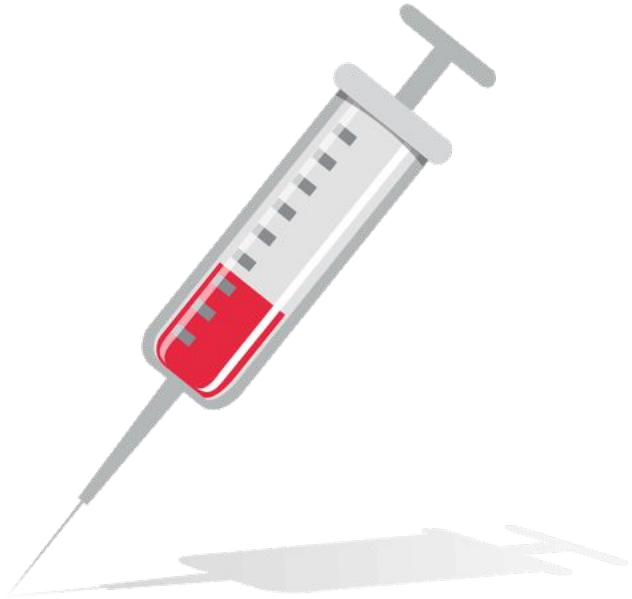
Kotlin not-to-do list

WHAT YOU SHOULD AVOID DOING IN KOTLIN

@MarcinMoskala



Part 1: Good code

Chapter 1: Safety

Item 1: Limit mutability
Item 2: Minimize the scope of variables
Item 3: Eliminate platform types as soon as possible
Item 4: Do not expose inferred types
Item 5: Specify your expectations on arguments and state
Item 6: Prefer standard errors to custom ones
Item 7: Prefer `null` or `Failure` result when the lack of result is possible
Item 8: Handle nulls properly
Item 9: Close resources with use
Item 10: Write unit tests

Chapter 2: Readability

Item 11: Design for readability
Item 12: Operator meaning should be consistent with its function name
Item 13: Avoid returning or operating on `Unit?`
Item 14: Specify the variable type when it is not clear
Item 15: Consider referencing receiver explicitly
Item 16: Properties should represent state, not behavior
Item 17: Consider naming arguments
Item 18: Respect coding conventions

Part 2: Code design

Chapter 3: Reusability

Item 19: Do not repeat knowledge
Item 20: Do not repeat common algorithms
...


YOU SHALL NOT PASS!

Hiding too much

Item 14: Specify the variable type when it is not clear

```
val num = 10

val name = "Marcin"

val ids = listOf(12, 112, 554, 997)

val data = getData()
```

Item 14: Specify the variable type when it is not clear

```
val num = 10

val name = "Marcin"

val ids = listOf(12, 112, 554, 997)

val data: UserData = getData()
```

Type inference is exact

```
open class Animal
class Zebra: Animal()

fun main() {
    var animal: Animal = Zebra()
    animal = Animal()
}
```

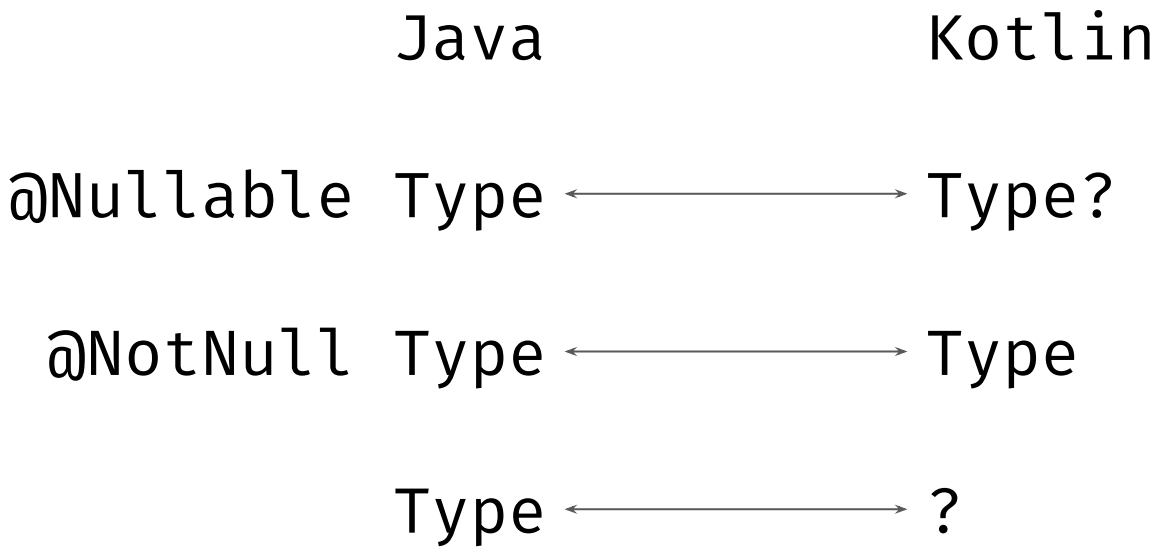
Item 4: Do not expose inferred types

```
interface CarFactory {
    fun produce() = DEFAULT_CAR
}

val DEFAULT_CAR = Fiat126P()
```

