object NotFound : FetchingResult
    data object NetworkError : FetchingResult
}

sealed interface ParsingResult {
    data class Success(val user: U
    data object ParsingError : Pars
}

fun fetchUserSealed(): FetchingResu
fun Json.parseUserSealed(): Parsing
```

Generic Result-like class?

```
sealed interface Result<out T> {
    data class Success<T>(val value: T) : Result<T>
    data class Failure(val error: Throwable) : Result<Nothing>
}
```

Toy example: Result

```
class NetworkException : Exception()
class ParsingException : Exception()

fun fetchUserResult(): Result<Json> {
    ...
    return Result.failure(NetworkException())
    ...
}

fun Json.parseUserResult(): Result<User> {
    ...
    return Result.failure(ParsingException())
}
```

Toy example: Result

```
fun getUserResult(): Result<User> {
    val user = fetchUserResult().getOrElse { return Result.failure(it) }
    val parsedUser = user.parseUserResult().getOrElse { return Result.failure(it) }
    return Result.success(parsedUser)
}

fun usageResult() {
    getUserResult()
        .onSuccess { user → println(user.name) }
        .onFailure { error →
            when (error) {
                is NetworkException → println("Failed to fetch user")
                is ParsingException → println("Failed to parse user")
                else → println("Unhandled error")
            }
        }
}
```

Toy example: Result


```
fun getUserResult(): Result<User> {
    val user = fetchUserResult().getOrElse { return Result.failure(it) }
    val parsedUser = user.parseUserResult().getOrElse { return Result.failure(it) }
    return Result.success(parsedUser)
}

fun usageResult() {
    getUserResult()
        .onSuccess { user → println(user.name) }
        .onFailure { error →
            when (error) {
                is NetworkException → println("Failed to fetch user")
                is ParsingException → println("Failed to parse user")
                else → println("Unhandled error")
            }
        }
}
```

Toy example: Result

Verbosity 👎

Functional operators 50/50

Non-exhaustive when 👎

```
fun getUserResult(): Result<User> {
    val user = fetchUserResult().getOrElse { return Result.failure(it) }
    val parsedUser = user.parseUserResult().getOrElse { return Result.failure(it) }
    return Result.success(parsedUser)
}

fun usageResult() {
    getUserResult()
        .onSuccess { user → println(user.name) }
        .onFailure { error →
            when (error) {
                is NetworkException → println("Failed to fetch user")
                is ParsingException → println("Failed to parse user")
                else → println("Unhandled error")
            }
        }
}
```

Original example

```
fun fetchUser(): Json? { … }
fun Json.parseUser(): User? { … }

fun getUser(): User? = fetchUser()?.parseUser()

fun usage() {
    val user = getUser()
    println(user.name) // error
    if (user == null) {
        println("Failed to get user")
        return
    }

    println(user.name)
}
```

Suppose you have one rabbit.

Now suppose someone gives you one more rabbit.

Now, if you count your rabbits, you have two rabbits. So one rabbit plus one rabbit equals two rabbits. So one plus one equals two.

1 + 1 = 2

And that is how arithmetic is done.

Now that you understand the basic idea behind arithmetic, let's take a look at a simple easy-to-understand example that puts into practice what we just learned.

Try It Out
Example 1.7

$$\log\Pi(N) = \left(N + \frac{1}{2}\right)\log N - N + A - \int_N^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{x}, \quad A = 1 + \int_1^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{x}$$

$$\log\Pi(s) = \left(s + \frac{1}{2}\right)\log s - s + A - \int_0^\infty \frac{\overline{B}_1(t)\mathrm{d}t}{t+s}$$

$$\log\Pi(s) = \lim_{n\to\infty}\left[s\log(N+1) + \sum_{n=1}^{N}\log n - \sum_{n=1}^{N}\log(s+n)\right]$$

$$= \lim \left[s\log(N+1) + \int_1^N \log x\mathrm{d}x - \frac{1}{2}\log N + \int_1^N \frac{\overline{B}_1(x)\mathrm{d}x}{x}\right.$$

If the authors of computer programming books wrote arithmetic textbooks...

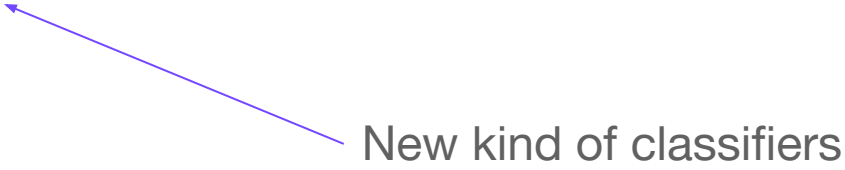

WORK IN
PROGRESS

Rich errors

