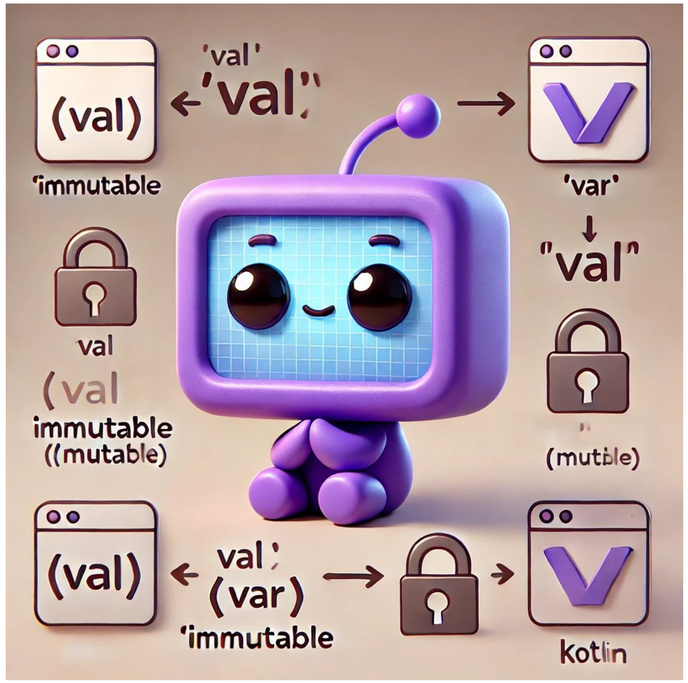
JETBRAINS

Better Immutability in Kotlin

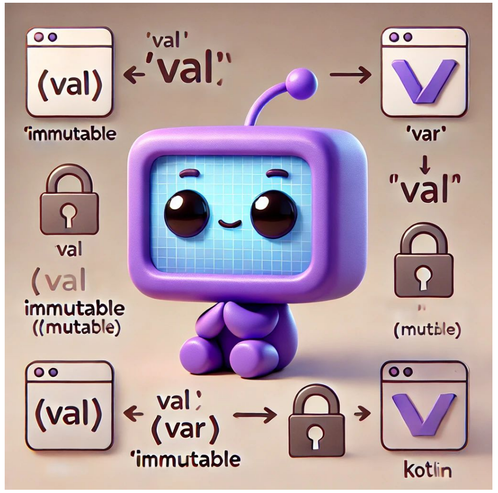
Building on top of Valhalla

Marat Akhin
Researcher @ **Kotlin Language Evolution**

Agenda


(val)
'val'
'val'
'immutable
'var'
"val"
val
(val
immutable
((mutable)
(mutible)
(val)
val
(var)
'immutable
Kotlin


Why should immutabilty be in Kotlin?
Safety, consislity
Sanfetty sonsulity
Kotlin



What Kotlin already has?

(Inline) value classes

```
@JvmInline
value class ZipCode(private val value: String)
```

- Restricted value types (also known as inline types)
  - A single `val` (aka read-only aka `final`) property allowed
  - No identity guarantees
- They are inlined when possible as an optimization

- **Zero-cost wrappers** + *shallow immutable value types*

Inline value class = box + ...

```java
public final class ZipCode {
  private final String value;

  public String toString();
  public int hashCode();
  public boolean equals(Object other);

  private ZipCode(String value);

  // public String getValue();

  // ...
}
```

Inline value class = … + inlining

```
public final class ZipCode {
  // ...

  public static String toString-impl(String value);
  public static int hashCode-impl(String value);
  public static boolean equals-impl(String value, Object other);
  public static final boolean equals-impl0(String v1, String v2);

  public static String constructor-impl(String value);
  public static final ZipCode box-impl(String value);
  public final String unbox-impl();
}
```

Inlining is an optimization

```kotlin
@JvmInline
value class Foo(val i: Int) : I

fun asInline(f: Foo) {}

fun test() {
   val foo: Foo = Foo(42)
   asInline(foo) // No additional
                 // allocations
}
```

```java
public void asInline-GWb7d6U(int i);

public void test() {
   int foo = Foo.constructor-impl(42);
   asInline-GWb7d6U(foo);

}
```

