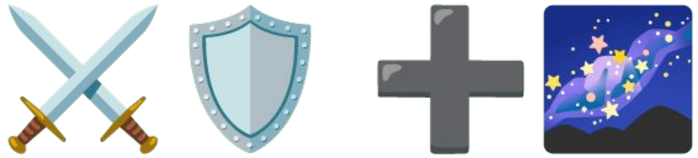
KotlinConf 2025

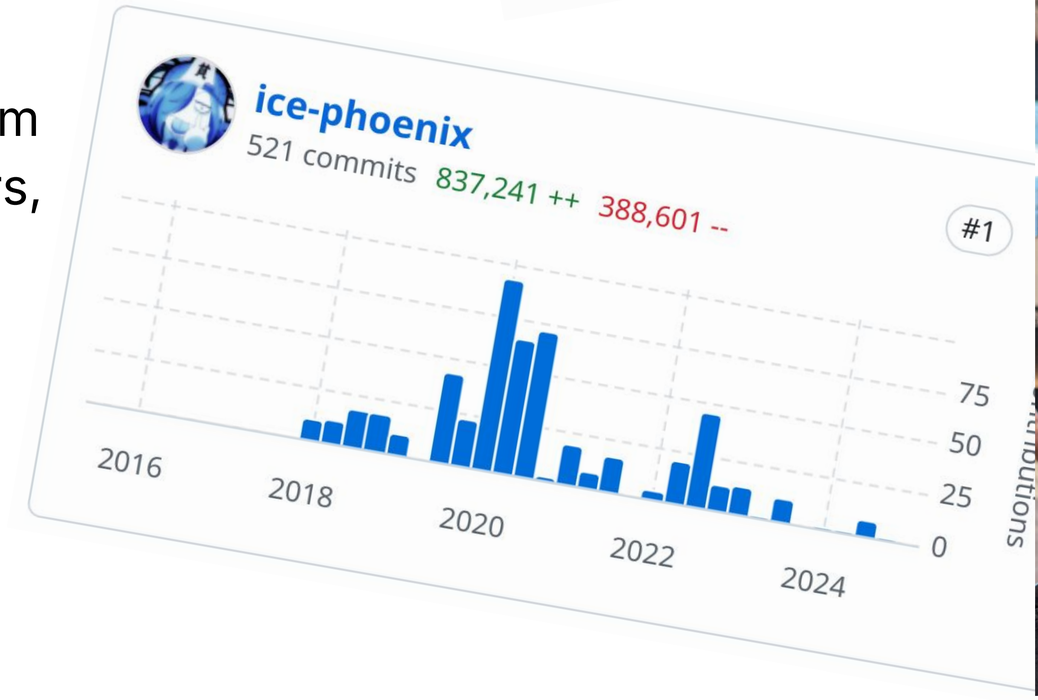


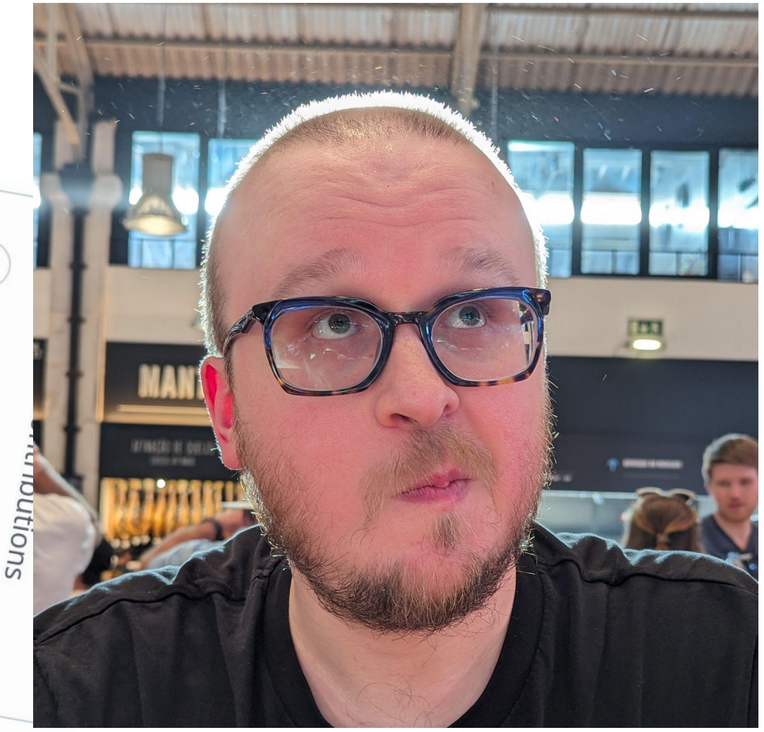
Marat Akhin

@tau_phoenix

/usr/bin/whoami

- Researcher @ Kotlin Language Evolution team
- Worked on the Kotlin language specification and know a thing or two about how Kotlin works
- Also did research in program analysis of different flavours, including Kotlin compiler fuzzing



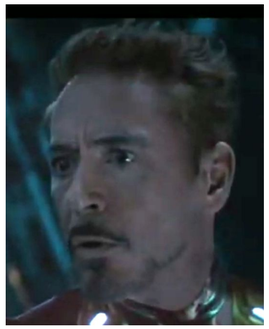


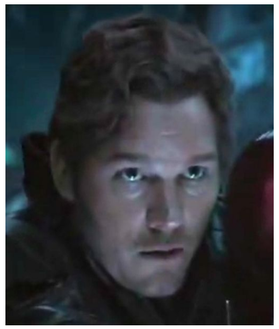
What the talk is / isn’t about

- ✅ New immutability features coming to Kotlin
  - In the form of **(deeply) immutable value classes**, checked and enforced by the compiler
  - Also as **copy vars** as a way to ergonomically update immutable data
- 🙅‍♂️ How immutable data helps your software design
  - For this, go and watch “***Good Old Data***” talk by Andrey Zaytsev
  - Right after this talk (14:00 – 14:45) at *Aud 11+12*
- 🙅‍♂️ Other aspects of immutability


MARVEL
IMMUTABILITY
UNIVERSE

Avengers: Immutability Wars (2018)

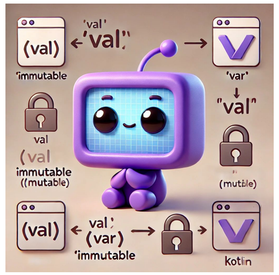




What is Immutability?

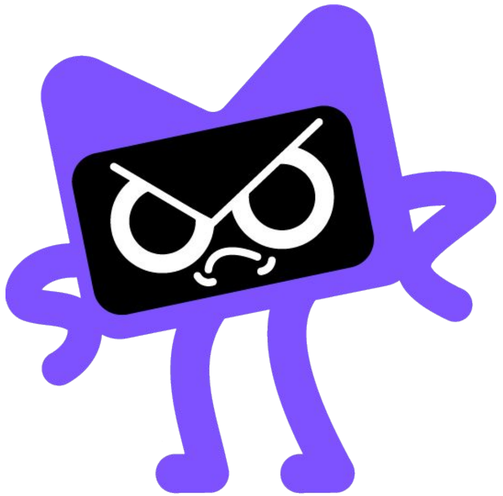
Where is Immutability?

Why is Immutability?

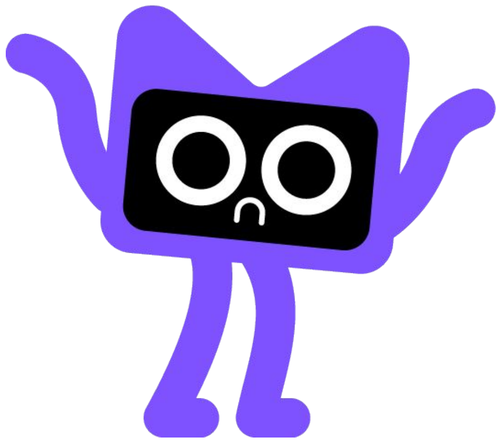


What is immutability?

Immutability is when something cannot be changed



Immutability is when something cannot be changed



Something is...

A property of a (data) class

```
data class Person(val name: String, val age: Int)


fun test() {
    val me = Person("Marat", 39)
    me.age = 25 // [VAL_REASSIGNMENT] 'val' cannot be reassigned.
}
```

Something is...