```
error class NetworkError
error class ParsingError
```

Rich errors

```
error class NetworkError
error class ParsingError
```



Rich errors

```
error class NetworkError
error class ParsingError

fun fetchUserResult(): Json | NetworkError
fun Json.parseUserResult(): User | ParsingError

fun getUser(): User | NetworkError | ParsingError
```

Rich errors

```
error class NetworkError
error class ParsingError

fun fetchUserResult(): Json | NetworkError
fun Json.parseUserResult(): User | ParsingError

fun getUser(): User | NetworkError | ParsingError
```

New syntax for declaring union types with errors

Rich errors

```
error class NetworkError
error class ParsingError

fun fetchUserResult(): Json | NetworkError
fun Json.parseUserResult(): User | ParsingError

typealias UserError = NetworkError | ParsingError

fun getUser(): User | UserError
```

Rich errors

```
fun fetchUser(): Json | NetworkError { … }
fun Json.parseUser(): User | ParsingError { … }
fun getUser(): User | UserError = fetchUser()?.parseUser()

fun usage() {
    val user = getUser()
    println(user.name) // error
    if (user !is User) {
        println("Failed to get user")
        return
    }

    println(user.name)
}
```

Rich errors

```
fun fetchUser(): Json | NetworkError { … }
fun Json.parseUser(): User | ParsingError { … }
fun getUser(): User | UserError = fetchUser()?.parseUser()

fun usage() {
    val user = getUser()
    println(user.name) // error
    if (user !is User) {
        println("Failed to get user")
        return
    }

    println(user.name)
}
```

Ergonomics for chain-calls 💪

Type-safety 👍

Conciseness, no nesting 👍

Rich errors

```
fun fetchUser(): Json | NetworkError { … }
fun Json.parseUser(): User | ParsingError { … }
fun getUser(): User | UserError = fetchUser()?.parseUser()

fun usage() {
    when (val user = getUser()) {
        is User → println(user.name)
        is NetworkError → println("Failed to fetch user")
        is ParsingError → println("Failed to parse user")
    }
}
```

Exhaustive when 😊

Clear error 👍

Union types? Finally???

```
fun getUser(): User | NetworkError | ParsingError
```

Errors are attributes to the main type

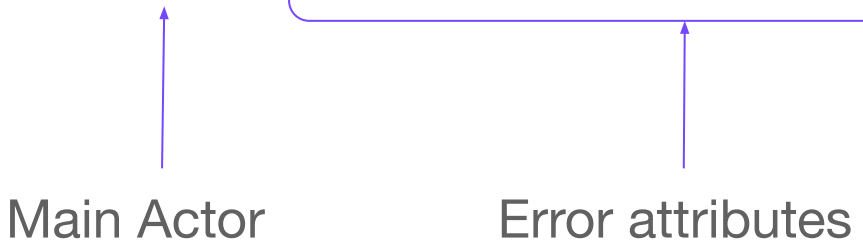
```
fun getUser(): User | NetworkError | ParsingError
```

Main Actor

Error attributes

Errors are attributes to the main type

```
fun getUser(): User | NetworkError | ParsingError
```



- One main actor, but there can be several error attributes
- **No** `Int | String`

Errors are attributes to the main type