Boxing is “the fallback”

```kotlin
fun <T : Any> asGeneric(x: T) {}
fun asInterface(i: I) {}
fun asNullable(i: Foo?) {}


fun test() {
   val foo: Foo = Foo(42)
   asGeneric(foo)    // Boxing
   asInterface(foo) // Boxing
   asNullable(foo)  // Boxing
}
```

```java
public void asGeneric(@NotNull Object x);
public void asInterface(@NotNull I i);
public void asNullable-N3I3QIo(
   @Nullable Foo i);

public void test() {
  int foo = Foo.constructor-impl(42);
  asGeneric(Foo.box-impl(foo));
  asInterface(Foo.box-impl(foo));
  asNullable-N3I3QIo(Foo.box-impl(foo));
}
```

Problem: mangling

- We want to inline them ⇒ we need to mangle declarations which work with them
- Declarations are mangled ⇒ Java interoperability suffers

```
public void asNullable-N3I3QIo(@Nullable Foo i);

public void properJavaMethod() {
  int myInlinedFooValueWhichIsTrustworthy = 42;
  asNullable-N3I3QIo(Foo.box-impl(myInlinedFooValueWhichIsTrustworthy));
  // Nope, cannot call (x2)
}
```

More problems: one val is not enough

- Users have been asking for “multi-field value classes“ for a long time
  - https://youtrack.jetbrains.com/issue/KT-1179 (January 27th, 2012)
- Inlining not one, but multiple fields is a significant jump in implementation complexity
  - You need to scalarize one-to-many and not one-to-one

A lot of problems, is there no hope?

Solution: @JvmExposeBoxed

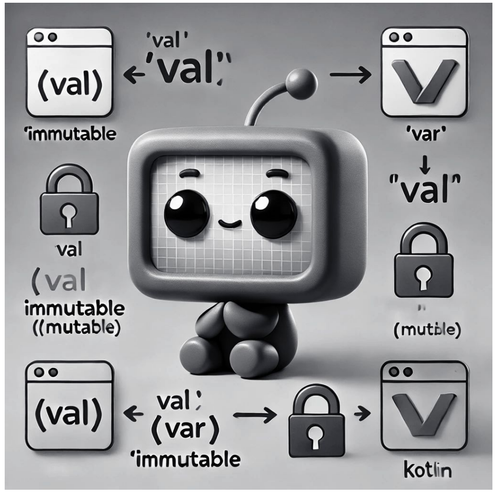
```fun
@JvmExposeBoxed // Adapt mangling
@JvmInline
value class Foo(val i: Int) : I

@JvmExposeBoxed // Adapt mangling
fun asInline(f: Foo) {}
```

```fun
// …
@JvmExposeBoxed
public Foo(int i) {
  this(i, (BoxingConstructorMarker)null);
  constructor-impl(i);
}
// …
@JvmExposeBoxed
public void asInline(@NotNull Foo f) {
  asInline-GWb7d6U(f.unbox-impl());
}
// …
```

Solution: ???

- We can solve problems around things we (**Kotlin**) control
  - We can enable or disable or change mangling
- As Kotlin compiles to the JVM bytecode, there are things we do not fully control
  - We cannot fully optimize boxing
  - We cannot have inline multi-field value classes
  - We cannot do fully efficient value arrays

Kotlin user space	Kotlin language space	**Kotlin language space**
	JVM user space	**JVM bytecode space**



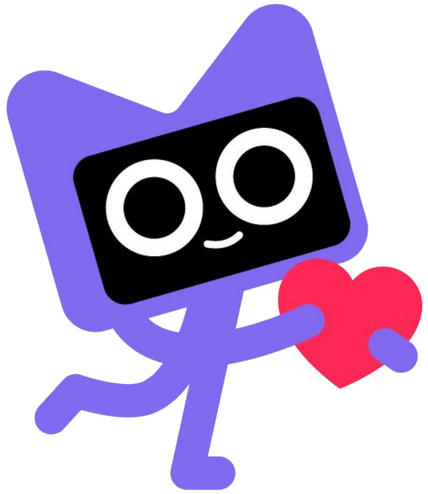
Project Valhalla

JEP 401

- Value classes and objects on the JVM platform
- Three main goals
  - Provide types which opt-out of identity
  - Migrate effectively value-based classes from the JDK to actual value classes
  - Allow better run-time optimizations of these value classes and their objects