A collection from the standard library

```
fun test() {
    val bestKotlinVersions = listOf("1.3.72", "1.9.25", "2.0.0")
    bestKotlinVersions.add("3.0.0")
    // [UNRESOLVED_REFERENCE] Unresolved reference 'add'.
}
```

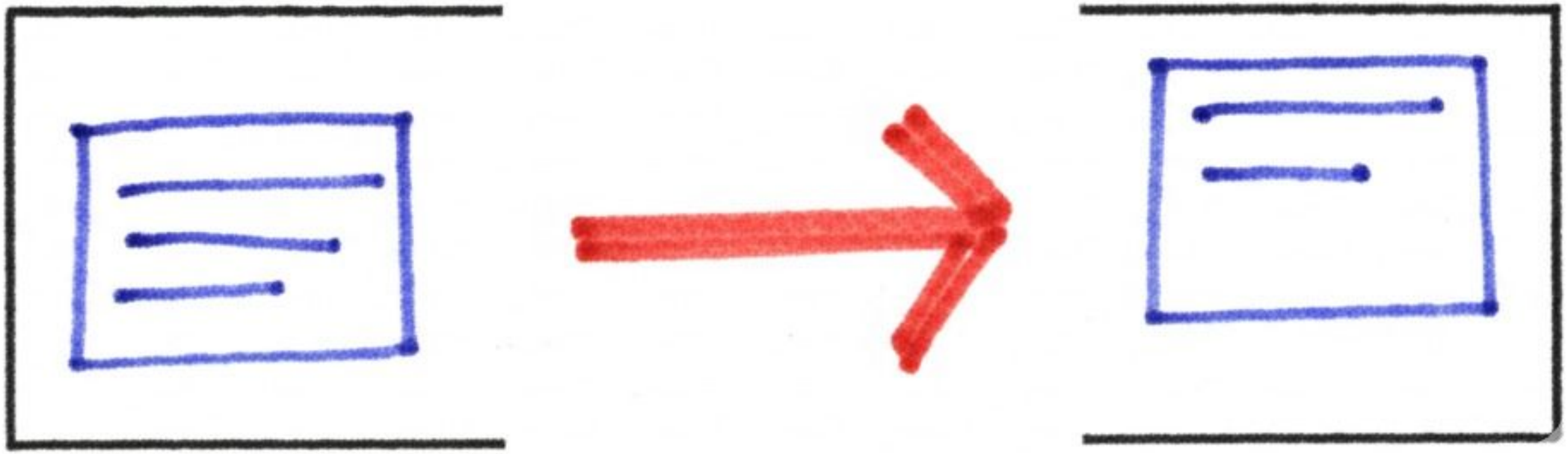
Something is...

Actually anything

- Any entity in your programming language
  - An interface
  - A type parameter
  - A property
  - An anonymous object literal
  - ...

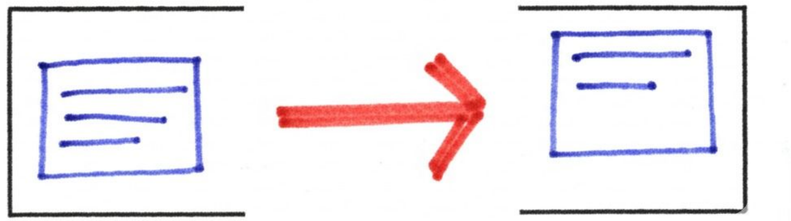
Controlled **manually** (by the developer not making mistakes) or **automatically** (by compiler enforcing proper use)

Cannot be changed means…

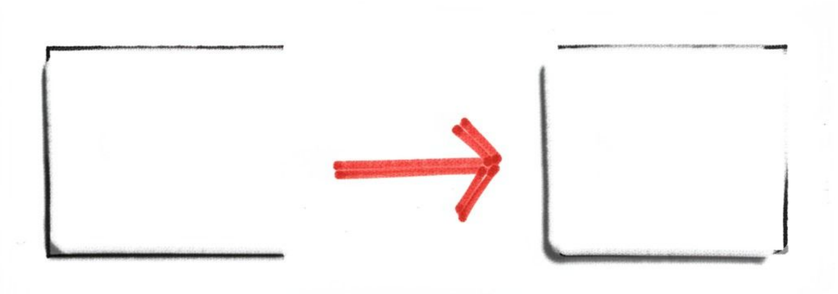


Cannot be changed means…

- Your *physical* state cannot be changed
- Also known as **concrete** state
- Informally, your “bits-and-bytes” are frozen
  - No matter how you view the concrete state, the observable results will always be the same
- Example: raw byte array, `java.lang.String`

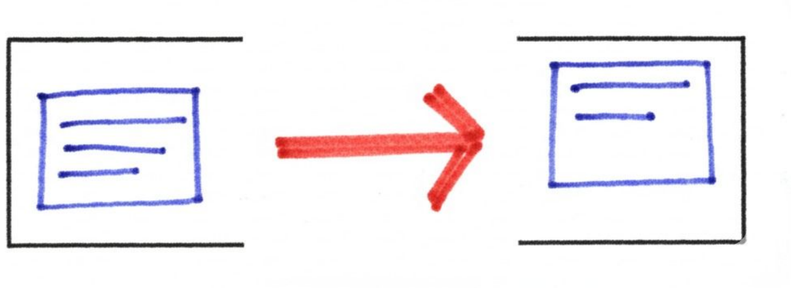


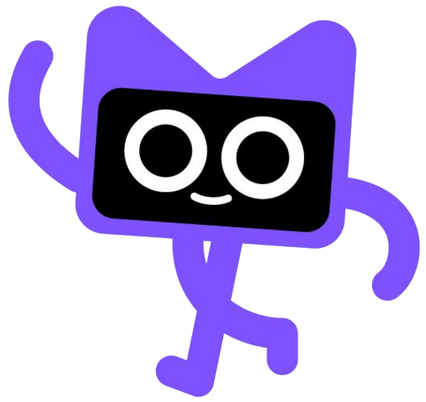
Cannot be changed means...

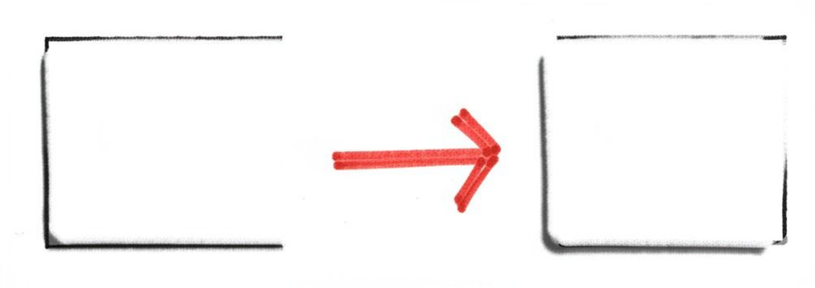


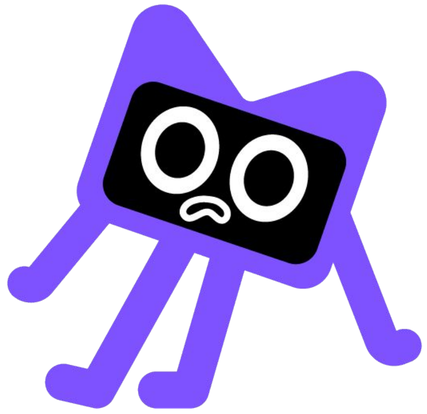
- Your *logical* state cannot be changed
- Also known as **abstract** state
- Informally, your observable state is fixed
  - Your concrete state may change, but it is impossible to observe from the outside
- Example: skip lists, splay trees

Cannot be changed means...