Fiat126P

Null safety

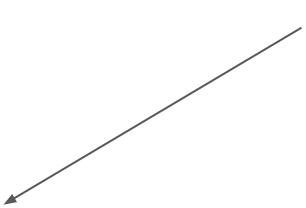
Java types and nullability



Some of nullability annotations supported by Kotlin

- JetBrains (`@Nullable` and `@NotNull` from `org.jetbrains.annotations`)
- Android (`@Nullable` and `@NonNull` from `androidx.annotation`, `com.android.annotations` and `android.support.annotations`)
- JSR-305 (`@Nullable`, `@CheckForNull` and `@Nonnull` from `javax.annotation`)
- JavaX (`@Nullable`, `@CheckForNull`, `@Nonnull` from `javax.annotation`)
- FindBugs (`@Nullable`, `@CheckForNull`, `@PossiblyNull` and `@NonNull` from `edu.umd.cs.findbugs.annotations`)
- ReactiveX (`@Nullable` and `@NonNull` from `io.reactivex.annotations`)
- Eclipse (`@Nullable` and `@NonNull` from `org.eclipse.jdt.annotation`)
- Lombok (`@NonNull` from `lombok`)

```java
// Java
public class UserRepo {
    public User getUser() {
        //...
    }
}
```




```kotlin
// Kotlin
val repo = UserRepo()
val user1 = repo.user          // Type of user1 is User!
val user2: User = repo.user    // Type of user2 is User
val user3: User? = repo.user // Type of user3 is User?
```

Item 3: Eliminate platform types as soon as possible

```kotlin
// Kotlin
fun statedType() {
    val value: String = JavaClass().value // NPE
    //...
    println(value.length)
}

fun platformType() {
    val value = JavaClass().value
    //...
    println(value.length) // NPE
}
```

String!

```
interface UserRepo {
    fun getUserName() = JavaClass().value
}
```

Declaration has type inferred from a platform call, which can lead to unchecked nullability issues. Specify type explicitly as nullable or non-nullable. more... (⌘F1)

```
class RepoImpl: UserRepo {
    override fun getUserName(): String? {
        return null
    }
}

fun main() {
    val repo: UserRepo = RepoImpl()
    val text: String = repo.getUserName()
    print("User name length is ${text.length}")
}
```

Runtime NPE

Receiver hiding

```
sourceList += SourceEntity().apply {
    id = it.id
    category = it.category
    country = it.country
    description = it.description
}
```

Item 15: Consider referencing receiver explicitly

```
fun <T : Comparable<T>> List<T>.quickSort(): List<T> {
    if (size < 2) return this
    val pivot = this.first()
    val (smaller, bigger) = this.drop(1)
            .partition { it < pivot }
    return smaller.quickSort() + pivot + bigger.quickSort()
}
```

Item 15: Consider referencing receiver explicitly

```
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .apply { print("Created $name") }

    fun create(name: String): Node? = Node(name)
}

val node = Node("parent")
node.makeChild("child") // Prints: Created parent
```

Item 15: Consider referencing receiver explicitly

```
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .apply { print("Created ${this.name}") }

    fun create(name: String): Node? = Node(name)
}

val node = Node("parent")
node.makeChild("child") // Prints nothing
```

Error: Receiver is nullable

Item 15: Consider referencing receiver explicitly

```
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .apply { print("Created ${this?.name}") }

    fun create(name: String): Node? = Node(name)
}

val node = Node("parent")
node.makeChild("child") // Prints: Created parent.child
```

Item 15: Consider referencing receiver explicitly

```
class Node(val name: String) {

    fun makeChild(childName: String) =
        create("$name.$childName")
            .also { print("Created ${it?.name}") }

    fun create(name: String): Node? = Node(name)
}

val node = Node("parent")
node.makeChild("child") // Prints: Created parent.child
```