JEP 401

- Value classes and objects on the JVM platform
- Three main goals
  - **Provide types which opt-out of identity**
  - Migrate effectively value-based classes from the JDK to actual value classes
  - **Allow better run-time optimizations of these value classes and their objects**



JEP 401 value classes

```
value record Contact(String name, Address address) {}
value record Address(String street, String zipCode) {}
```

- JEP 401 value class == Kotlin `@JvmInline` value class
- Also we do not need to do anything on the Kotlin side
  - No mangling
  - No manual inlining
- Also we get multiple `val`s for free

What works

Immutability (with no identity) / value semantics

- For immutable things, identity “leaks” the mutability of the value box
  - Two structurally equal immutable things are not truly interchangeable anymore
- Example: Jetpack Compose
  - TL;DR: reactive UI framework
  - Needs to track the UI state changes

```
@Composable
fun Home(..., contact: Contact, ...) {
   ContactList(...) {
      ContactDetails(..., contact, ...)
      // Do I need to redraw this UI?
   }
}
```

What works

Single representation

- No need to have two flavors of one thing: boxed and inlined
- Optimizations are done by the JVM
  - It can do them better
  - It can do them at runtime
- Inlining is an optimization == we can drop it

```
~~@JvmInline~~
value class ZipCode(private val value: String)
```

Single representation

```
public final value class ZipCode {
  // ...

  public static String toString-impl(String value);
  public static int hashCode-impl(String value);
  public static boolean equals-impl(String value, Object other);
  public static final boolean equals-impl0(String v1, String v2);

  public static String constructor-impl(String value);
  public static final ZipCode box-impl(String value);
  public final String unbox-impl();
}
```

What works

Default implementations

- JVM provides default implementations of `equals` / `hashCode`
  - They do what Kotlin-generated implementations do
- We can skip generating them ourselves

```
~~@JvmInline~~
value class ZipCode(private val value: String)
```

Default implementations

```
public final value class ZipCode {
  private final String value;

  public String toString();
  public int hashCode();
  public boolean equals(Object other);

  public ZipCode(String value);

  // ...
}
```

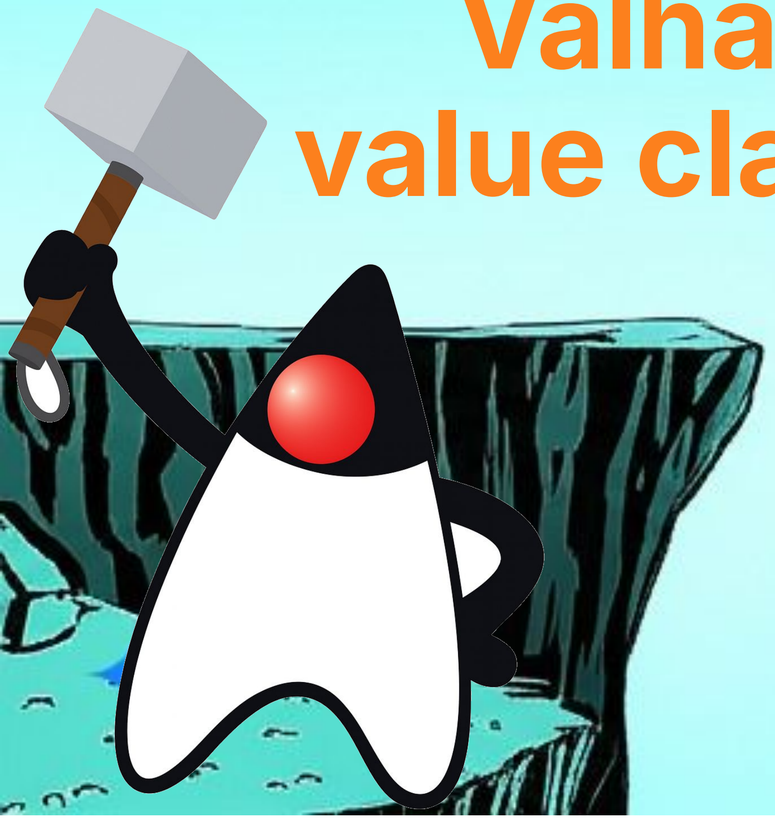
What works

Ability to have several fields

- “Inlining not one, but multiple fields is a significant jump in implementation complexity” is not true anymore
  - If you do not need to do anything special in Kotlin-language-space, there is no complexity

```
value class Book(val title: String, val isbn: ISBN,
   val publisher: PublishingHouse)
```

Valhalla
value classes



Kotlin
value classes

What does
not work?