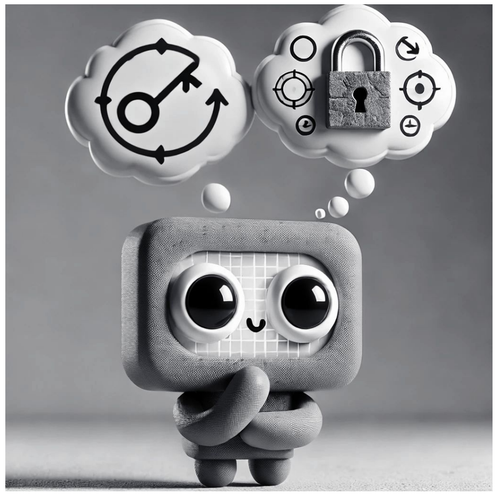


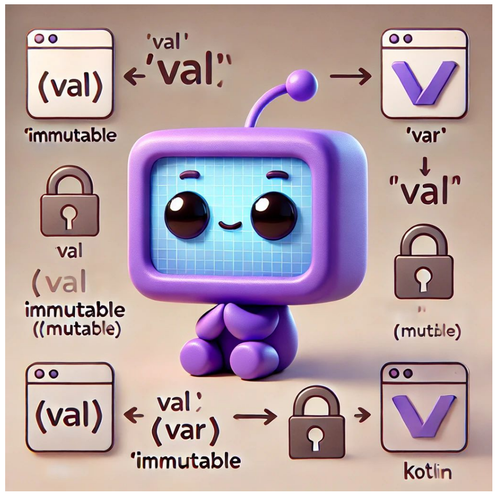




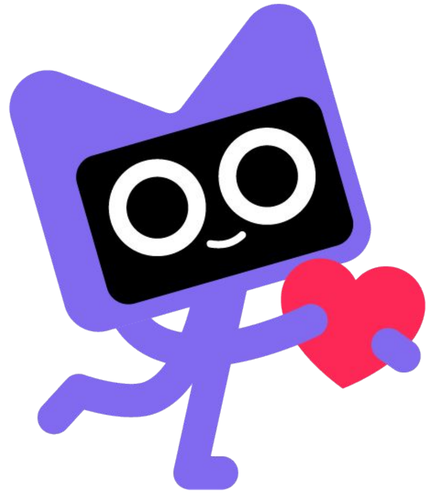


Immutability is when something cannot be changed





Where is immutability?

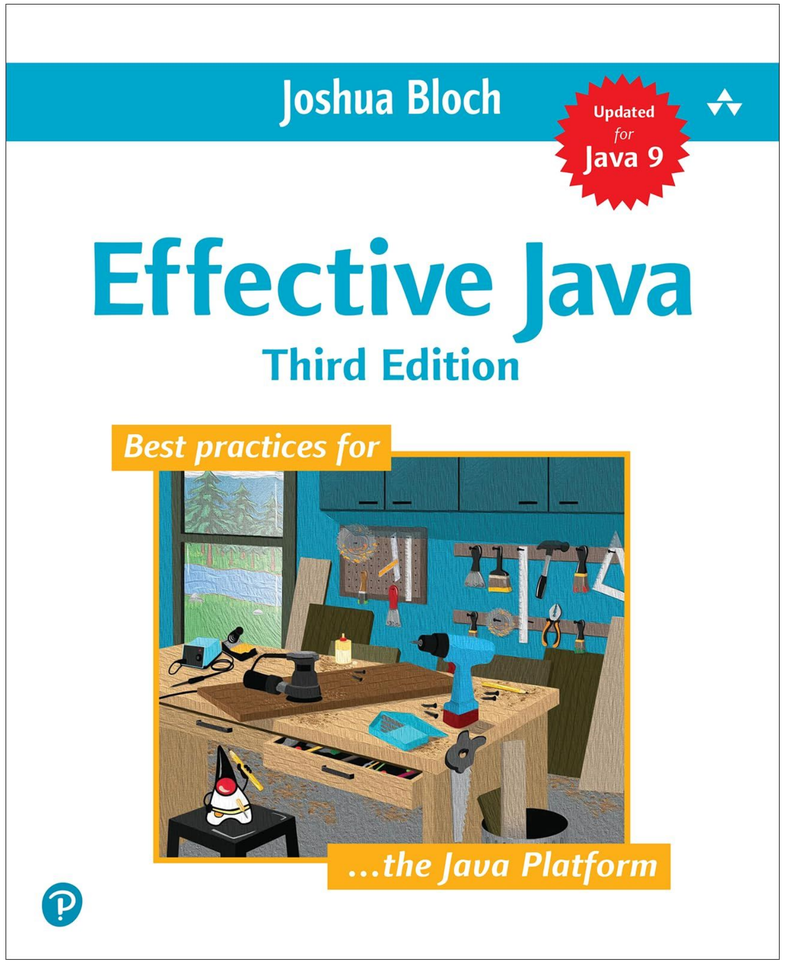


Kotlin supports simple pragmatic immutability

Effective Java: Item 17

Minimize mutability

- Don’t provide methods that modify the object’s state (known as mutators)
- Ensure that the class can’t be extended
- Make all fields final
- Make all fields private
- Ensure exclusive access to any mutable components



Don’t provide mutators

```
data class Point(val x: Int, val y: Int)
// No mutators are provided to change x or y
```

Prefer using immutable data to mutable. Always declare local variables and properties as `val` rather than `var` if they are not modified after initialization.

~~Make all fields private~~

```
class UserGroup(val members: List<User>) {
    //                ^
    //                this is not a field
    val admins: List<User> = members.filter { it.isAdmin() }
    //  ^
    //  this is not a field either
}
```

In Kotlin, a field is only used as a part of a property to hold its value in memory. Fields cannot be declared directly. However, when a property needs a backing field, Kotlin provides it

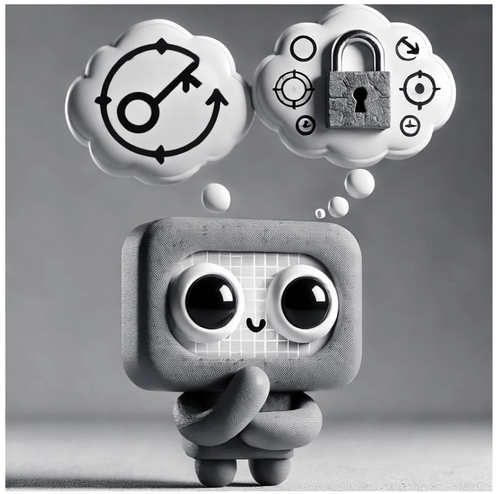
Ensure exclusive access to any mutable components

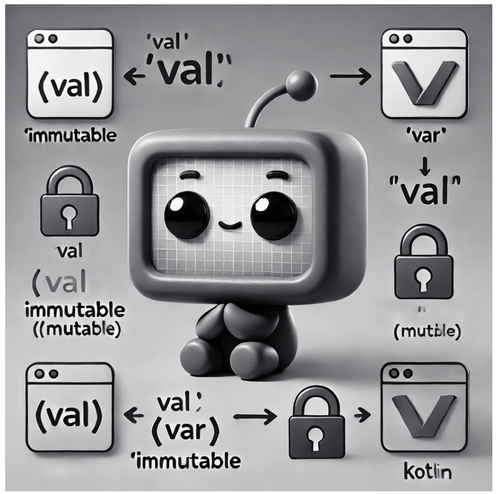
```
class UserCache {
    private val users: MutableMap<String, User> = mutableMapOf()
    val currentUsers: Collection<User>
        get() = users.values
                // mutable collection as read-only view
    val frozenCurrentUsers: Collection<User>
        get() = users.values.toList()
                // mutable collection defensively copied
}
```

Effective Java: Item 17

Minimize mutability

- ✅ Don’t provide methods that modify the object’s state (known as mutators)
- ✅ Ensure that the class can’t be extended
- ✅ Make all fields final
- ~~Make all fields private~~
- ☑️ Ensure exclusive access to any mutable components





Why is immutability?

Effective Java: Item 17

Immutable classes are easier to design, implement, and use than mutable

- Immutable objects are simple
- Immutable objects are inherently thread-safe
- Not only can you share immutable objects, but they can share their internals
- Immutable objects make great building blocks for other objects
- Immutable objects provide failure atomicity for free

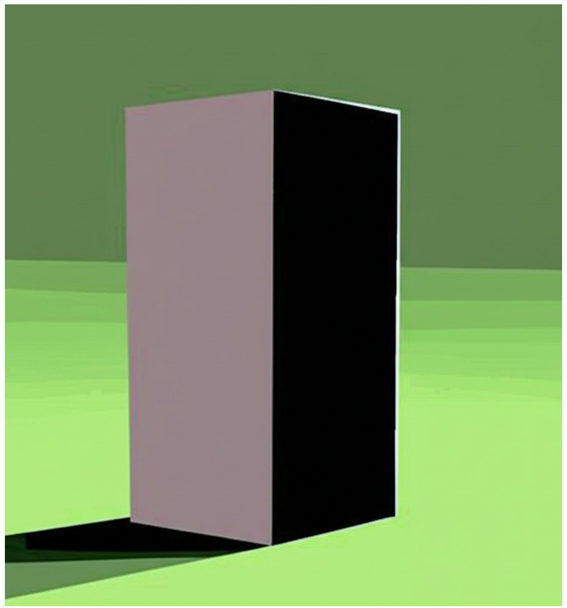
Is that all?