```
fun getUser(): User | NetworkError | ParsingError
```

Main Actor

Error attributes

```
getUser().
```

```
lastName                    String
firstName                   String
toString()                  String
javaClass for T in… Class<User>
to(that: B) for … Pair<User, B>
equals(other: Any?)         Boolean
hashCode()                  Int
also { block: (User) → U… User
```

Rich errors: design principles

Rich errors: design principles

- Errors work as values. No special rules for control flow

Rich errors: design principles

- Errors work as values. No special rules for control flow

```
fun exceptionExample() {
    ...
    throw NetworkException()
    ...
}
```

Where will the program execution continue? And will this exception be caught at all?

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases

```
fun process(userName: String) {
   require(userName.isNotBlank()) { "userName must not be blank" }
   require(userName.first().isLetter()) {
      "userName ($userName) must start with a letter"
   }
   …
```

Unrecoverable cases; extensions to the type system

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases
- Exhaustiveness is crucial

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases
- Exhaustiveness is crucial

```
when (val user = getUser()) {
    is User → println(user.name)
    is NetworkError → println("Failed to fetch user")
    is ParsingError → println("Failed to parse user")
}
```

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases
- Exhaustiveness is crucial
- Chain-calls and shortcuts when working with errors matters

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases
- Exhaustiveness is crucial
- Chain-calls and shortcuts when working with errors matters