Item 41: Avoid member extensions

```
class PhoneBook {
    // ...

    fun String.isPhoneNumber() =
      length == 7 && all { it.isDigit() }
}


PhoneBook().apply { "1234567890".isPhoneNumber() }
```

Item 41: Avoid member extensions

```
class PhoneBook {
    // ...
}

private fun String.isPhoneNumber() =
     length == 7 && all { it.isDigit() }


val ref = String::isPhoneNumber
val str = "1234567890"
val boundedRef = str::isPhoneNumber
```

Item 41: Avoid member extensions

```
class A {
    val a = 10
}
class B {
    val a = 20
    val b = 30

    fun A.test() = a + b
}
```

Where is it from?

```
class A {
    //...
}
class B {
    //...

    fun A.update() = ...
}
```

What shell we update?
A or B?

Choosing short over readable

Readability vs conciseness

APL:

```
life←{↑1 ωV.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

J:

```
life=:[:+/(3 4=/[:+/(,/,"0/~i:1)|.])*.1,:]
```

Item 11: Design for readability

```
if (person != null && person.isAdult) {
    view.showPerson(person)
} else {
    view.showError()
}
```

```
person?.takeIf { it.isAdult }
    ?.let(view::showPerson)
    ?: view.showError()
```

```
if (person != null && person.isAdult) {
    view.showPerson(person)
    view.hideProgressWithSuccess()
} else {
    view.showError()
    view.hideProgress()
}

person?.takeIf { it.isAdult }
    ?.let {
        view.showPerson(it)
        view.hideProgressWithSuccess()
    } ?: run {
        view.showError()
        view.hideProgress()
    }
```

Item 11: Design for readability

```
if (person != null && person.isAdult) {
    view.showPerson(person)
} else {
    view.showError()
}
```

```
person?.takeIf { it.isAdult }
    ?.let(view::showPerson)
    ?: view.showError()
```

Show when person is `null`, not adult, or `showPerson` returns `null`.

Item 13: Avoid returning or operating on `Unit?`

Why would we return `Unit?`?

To use nullability support for other purposes.

```
fun api(): Any? {
    if (!correctHash()) return "Incorrect hash"
    //...
}

fun correctHash(): Boolean = call.hasCorrectHash
```

Item 12: Operator meaning should be consistent with its function name

```
fun Int.factorial(): Long = (1..this).product()

print(10 * 6.not()) // 7200

print(10 * !6) // 7200

operator fun Int.not() = factorial()
```

Item 12: Operator meaning should be consistent with its function name

Operator	Corresponding Function
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()
++a	a.inc()
--a	a.dec()
a+b	a.plus(b)
a-b	a.minus(b)
a*b	a.times(b)
a/b	a.div(b)
a..b	a.rangeTo(b)
a in b	a.contains(b)

```
x + y

=

x.plus(y)
```

Item 12: Operator meaning should be consistent with its function name

```
val a = "1, 2, 3, 4"
print(a.removeAll(", ")) // 1234
```

Unclear cases

```
operator fun Int.times(operation: () -> Unit): ()->Unit = {
    repeat(this) { operation() }
}

val tripledHello = 3 * { print("Hello") }

tripledHello() // Prints: HelloHelloHello
```

Unclear cases

```
operator fun Int.times(operation: () -> Unit): ()->Unit {
    repeat(this) { operation() }
}

3 * { print("Hello") } // Prints: HelloHelloHello
```

Unclear cases

```
infix fun Int.timesRepeated(operation: ()->Unit) = {
    repeat(this) { operation() }
}

val tripledHello = 3 timesRepeated { print("Hello") }
tripledHello() // Prints: HelloHelloHello


repeat(3) { print("Hello") } // Prints: HelloHelloHello
```

Disrespecting contracts

Item 27: Minimize elements visibility

- A class cannot be responsible for its own state when properties that represent this state can be changed from the outside.
- It is easier to track how class changes when they have more restricted visibility.