Why should we trust this?

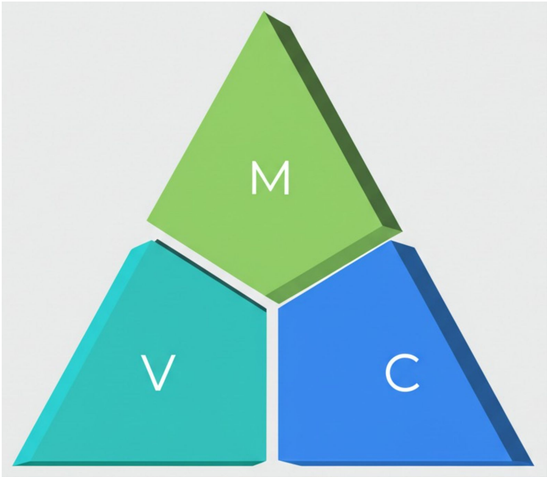
UI programming

Caching

UI programming



“Monolithic” UI

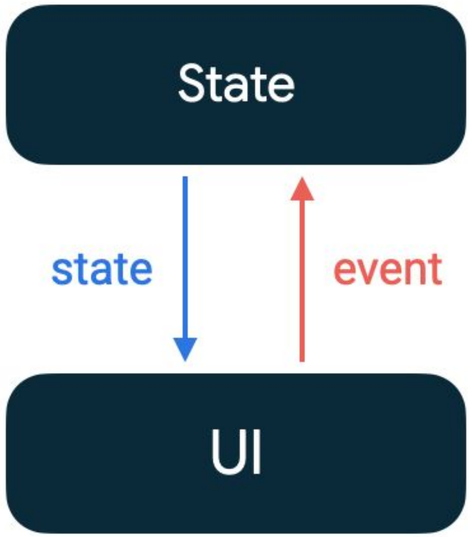


Model-View-Controller

Reactive UI

Jetpack Compose

- Declarative / reactive approach to UI
- Informally, “takes the C out of MVC”
  - Views are updated automatically when your **Models change**
- For some input (your app state), remembers the output (your rendered UI)
  - Skips recomposition if input did not change



- **This means you need to track the state changes**

Stability in Compose

```
data class Contact(val name: String, val address: Address)
data class Address(var street: String, var zipCode: String)

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

Stability in Compose

```
data class Contact(val name: String, val address: Address)
data class Address(var street: String, var zipCode: String)
```

Tracking changes in mutable data is hard

Compose compiler reports:

```
unstable class Contact {
 stable val name: String
 unstable val address: Address
 <runtime stability> = Unstable
}
```

```
unstable class Address {
 unstable val street: String
 unstable val zipCode: String
 <runtime stability> = Unstable
}
```

Stability in Compose

```
data class Contact(val name: String, val address: Address)
data class Address(val street: String, val zipCode: String)
```

Compose compiler reports:

```
  stable class Contact {
 stable val name: String
   stable val address: Address
 <runtime stability> =   Stable
}
```

```
  stable class Address {
   stable val street: String
   stable val zipCode: String
 <runtime stability> =   Stable
}
```

Stability in Compose

```
data class Contact(val name: String, val address: Address)
data class Address(val street: String, val zipCode: String)
```

Tracking changes in (deeply) immutable data is easy

Compose compiler reports:

```
stable class Contact {
  stable val name: String
  stable val address: Address
  <runtime stability> = Stable
}
```

```
stable class Address {
  stable val street: String
  stable val zipCode: String
  <runtime stability> = Stable
}
```

Best practices

Make the class immutable

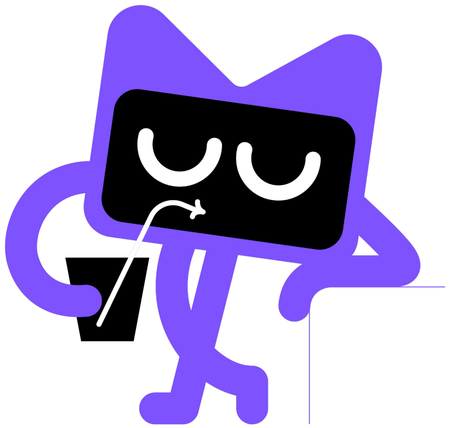
You can also try to make an unstable class completely immutable.

- **Immutable**: Indicates a type where the value of any properties can never change after an instance of that type is constructed, and all methods are referentially transparent.
    - Make sure all the class's properties are both `val` rather than `var`, and of immutable types.
    - Primitive types such as `String, Int`, and `Float` are always immutable.
    - If this is impossible, then you must use Compose state for any mutable properties.
- **Stable**: Indicates a type that is mutable. The Compose runtime does not become aware if and when any of the type's public properties or method behavior would yield different results from a previous invocation.

Caching

- Also known as memoization
- The most well-known way of “space-time tradeoff”
- For some input (your arguments), remembers the output (your computed result)
  - Avoids recomputation if input did not change

- **Duh, that’s the same as UI programming!**



Caching in practice

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

class LibraryRepository {
  val books: ImmutableList<Book>
    get() = apiCache.get(Path.BOOKS)
}
```

Caching in practice

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


class LibraryRepository {
  val books: ImmutableList<Book>
    get() = apiCache.get(Path.BOOKS)
}
```

```
fun egoisticClient() {
    val books = LibraryRepository().books

    // ...

    // "Fix" the book capitalization
    books.forEach {
        it.title = capitalize(it.title)
    }
    // Cache is now invalid!
}
```

Caching in practice

```
data class Book(val title: String, val isbn: ISBN)
```

```
class LibraryRepository {
  val books: ImmutableList<Book>
    get() = apiCache.get(Path.BOOKS)
}
```

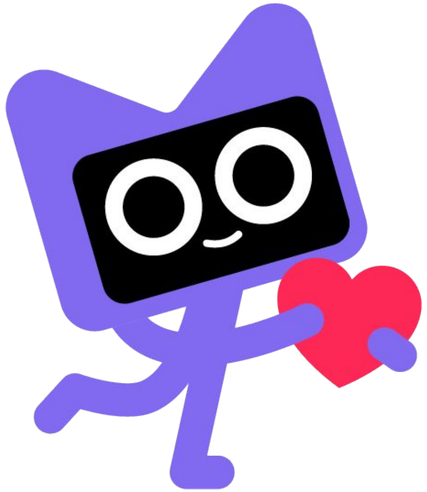
```
fun immutableClient() {
    val books = LibraryRepository().books

    // ...

    // Fix the book capitalization
    val fixedBooks = books.map {
        it.copy(title = capitalize(it.title))
    }
    // Cache is still valid!
}
```

Caching cares about outputs

- For some input (your arguments), remembers the output (your computed result)
  - Avoids recomputation if input did not change
- Also needs **outputs** to not change
  - Otherwise you can corrupt your cache

- **This means you need to return immutable data**

(better) immutability



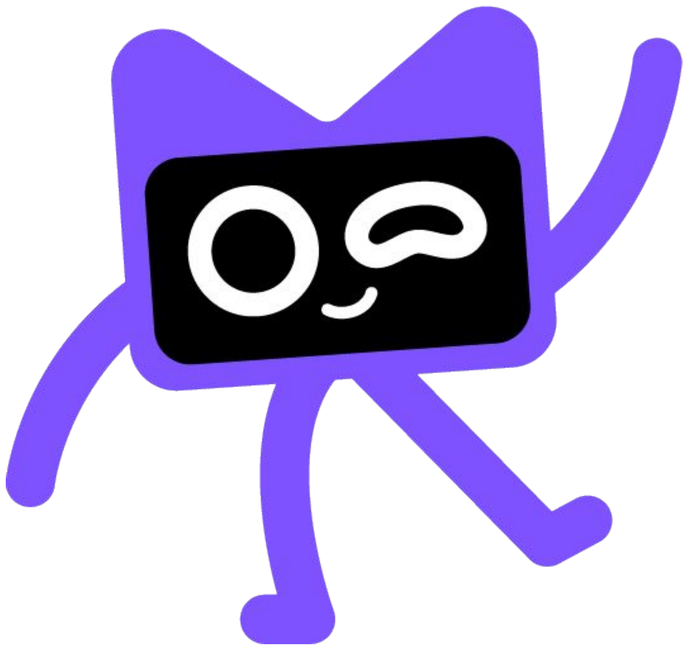
simple pragmatic immutability

(better) immutability

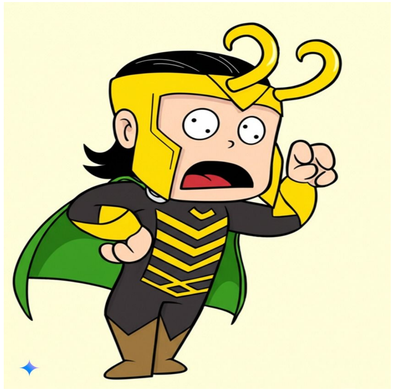
Thor: Valhalla (202X)

What is missing?

What is coming to Kotlin?



What about (project) Valhalla?



What is missing?

Immutable something