```
val json = fetchUser()
if (json is Error) {
    ...
}

val parsedUser = json.parseUser()
if (parsedUser is Error) {
    ...
}
```



```
fetchUser()?.parseUse()
fetchUser()!!
```

Rich errors: design principles

- Errors work as values. No special rules for control flow
- Errors are designed for recoverable cases
- Exhaustiveness is crucial
- Chain-calls and shortcuts when working with errors matters
- Keep the type system *sane and polynomial*

Error handling in programming languages

- “Two-value returns”
  - Go, Lua, Julia: `(float64, error)`
  - `try` operator in JavaScript
- Special “wrapped” value and shortcuts:
  - `Rust:` `Result<T, E> + ?`
  - `Swift:` `throws, try/catch + Result`
- `Zig:` error union types: `Error!T, FileNotFound!File`

Rich errors: design

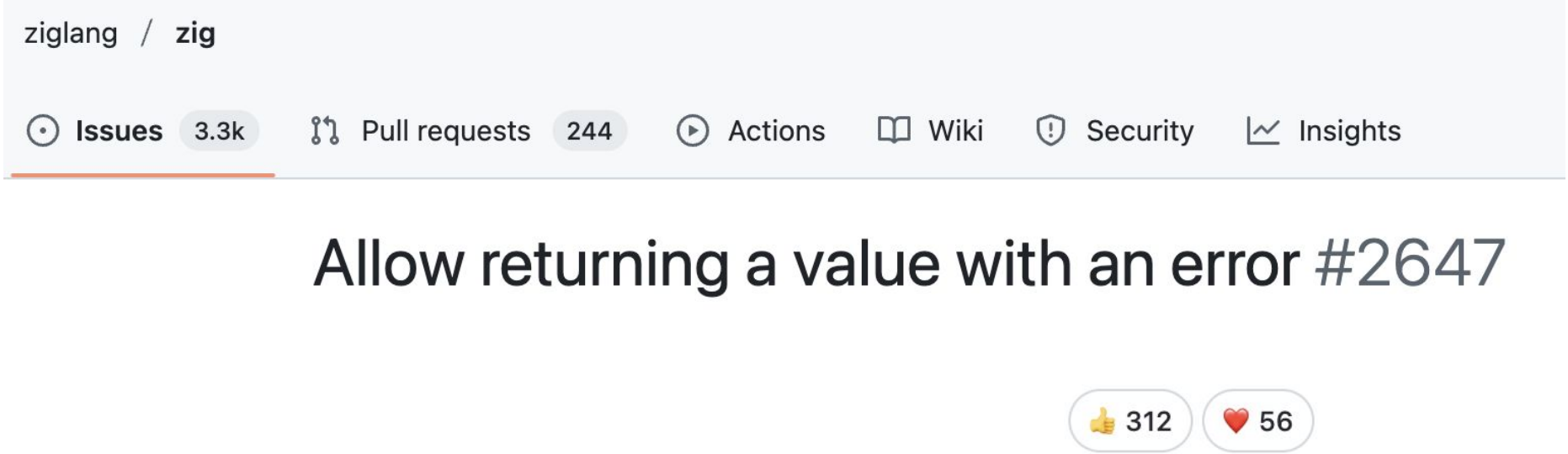
Rich errors: design

- New kind of **`error`** classifiers

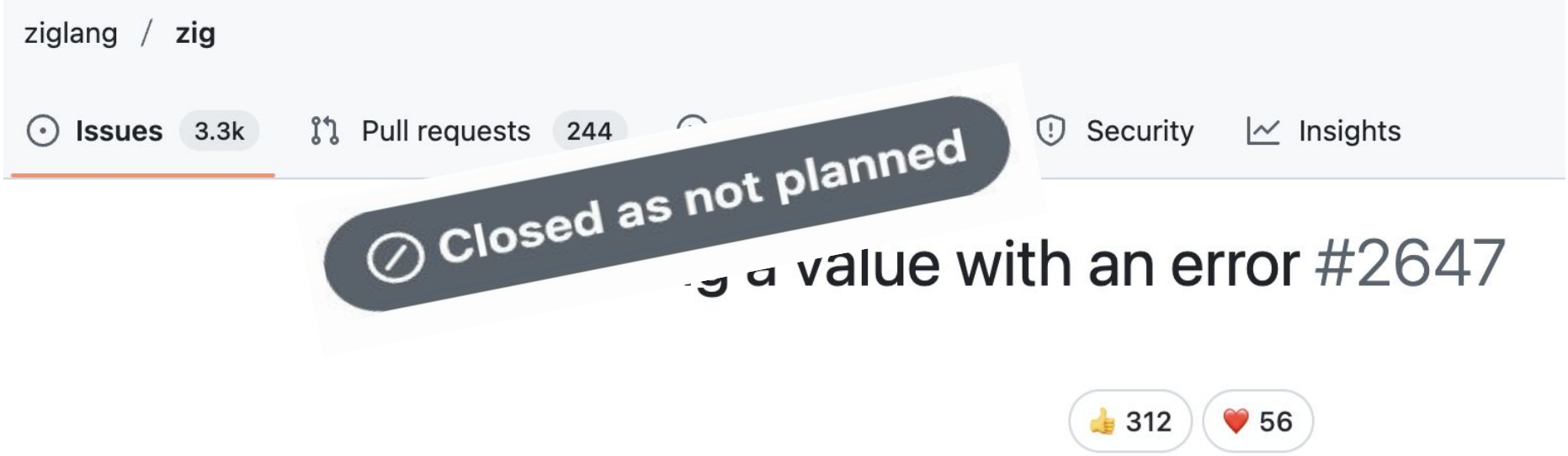
```
error object DuplicateIdError // good!
error class ParsingError(val offset: Int, val message: String) // good!

... return ParsingError(0, "must be an identifier")
```

Zig: error union types



Zig: error union types



Rich errors: design

- New kind of **`error`** classifiers

```
error object DuplicateIdError // good!
error class ParsingError(val offset: Int, val message: String) // good!