What does not work

Early initialization

- This is a new concept for both Java and Kotlin
- This is a not-yet-fully-finalized concept
  - e.g., some important changes on ~~2025/07/22~~ 2025/08/05

For convenience, the early construction rules are relaxed by this JEP to allow the class's fields to be *read* as well as *written*—both references to the field `name` in the above constructor are legal.

This scheme is also appropriate for identity records, so this JEP modifies the language rules for records such that their constructors always run in the early construction phase.

Early initialization

In **Kotlin** initialization **begins** with `super(...)`

- i.e. currently everything happens in late construction phase
- Local solution: value classes are also a new concept
  - They can have different initialization rules
  - Aka everything happens in early construction phase and value class initialization **ends** with `super(...)`
- But that's not enough 😭

Early initialization

Global solution: **???**

- We have several **Kotlin**-specific problems
  - E.g., reading a property == calling its getter

```
class HexStringParser(private val EXPONENT_WIDTH: Int, ...) {
    init {
        this.EXPONENT_BASE = (-1L shl (EXPONENT_WIDTH - 1)).inv()
        this.MAX_EXPONENT = (-1L shl EXPONENT_WIDTH).inv()
        // the rhvs here access class properties and not ctor parameters
    }
    // ...
}
```

Early initialization and properties

```
class HexStringParser(private val EXPONENT_WIDTH: Int, ...) {
    init {
        this.EXPONENT_BASE = (-1L shl (EXPONENT_WIDTH - 1)).inv()
        this.MAX_EXPONENT = (-1L shl EXPONENT_WIDTH).inv()
        // the rhvs here access class properties and not ctor parameters
    }
    // ...
}
```

- We can introduce new syntax to access ctor parameters — that’s awkward
- We can change the resolve when it matters — that’s fragile and confusing
- We can change the resolve everywhere — that’s a breaking change

Early initialization

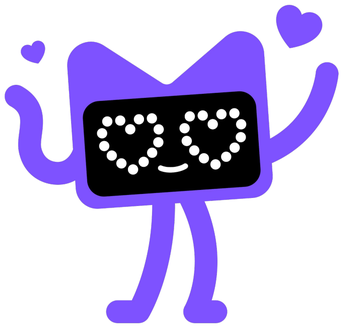
Global solution: **???**

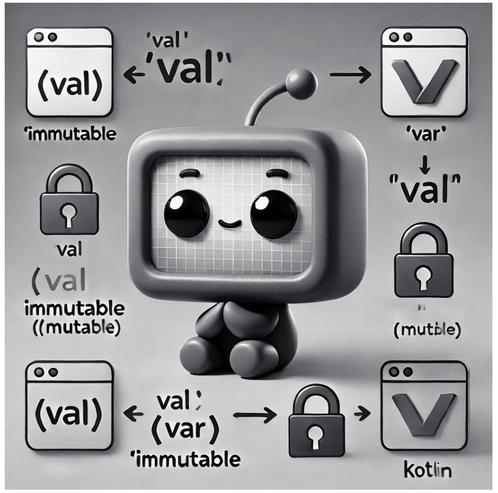
- We have several **Kotlin**-specific problems
  - E.g., how to mark early construction phase

- In Kotlin every class can (and usually does) have a primary (canonical) constructor
  - But some classes have only secondary (regular) constructors
- Same syntax for early construction phase does not work for one or the other case
- Different syntaxes look awkward

TL;DR: project Valhalla is our foundation

- When project Valhalla is released, Kotlin gets much better value classes for free
  - We solve all our current problems
  - We can solve the new Valhalla-induced problems
    - Because they are around things we (**Kotlin**) control
    - The solutions may not be easy, but they are definitely possible