```
data class Contact(val name: String, val address: Address)
data class Address(var street: String, var zipCode: String)
```

- We have immutable references (`val`s), enforced by compiler
- We do not have immutable values, enforced by compiler
  - Neither immutable instances
  - Nor **immutable types**

Immutable types

```
data class Contact(val name: String, val address: Address)
data class Address(val street: String, val zipCode: String)
```

- What restrictions should immutable types have?
  - Shallow immutability
    - All stored (*concrete*) properties are `val`s
  - Deep immutability
    - All stored (*concrete*) properties are `val`s
    - All stored (*concrete*) properties are of deeply immutable types

Immutable types

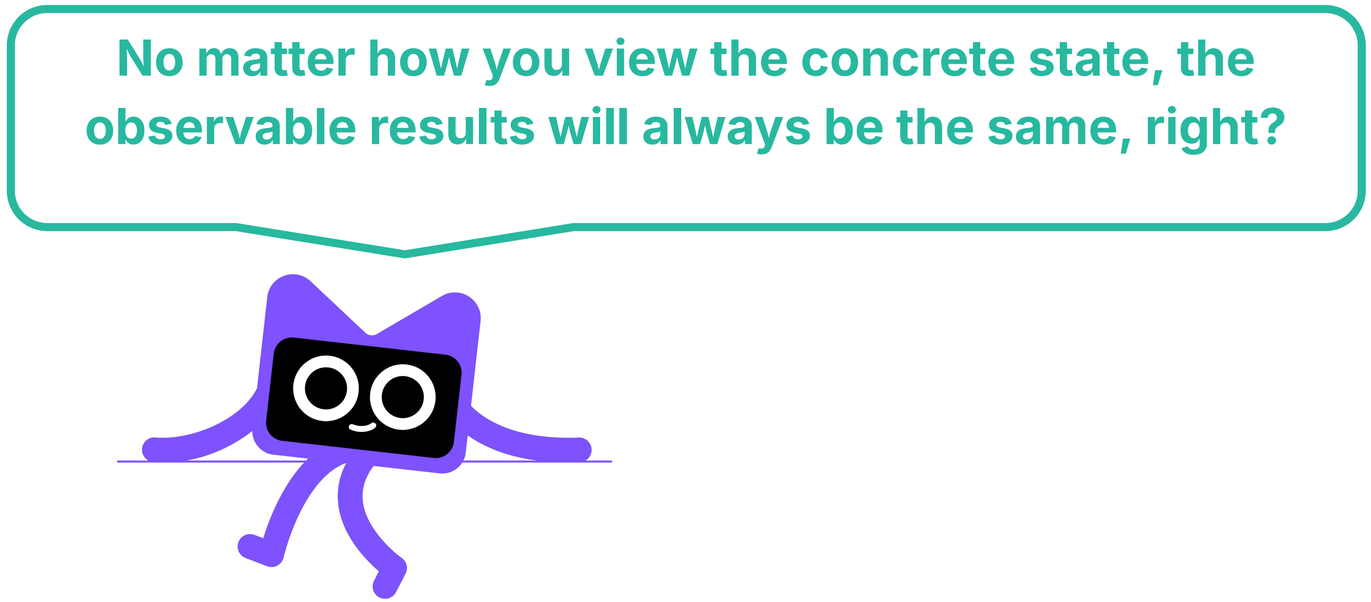
```
data class Contact(val name: String, val address: Address)
data class Address(val street: String, val zipCode: String)
```

- What restrictions should immutable types have?
  - Shallow immutability
    - All stored (*concrete*) properties are `val`s
  - Deep immutability
    - All stored (*concrete*) properties are `val`s
    - All stored (*concrete*) properties are of deeply immutable types

We can check concrete state automatically

Immutable types

```
data class Contact(val name: String, val address: Address)
data class Address(val street: String, val zipCode: String)
```



Concrete state vs identity

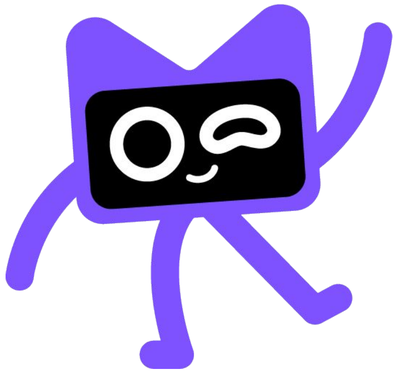
```
data class Contact(val name: String, val address: Address)
data class Address(val street: String, val zipCode: String)

val a = Address("Gelrestraat 16", "1079 MZ")
val b = Address("Gelrestraat 16", "1079 MZ")

println(a == b)  // Yay!
println(a === b) // Oops...
```

Identity “leaks” the mutability of the box

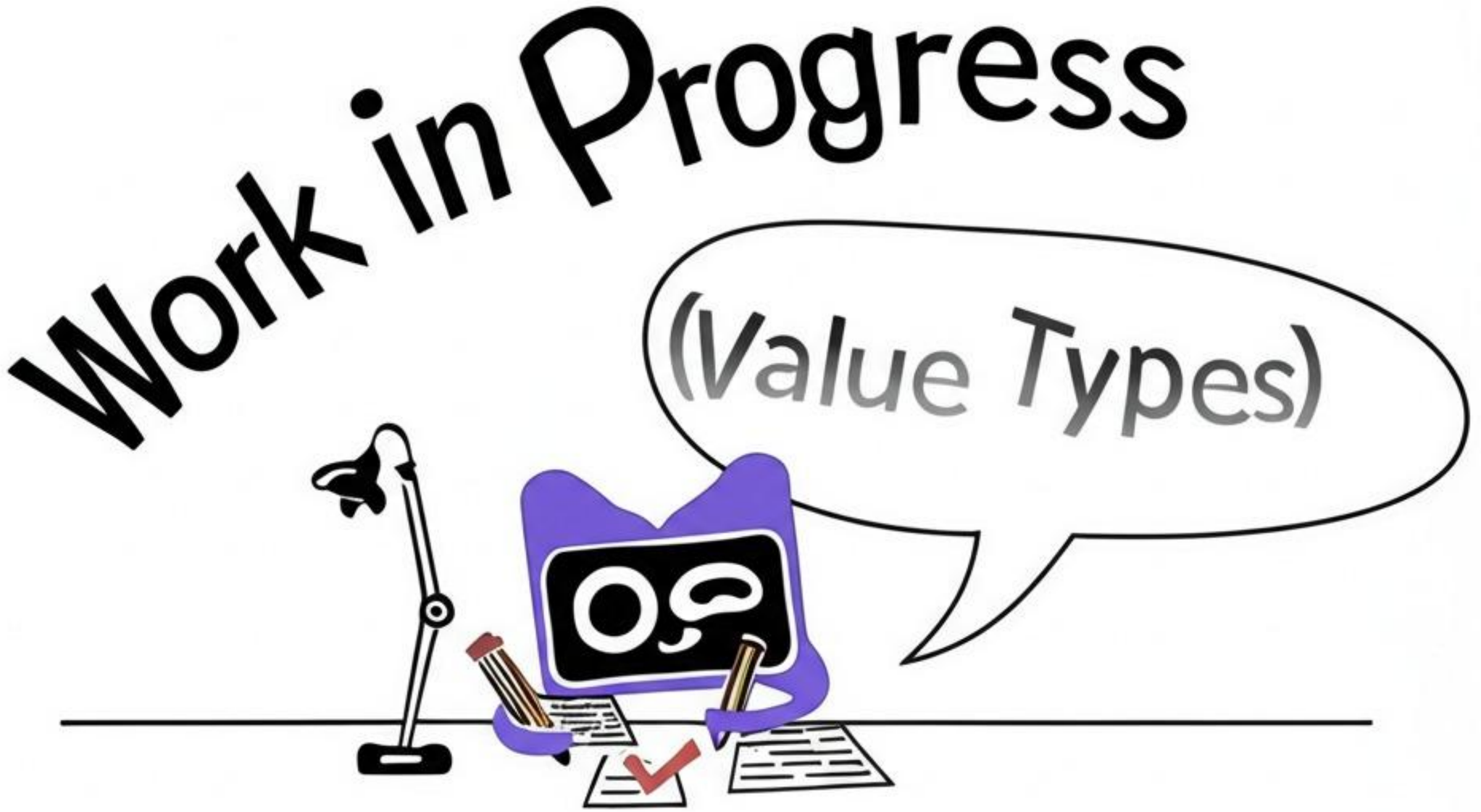
What is missing?

- The ability to declare (deeply) immutable types
  - Which are **checked** and **enforced** by the compiler
- These immutable types should not rely on identity
  - For immutable data, **only data matters**

- **We need (better) value types**



What is coming to Kotlin?


Work in Progress
(Value Types)

Inline value types

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

- Kotlin already has restricted value types (previously known as inline types)
  - No identity ✅
  - All stored (concrete) properties are `val`s ✅
  - Only one stored property allowed 😭
  - Only shallow immutability 😭
- They are also inlined when possible
  - But it is first and foremost an **optimization** 🗣️📢

Inline value types

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

- Kotlin already has restricted value types (previously known as inline types)
  - No identity ✅
  - All stored (concrete) properties are `val`s ✅
  - Only one stored property allowed 😭
  - Only shallow immutability 😭
- They are also inlined when possible
  - But it is first and foremost an **optimization** 🗣️📢

Immutable value types