... return ParsingError(0, "must be an identifier")

open error class OpenError // compile-time error!
error class GenericError<T> // compile-time error!

typealias CompoundError = DuplicateIdError | ParsingError // good!
```

From sealed hierarchies to errors

```
sealed class FetchingResult<out T> {
    data class Success<T>(val data: T) : FetchingResult<T>()

    sealed class Error : FetchingResult<Nothing>() {
        object NetworkError : Error()
        object NotFound : Error()
        class ValidationError(val reason: String) : Error()
        class Unknown(val exception: Throwable) : Error()
    }
}
```

From sealed hierarchies to errors

```
error object NetworkError
error object NotFound
error class ValidationError(val reason: String)
error class Unknown(val exception: Throwable)

typealias FetchingError =
     NetworkError | NotFound | ValidationError | Unknown

fun <T> fetchData(): T | FetchingError { ... }
```

Rich errors: design

- New kind of **`error`** classifiers
  - Errors are not subtype of `Any?`, but of `Error`

```
error object DuplicateIdError

fun take(a: Any?) {}
fun <T> generic(t: T) {}

fun usage() {
    take(DuplicateIdError) // compile-time error!
    generic(DuplicateIdError) // compile-time error!
}
```

Rich errors: design

- New kind of **`error`** classifiers
  - Errors are not subtype of `Any?`, but of `Error`

```
error object DuplicateIdError

fun take(a: Error) {}
fun <T : Error> generic(t: T) {}

fun usage() {
   take(DuplicateIdError) // good!
   generic(DuplicateIdError) // good!
}
```

Rich errors: design

- New kind of **`error`** classifiers
- New syntax for declaring *disjoint* union types of errors

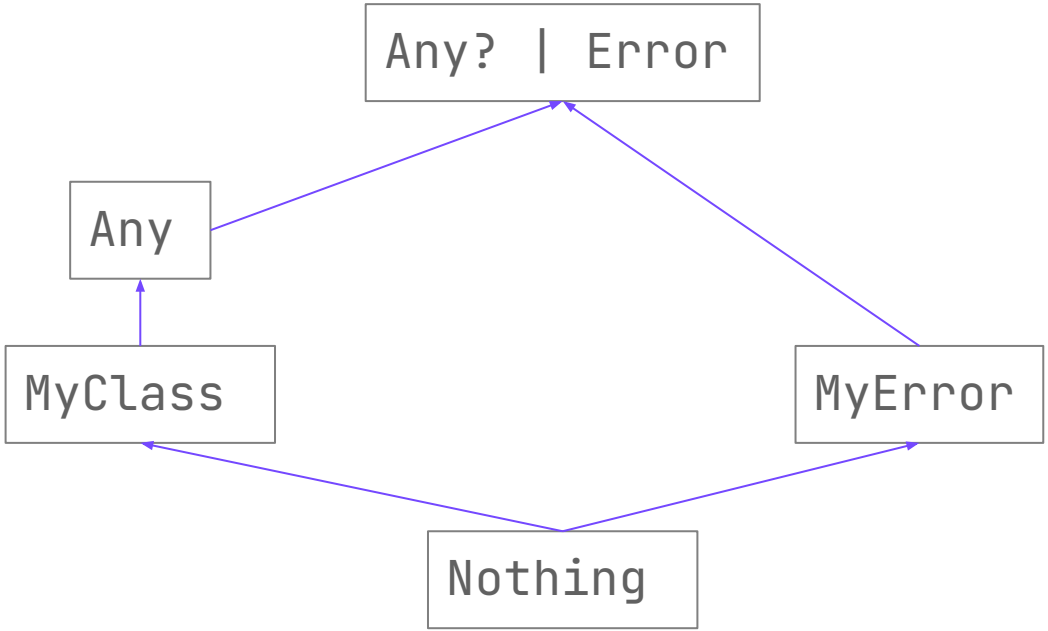
Rich errors: design

- New kind of **`error`** classifiers
- New syntax for declaring *disjoint* union types of errors
  - `Int | DuplicateIdError | ParsingError`
  - `DuplicateIdError | NetworkError`
  - `Int | String` // wrong!

Rich errors: design

- New kind of **`error`** classifiers
- New syntax for declaring *disjoint* union types of errors
  - `Int | DuplicateIdError | ParsingError`
  - `DuplicateIdError | NetworkError`
  - `Int | String` // wrong!
    - Having errors in separate hierarchy allows us to have `Int | Error`
    - Having errors always final allows us to have `Error1 | Error2`

Rich errors: type hierarchy



Bonus feature: in-place tags

Bonus feature: in-place tags