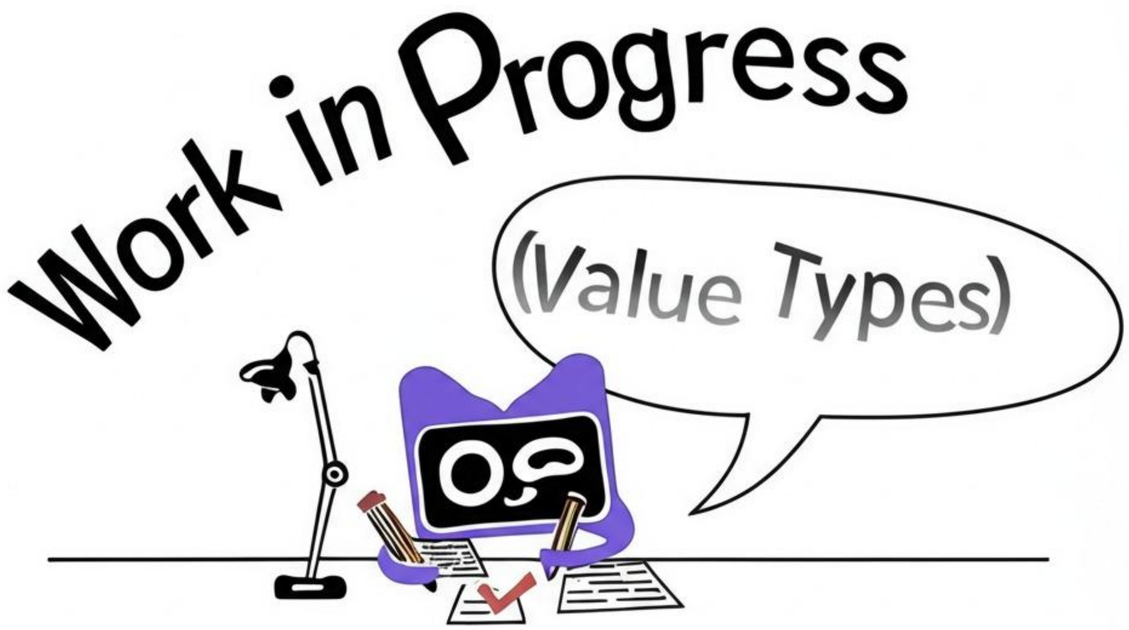




(Even) better immutability

Another point of view

- JEP 401 value class == shallow immutable value type
- Additional **Kotlin-specific** goals
  - Provide ergonomic updates of immutable data
  - Introduce “just enough” value semantics
  - Support deep immutability

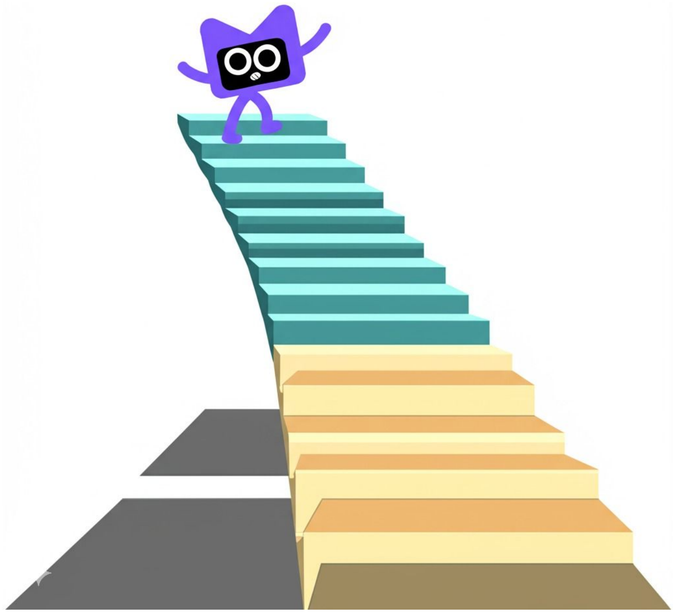


(Un)ergonomic updates

```
value class Book(val title: String, val isbn: ISBN,
   val publisher: PublishingHouse)

fun processBooks(books: ImmutableList<Book>) {
   // ...
   val fixedBooks = books.map {
      it.copy(title = capitalize(it.title))
      // or
      Book(title = capitalize(it.title), isbn = it.isbn,
           publisher = it.publisher)
   }
   // ...
}
```