```
immutable value class Contact(val name: String, val address: Address)
value class Book(val title: String, val isbn: ISBN)
```

- Kotlin is getting **(deeply) immutable** value classes **with multiple properties**
  - You can have more than one `val`s in your value classes
  - You can mark a value class `immutable` making it deeply immutable
  - Other restrictions are mostly the same as for inline value types
- They are **not** inlined
  - Inlining is first and foremost an **optimization**

Immutable value types


```
immutable value class Contact(val name: String, val address: Address)
value class Book(val title: String, val isbn: ISBN)
```

- Kotlin is getting **(deeply) immutable** value classes **with multiple properties**
  - You can have more than one `val`s in your value classes
  - You can mark a value class `immutable` making it deeply immutable
  - Other restrictions are mostly the same as for inline value types
- They are **not** inlined
  - Inlining is first and foremost an **optimization**

Deeply immutable value types

```
immutable value class ContactList(val contacts: List<Contact>) // Error
immutable value class Contact(val name: String, val address: Address) // Error
value class Address(val street: String, val zipCode: String)
```

- Checking for deep immutability requires more support
  - We know about (deep) immutability of value types
    - `value class` is shallow immutable, `immutable value class` is deeply immutable
  - What about regular (reference) types?
    - For example, `List`

Immutable reference types

```
immutable value class ContactList(val contacts: ImmutableList<Contact>)
```

- If you care about deeply immutable reference types, you can use some inherent knowledge
  - For example, knowledge about `kotlinx.collections.immutable`
- But you need to remember to use the correct immutable types

To resolve this, you can use immutable collections. The Compose compiler includes support for Kotlinx Immutable Collections. These collections are guaranteed to be immutable, and the Compose compiler treats them as such. This library is still in alpha, so expect possible changes to its API.

@Immutable reference types


```
@Immutable
public interface ImmutableList<out E> : List<E>, ImmutableCollection<E>
```

- Kotlin is getting `@Immutable` annotation to mark deeply immutable reference types
- The Kotlin compiler uses it as a contract
  - It considers such types as deeply immutable
  - It does not check whether this is actually true (**yet**)

Immutability and generics

```
@Immutable
public interface ImmutableList<out E> : List<E>, ImmutableCollection<E>
```

```
val contacts: ImmutableList<Contact> = fetchContacts() // OK
val immMatrix: ImmutableList<MutableList<Int>> = emptyMatrix() // Not OK
```

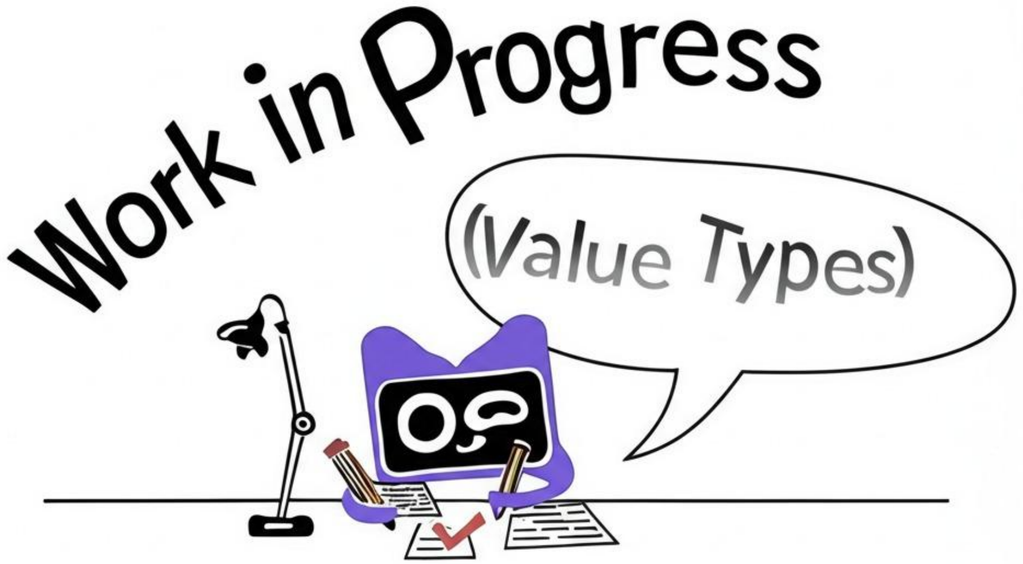
- Generic immutable types are *conditionally* immutable
  - Aka their immutability depends on how they are instantiated

immutable generics


```
@Immutable
public interface ImmutableList<out E> : List<E>, ImmutableCollection<E>
```

```
public fun <immutable E> immutableListOf(vararg elements: E): ImmutableList<E>
public fun <immutable T : Any?> mutableStateOf(value: T, …): MutableState<T>
```

- Kotlin begins tracking generics of deeply immutable types
  - Type parameters marked as `immutable` are checked to be deeply immutable on instantiation



(deeply) immutable value types with multiple properties

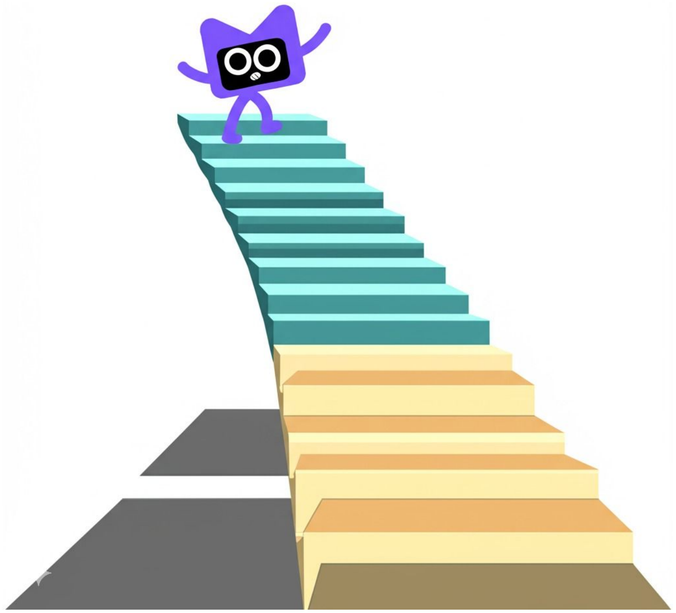
The old immutable world

```
data class Book(val title: String, val isbn: ISBN)


class LibraryRepository {

  val books: ImmutableList<Book>

    get() = apiCache.get(Path.BOOKS)
}
```

```
fun immutableClient() {
    val books = LibraryRepository().books

    // ...

    val fixedBooks = books.map {
        it.copy(title = capitalize(it.title))
    }

    // ...
}
```

The “copy” ladder

```
fun immutableClient() {
    …
    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(
                    …
                )
            )
        )
    }
    …
}
```



Maybe take the elevator instead?