```
fun <T> Sequence<T>.last(predicate: (T) → Boolean): T {
    var last: T? = null
    var found = false
    for (element in this) {
        if (predicate(element)) {
            last = element
            found = true
        }
    }
    if (!found) throw NoSuchElementException("no element")
    @Suppress("UNCHECKED_CAST")
    return last as T
}
```

Bonus feature: in-place tags

```
fun <T> Sequence<T>.last(predicate: (T) → Boolean): T {
    var last: T? = null
    var found = false
    for (element in this) {
        if (predicate(element)) {
            last = element
            found = true
        }
    }
    if (!found) throw NoSuchElementException("no element")
    @Suppress("UNCHECKED_CAST")
    return last as T
}
```

Bonus feature: in-place tags

```
error object NotFound

fun <T> Sequence<T>.last(predicate: (T) → Boolean): T {
    var last: T | NotFound = NotFound
    for (element in this) {
        if (predicate(element)) {
            last = element
        }
    }
    if (last == NotFound) throw NoSuchElementException("no element")
    return last
}
```

Bonus feature: optional values

- We often have to express that “a value might be missing”
  - `Optional<T>` in Java, Swift. `Option<T>` in Rust, `Maybe` in Haskell
  - Nullable types in Kotlin if the container works with non-nullable values…
    - `listOf(null, 42).firstOrNull()`

Bonus feature: optional values

- We often have to express that “a value might be missing”
  - `Optional<T>` in Java, Swift. `Option<T>` in Rust, `Maybe` in Haskell
  - Nullable types in Kotlin if the container works with non-nullable values…
    - `listOf(null, 42).firstOrNull()`

- Since rich errors exist in a separate type hierarchy, optional values can naturally fit into the picture, leveraging the ergonomics of nullable types:
  - `Optional<T> -> T | NotFound`
  - `fun <T> List<T>.first(): T | NotFound`

Rich errors: design

- New kind of **`error`** classifiers
- New syntax for declaring *disjoint* union types of errors
- Shortcuts and chain-calls

Safe-call, nulls

```
exprA ?. exprB ?. exprC
```

Safe-call, nulls

```
exprA?.exprB?.exprC
```

TypeA?

Safe-call, nulls

```
exprA?.exprB?.exprC
```

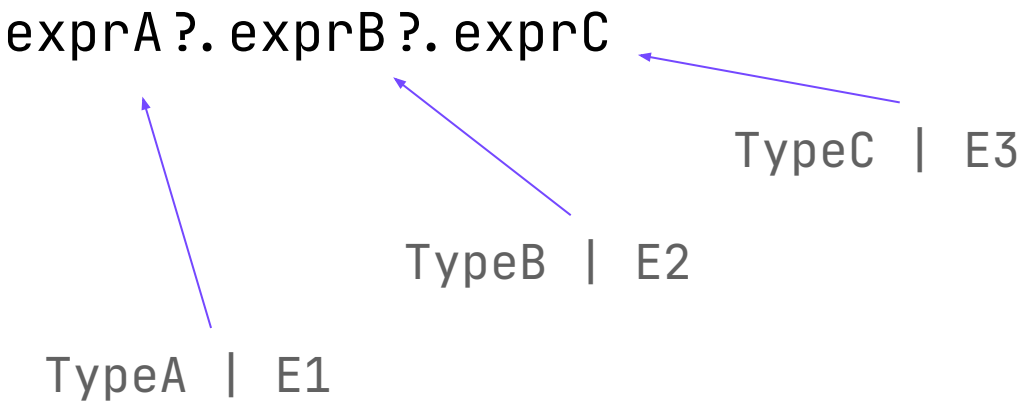
TypeB? — **null** from exprA *or* from exprB

Safe-call, nulls