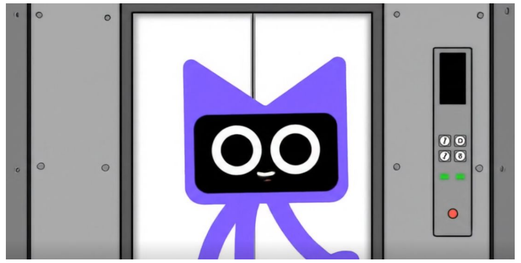
The “copy” ladder

```
fun processBooks(books: ImmutableList<Book>)
    // …
    val fixedBooks = books.map {
        it.copy(
            title = capitalize(it.title),
            isbn = it.isbn.copy(eanPrefix = 979),
            publisher = it.publisher.copy(
                address = it.publisher.address.copy(
                    …
                )
            )
        )
    }
    // …
}
```



Mutable code reads better

```
fun processBooks(books: ImmutableList<Book>)
    // …
    val fixedBooks = books.map {
        it.title = capitalize(it.title)
        it.isbn.eanPrefix = 979
        it.publisher.address.zipCode = "..."
        it
    }
    // ...
}
```



```
data class Book(var title: String, var isbn: ISBN,
    var publisher: PublishingHouse)
```

Problems of explicit copying

- Good code has more boilerplate than bad code
- The intent is hidden behind the ceremony

- Our users have already *“tasted the forbidden fruit”*
  - Swift / Go / C# / C++ have value types aka structs
  - With structs your code does not have any explicit-copy-related boilerplate
- Can we do something here?

copy var properties

```
value class Book(copy var title: String, copy var isbn: ISBN,
    copy var publisher: PublishingHouse)
```

- Kotlin has read-only `val` properties and mutable `var` properties
- We add copyable `copy var` properties
  - If you update them, the update also copies the outer object of the `copy var` and updates the reference
  - In other words, `copy var` is a built-in lens for its property
  - Or a built-in “wither” (JEP 468)

copy var properties

```
public final value class Book {
  private final String title;

  public String getTitle();
  public Book withTitle(String newTitle);

  // …
}
```

```
  book.title = fix(book.title)
```

```
  book = book.withTitle(fix(book.getTitle()))
```

copy vars are compiled to withers

Why withers and not something else?

- A known pattern in the Java ecosystem (e.g., see Lombok's `@With`)
  - Gives us convenient Java interop for free
- Has better binary compatibility story
  - Compared to `copy` / explicit ctor calls

Are withers the right choice?

Why **NOT** withers and something else?

- Derived record creation plans to go via record constructors

(JEP 468) Evaluation, in detail

A derived record creation expression is evaluated as follows:

…

5. Create a new instance of record class *R* as if by **evaluating a new class instance creation expression** (`new`) with the compile-time type of the origin expression and an argument list containing the local component variables

Multiple updates

```
book = Book(fix(book.getTitle()), book.getIsbn(), book.getPublisher())
book = Book(book.getTitle(), ISBN(..., 979), book.getPublisher())
```

```
book = Book(
    fix(book.getTitle()),
    ISBN(..., 979),
    book.getPublisher())
```

- Optimizing sequence of direct constructor calls
  - If JIT is able to do that for us, we may reconsider the `copy var` compilation scheme 🙏

copy var variables

```
var origin = Book(...)
var copy = origin
copy.title = fix(copy.title)
```

- `copy var` changes your value by creating a copy and assigning it to the variable

- Kotlin is reference-based through and through
  - If variables reference each other, we expect changes to one to be reflected in the other
  - Ergonomic updates with `copy var`s hide this
    - This is somewhat intentional, but unfortunate

```
var origin = mk<Book>()
copy var copy = origin
copy.title = fix(copy.title)
```

- Explicitly marking the reference, which allows `copy var` semantics, makes it visible in the code
- At the same time, it does not create too much boilerplate

copy var variables

```
var origin = Book(...)
var copy = origin
copy.title = fix(copy.title)
```

- `copy var` changes your value by creating a copy and assigning it to the variable

- Kotlin is reference-based through and through
  - If variables reference each other, we expect changes to one to be reflected in the other
  - Ergonomic updates with `copy var`s hide this
    - This is somewhat intentional, but unfortunate
-

```
var origin = Book(...)
copy var copy = origin
copy.title = fix(copy.title)
```