```
fun immutableClient() {
    val books = LibraryRepository().books

    // ...

    val fixedBooks = books.map {
        it.title = capitalize(it.title)
        it.isbn.eanPrefix = 979
        it.publisher.address.zipCode = "..."
        it
    }

    // ...
}
```

Maybe take the elevator instead?

```
fun immutableClient() {
    val books = LibraryRepository().books

    // ...

    val fixedBooks = books.map {
        it.title = capitalize(it.title)          // Kotlin compiler could take
        it.isbn.eanPrefix = 979                  //   the copy ladder for us
        it.publisher.address.zipCode = "..."
        it
    }

    // ...
}
```

Mutable value semantics *Lite*

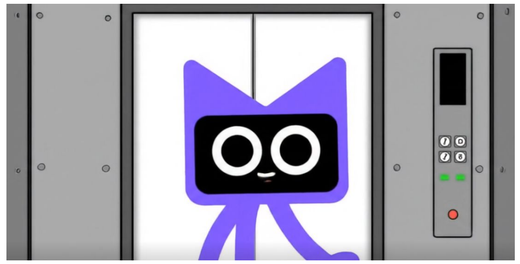
```
fun immutableClient() {
    val books = LibraryRepository().books

    // ...

    val fixedBooks = books.map { /* copy var fixedBook → */
        copy var fixedBook = it
        fixedBook.title = capitalize(fixedBook.title)
        fixedBook.isbn.eanPrefix = 979
        fixedBook.publisher.address.zipCode = "..."
        fixedBook
    }

    // ...
}
```

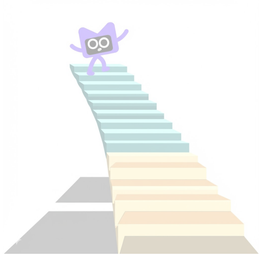
copy vars


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


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

copy vars

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

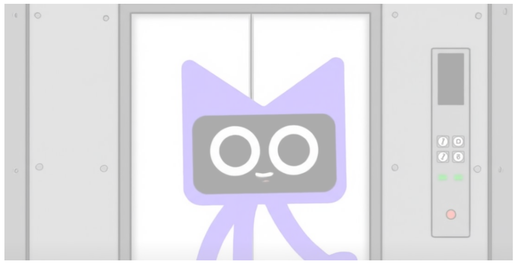


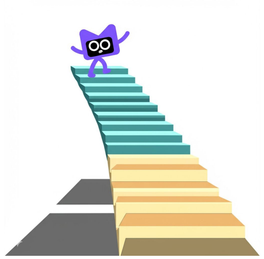


```
copy var book = books.first { ... }
book = book.withTitle(capitalize(book.title))
book = book.withIsbn(book.isbn.withEanPrefix(979))
```

copy vars

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

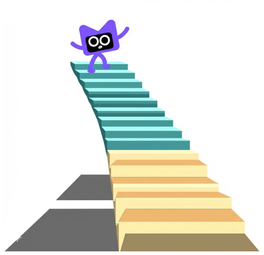




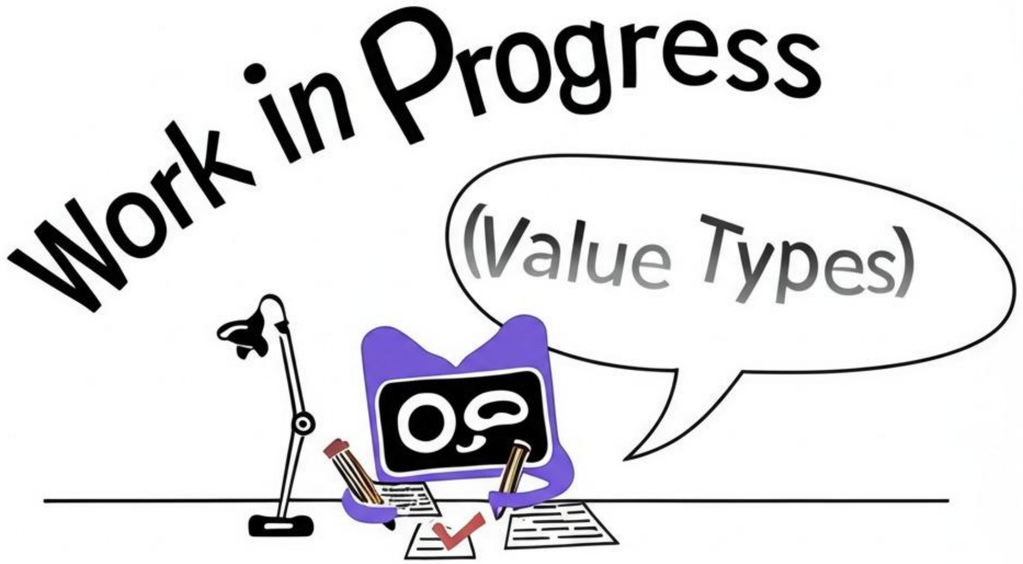
```
copy var book = books.first { ... }
book = book.withTitle(capitalize(book.title))
book = book.withIsbn(book.isbn.withEanPrefix(979))
```

copy vars

```
copy var book = books.first { ... }
book = book.with$Aggregated(
    title = capitalize(book.title),
    isbn = book.isbn.withEanPrefix(979)
)
```



- Kotlin compiler could optimize extra allocations for temporary objects
  - If we do not do this, we keep the same number of allocations as with manually controlled immutable data classes
  - If we do this, we get less allocations 💪



copy vars for ergonomic updates of immutable data

Multiple updates

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

Multiple updates as a fluent fun

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

```
fun Book.normalized(): Book {
    var self = this
    self.title = capitalize(self.title)
    self.isbn.eanPrefix = 979
    return self
}
```

Multiple updates as a fluent fun

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

```
fun Book.normalized(): Book {
    copy var self = this
    self.title = capitalize(self.title)
    self.isbn.eanPrefix = 979
    return self
}
```

Multiple updates as an “in-place” fun

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

```
fun Book.normalize(): ??? {
    // ???
}
```

copy funs


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

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

copy funs

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


// compiles down to fluent style


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

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


fun Book.normalize(): Book {
    // …
    return updatedBook
}
```

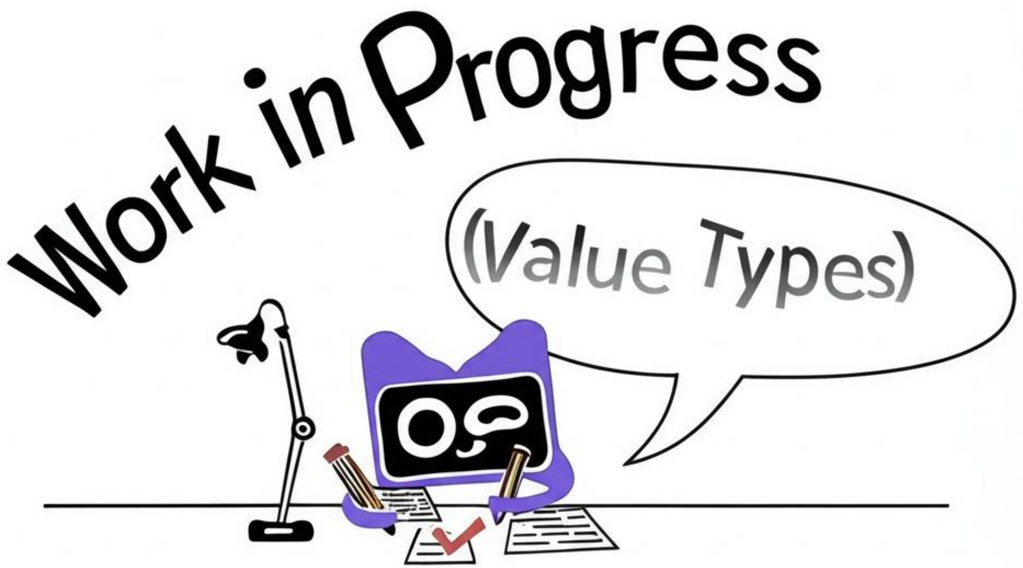
copy funs

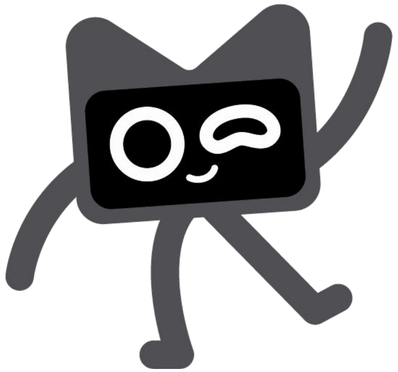
```
copy var book = books.first { ... }
book.normalize()


// compiles down to fluent style

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

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

fun Book.normalize(): Book {
    // …
    return self
}
```

What else is still being designed?