```
exprA?.exprB?.exprC
```

TypeC? — **null** from exprA *or* from exprB *or* exprC

Safe-call, rich errors



Safe-call, rich errors

```
exprA?.exprB?.exprC
```

```
TypeA | E1
```

Safe-call, rich errors

```
exprA ?. exprB ?. exprC
```

```
TypeB | E1 | E2 — E1 from exprA, E2 from exprB
```

Safe-call, rich errors

```
exprA ?. exprB ?. exprC
```

```
TypeC | E1 | E2 | E3
```

Safe-call, rich errors

```
exprA ?. exprB ?. exprC
```

```
TypeC | E1 | E2 | E3
```

Main “actor”, we can use its functionality

Fantasy: union types

```
interface A {
    fun foo()
}

interface B {
    fun bar()
}

fun test(aOrB: A | B) {
    aOrB.<caret> // which methods I can access in a type-safe manner?
}
```

Fantasy: union types

```
interface A {
    fun foo()
}

interface B {
    fun bar()
}

fun test(aOrB: A | B) {
    aOrB.<caret> // just methods from their common supertype — Any
}
```

Union types, Result, Either…

```
interface A {
    fun foo()
}

interface B {
    fun bar()
}

fun test(aOrB: A | B) {
    aOrB.<caret> // just methods from their common supertype — Any
}
```

Rich errors

```
interface A {
    fun foo()
}

interface B {
    fun bar()
}

fun test(aOrError: A | Error, bOrError: B | Error) {
    aOrError?.<caret> // can access foo
    bOrError?.<caret> // can access bar
}
```

Rich Errors: !! operator

```
fun fetchUser(): User? = null

fetchUser()!! // NullPointerException!
```

Rich Errors: !! operator

```
fun fetchUser(): User | NotFound = NotFound

fetchUser()!! ⇝ fetchUser().throw() // KotlinException(NotFound)
```

Rich Errors: !! operator

```
fun fetchUser(): User | NotFound = NotFound

fetchUser()!! ⇝ fetchUser().throw() // KotlinException(NotFound)


error class NotFound {
    operator fun throw(): Nothing {
        throw NotFoundException(...)
    }
}
```

Rich errors: elvis

- No `?:` operator, no way to handle the error on the right hand side

```
expr ?: return ...
```

Rich errors: elvis

- No `?:` operator, no way to handle the error on the right hand side

```
inline fun <T, E : Error> (T | E).ifError(default: (E) → T): T {
    return if (this is Error) default(this) else this
}


val result = expr.ifError { return }

expr.ifError { e →
    when (e) {
        is ParsingError → ...
        ...
    }
}
```

What about exceptions?

- Kotlin doesn’t have checked exceptions for a purpose
- Primary use exceptions should be for unrecoverable cases

From exceptions to errors

```
// stdlib

error class ThrowableError(val e: Throwable)

inline fun <R> runCatchingAsError(block: () → R): R | ThrowableError {
    return try {
        block()
    } catch (e: Throwable) {
        ThrowableError(e)
    }
}
```

From exceptions to errors

```
// stdlib

error class ThrowableError(val e: Throwable)

inline fun <R> runCatchingAsError(block: () → R): R | ThrowableError {
    return try {
        block()
    } catch (e: Throwable) {
        ThrowableError(e)
    }
}
```

And a version to handle CancellationException!

From exceptions to errors

```
fun tryRead(path: Path): List<String> | ThrowableError =
    runCatchingAsError {
        Files.readAllLines(path)
    }
```

tl;dr

- Rich Errors are designed to handle recoverable error cases
  - Exceptions for everything else
- Generalization of nulls-as-errors pattern
  - Including ergonomics of ?. and !!
- Built-in support for patterns around Result, Optional
- In Progress, expect KEEPs about errors this summer

@mzarechenskiy

Thank You, and Don’t Forget to Vote