- Explicitly marking the reference, which allows `copy var` updates, makes it visible in the code
- At the same time, it does not create too much boilerplate

Multiple updates as a function

You should be able to
abstract multiple updates together

```
copy var book = books.first { ... }
book.title = capitalize(book.title)
book.isbn.eanPrefix = 979
```

```
var book = books.first { ... }
book = book.normalized()
```

```
fun Book.normalized(): Book {
    copy var self = this
    // do updates on self
    return self
}
```

copy functions

```
copy var book = books.first { ... }
book.normalize()
```

```
copy fun Book.normalize(): Unit {
    // copy var this
    title = capitalize(title)
    isbn.eanPrefix = 979
}
```

Feature interaction

```
value class Book(copy var title: String, copy var isbn: ISBN,
   copy var publisher: PublishingHouse)
```

- `copy var` properties work great in isolation
- What about feature interaction?

When your code uses three Kotlin language features at the same time, something will break. ©

Anyone from the Kotlin development team

Feature interaction

```
value class Book(copy var title: String, copy var isbn: ISBN,
   copy var publisher: PublishingHouse)
```

- `copy var` properties work great in isolation
- What about feature interaction?
  - Lambdas

It’s not a JVMLS talk if there are no lambdas. © *Anyone from the JVMLS*

When your code uses three Kotlin language features at the same time, something will break. ©
Anyone from the Kotlin development team

copy vars and lambdas

```
value class Book(copy var title: String, copy var isbn: ISBN,
    copy var publisher: PublishingHouse)
```

```
copy var book = Book(...)
loadBookTitleOrNull()?.let {
    book.title = it
}
```

```
copy var book = Book(...)
val title = loadBookTitleOrNull()
if (title ≠ null) { book.title = title }
```

- In Kotlin, a lot of things are actually lambdas
- And people use them to extend the language

copy vars and lambdas

```
public inline fun <T> T.copyWith(block: T.() → T): T = this.block()


val book = Book(...)
val book2ndEdition = book.copyWith {
   copy var self = this
   self.title = "${it.title} (2nd edition)"
   self
}
```

This is the same boilerplate as with multiple updates, now in the shape of a lambda

copy lambdas

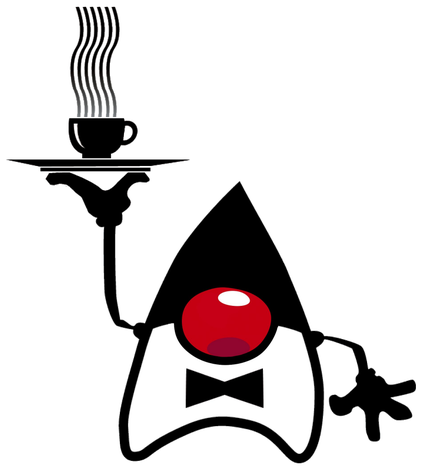
```
public inline fun <T> T.copyWith(block: copy T.() → Unit): T {
   copy var self = this
   self.block()
   return self
}

val book = Book(...)
val book2ndEdition = book.copyWith { /* copy var this */
   title = "${it.title} (2nd edition)"
}
```

copy lambdas

```
public inline fun <T> T.copyWith(block: copy T.() → Unit): T
```

- `copyWith` stdlib function (final name TBD) means you can opt to disallow using explicit `copy var`s in your code (e.g., with a linter) and use “wither”-like style with explicit copying

- If there are other similar use-cases, people can use `copy` lambdas for them as well



copy vars and existing lambdas

Function	Object reference	Return value	Is extension function
`let` ↗	`it`	Lambda result	Yes
`run` ↗	`this`	Lambda result	Yes
`run` ↗	-	Lambda result	No: called without the context object
`with` ↗	`this`	Lambda result	No: takes the context object as an argument.
`apply` ↗	`this`	Context object	Yes
`also` ↗	`it`	Context object	Yes

copy vars and scope functions

```
val numbers = mutableListOf("one", "two", "three")
numbers
    .also { println("The list elements before adding new one: $it") }
    .add("four")
```

```
val str: String? = loadStringOrNull()
val length = str?.let { it: String →
    println("'let' called on $it")
    processNonNullString(it)
    it.length
}
```