- Do we need `value interfaces`?
- How value types are captured by lambdas?
- Are immutable value arrays a thing or not?
- Do we need `copy funs` with custom return types?
- Do we need `copy var` function parameters?
- Do we need `copy` functional types?
- ...





What about project Valhalla?

JEP 401

- Value classes on the JVM platform
- Two main goals
  - Provide types which opt-out of identity
  - Support run-time optimizations of these types

JEP 401 value classes

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

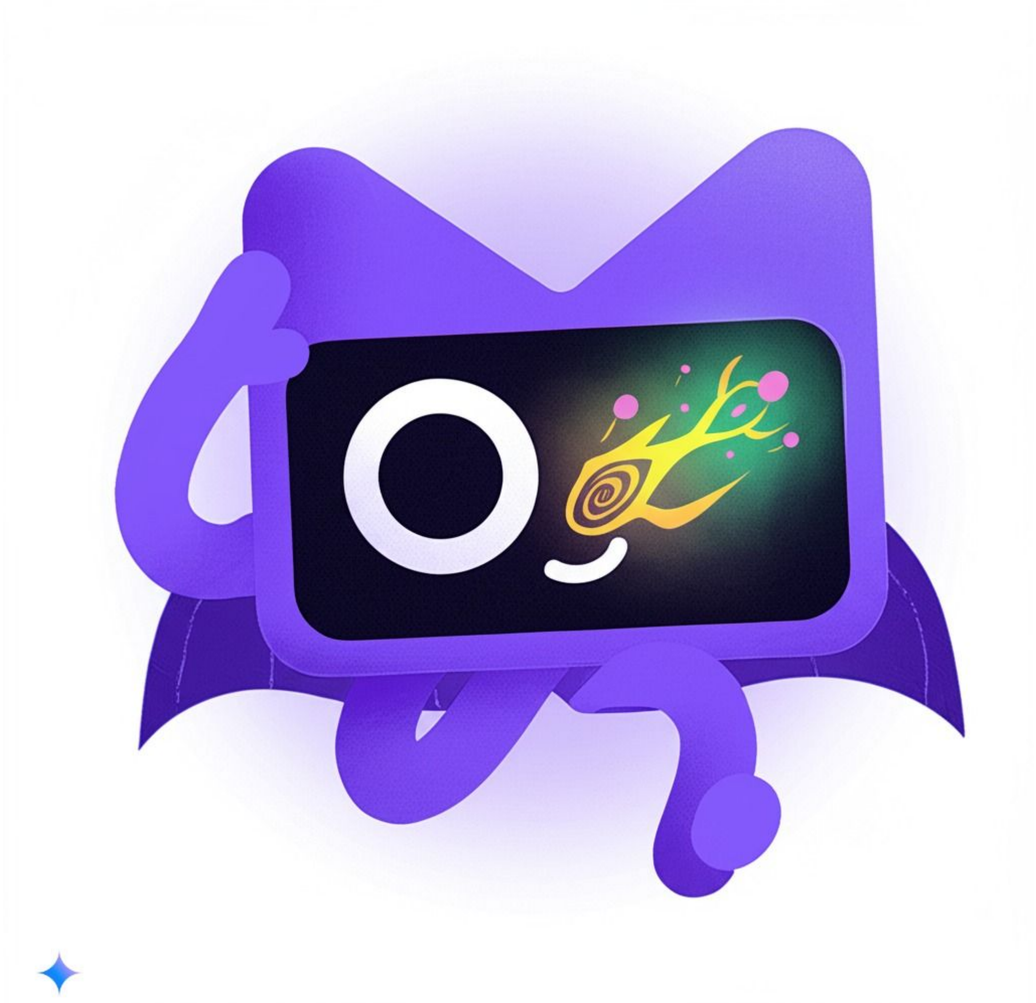
- JEP 401 value class == shallow immutable type
- This is the basic building block for immutability in Kotlin, to which we add
  - Deep immutability
  - Ergonomic updates of immutable data
  - Better support across other language features

It’s about immutability

- We care about immutability of data, that’s our **main goal**
- If we can also get run-time optimizations for immutable data, that’s a **bonus**
  - When Valhalla comes, we get this bonus **for free**

- Once we get the user experience with immutability right, we can think about adding optimizations
- If you care about optimizations first, please reach out!

The Infinity Saga (20YY)



Immutability in Kotlin specification

- How many times is **immutability** mentioned in the Kotlin language specification?
  - never
  - 10 times
  - more than 25 times

Immutability in Kotlin specification

- How many times is immutability mentioned in the Kotlin language specification?
  - never
  - **10 times**
  - more than 25 times

8 of 10 times immutability is mentioned with **smart casts**

Immutability and smart casts

```
data class Pair<out A, out B>(val first: A, val second: B)

fun prettyPrint(person: Pair<String?, String?>) = buildString {
    if (person.first ≠ null ) append(person.first) // No smart cast
    // ...
}
```

- `Pair` comes from another module, we do not support smart casts for such types

Immutability and smart casts

```
value class Pair<out A, out B>(val first: A, val second: B)

fun prettyPrint(person: Pair<String?, String?>) = buildString {
    if (person.first ≠= null ) append(person.first) // Yes smart cast
    // ...
}
```

- Value types cannot be changed concurrently
  - It is safe to propagate the smart cast knowledge in more cases

(Im)mutability and concurrency

- Immutable objects are simple
- Immutable objects are inherently thread-safe
- Not only can you share immutable objects, but they can share their internals
- Immutable objects make great building blocks for other objects
- Immutable objects provide failure atomicity for free

Mutability and concurrency

```
data class State(var counter: Int = 0)

fun main() {
    val state = State()
    runBlocking(Default) {
        launch { repeat(1000) { state.counter++ } }
        launch { repeat(1000) { state.counter++ } }
    }
    println(state.counter) // ????
}
```

Mutability and concurrency

```
data class State(var counter: Int = 0)


fun main() {
    val state = State()
    runBlocking(Default) {
        launch { repeat(1000) { state.counter++ } } // Data…
        launch { repeat(1000) { state.counter++ } } // …race!
    }
    println(state.counter) // 1986
}
```

Value types and concurrency

```
immutable value class State(copy var counter: Int = 0)


fun main() {
    copy var state = State()
    runBlocking(Default) {
        launch { repeat(1000) { state.counter++ } }
        launch { repeat(1000) { state.counter++ } }
    }
    println(state.counter) // ????
}
```

Value types and concurrency

```
value class State(copy var counter: Int = 0)

fun main() {
    copy var state = State()
    runBlocking(Default) {
        launch { repeat(1000) { state.counter++ } }
        launch { repeat(1000) { state.counter++ } } // Cannot capture copy var
                                                    // with a possible race
    }
    println(state.counter) // ????
}
```

Immutability and concurrency

- It is *always* safe to share `val` reference to immutable data
- It is *often* safe to share (`copy`) `var` reference to immutable data
  - If you can prove it is not accessed concurrently
- It is *sometimes* safe to share any reference to mutable data
  - If mutable data is designed to be concurrent-safe

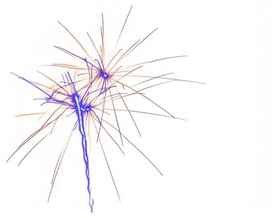
- **The exact goals and rules of these are to be designed…**

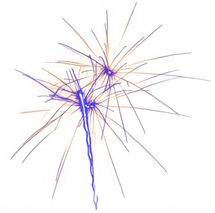
Recap

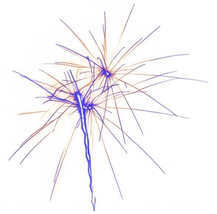
(deeply) immutable value types
with multiple properties

copy vars / funs for ergonomic
updates of immutable data

better smart casts
safer concurrency







something else?

@tau_phoenix

KT-77734