copy vars and scope functions

```
public inline fun <T> T.also(block: (T) → Unit): T {
    contract { callsInPlace(block, InvocationKind.EXACTLY_ONCE) }
    block(this)
    return this
}


copy var book = Book(...)
book.also { it: Book →
    it.title = "${it.title} (2nd edition)" // Compilation error!
}
```

scope functions are not polymorphic

```
public inline fun <T> T¹.also(block: (T²) → Unit): T³ = …


copy var book = Book(...)
book.also { it: Book →
   it.title = "${it.title} (2nd edition)" // Compilation error!
}
```

- All of T^1, T^2 and T^3 actually have the same `copy-var`-ness
  - But this fact is not known without knowing the implementation details

We do not have a good solution here at the moment

Feature interaction

```
value class Book(copy var title: String, copy var isbn: ISBN,
    copy var publisher: PublishingHouse)
```

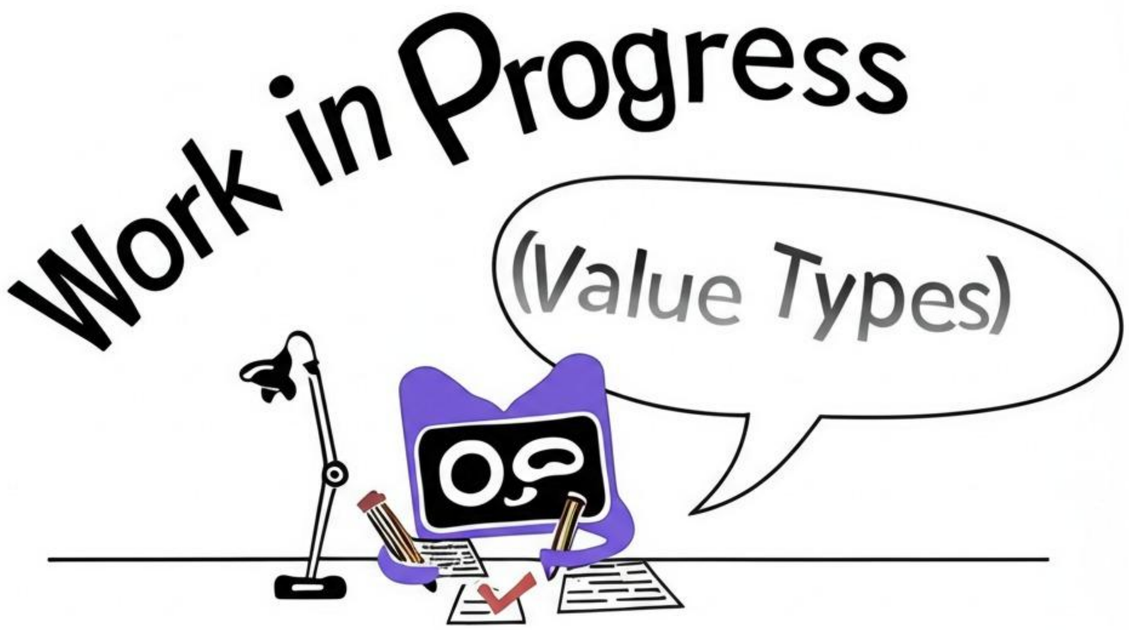
- `copy var` properties work great in isolation
- What about feature interaction?
  - 🔎 Lambdas
  - Inheritance
  - Capturing
  - Reference types
  - ...

When your code uses three Kotlin language features at the same time, something will break. ©

Anyone from the Kotlin development team

Recap

- JEP 401 value class == shallow immutable value type
- Additional **Kotlin-specific** goals
  - ✅ Provide ergonomic updates of immutable data
  - ➡️ Introduce “just enough” value semantics
  - ➡️ Support deep immutability
    - JVMLS 2026?



All in all

- Kotlin already has limited forms of immutability
  - A lot of support is implemented in our language-space
- Project Valhalla allows us to do less, because things move to the JVM-runtime-space
  - It provides a solid foundation for immutability on the JVM
- On top of project Valhalla, we can build even more immutability
  - Ergonomic updates
  - “Almost proper” value semantics
  - Deep immutability