```
class UserRepository {
    private var prevUser: User? = null

    fun getUser(): User {
        // ...
    }

    private fun connectWithDb(): Connection {
        // ...
    }
}
```

- It is easier to learn and maintain a smaller interface.
- When we want to make changes, it is way easier to expose something new, rather than to hide an existing element.

Item 29: Respect abstraction contract

```
class Employee {
    private fun privateFunction() {
        println("You won't get me!")
    }
}


Employee::class
    .declaredMemberFunctions
    .first { it.name == "privateFunction" }
    .apply { isAccessible = true }
    .call(Employee()) // Prints: You won't get me!
```

Item 29: Respect abstraction contract

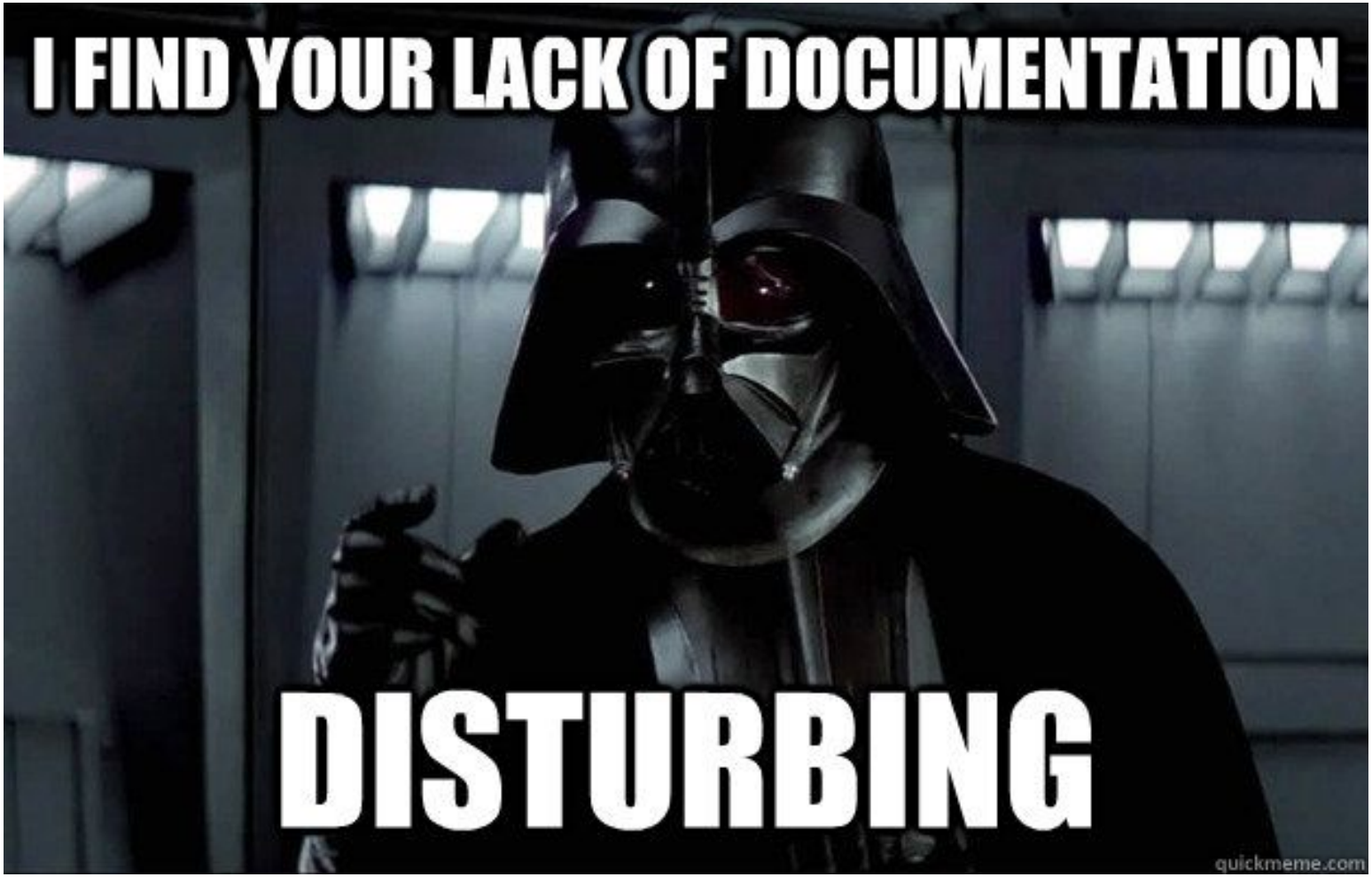
```
inline fun <reified T> T.callMethod(name: String): Any? =
    T::class
        .declaredMemberFunctions
        .first { it.name == name }
        .apply { isAccessible = true }
        .call(this)
```

```
Employee().callMethod("privateFunction")
// You won't get me!
```

```
class UserRepository {
    private var prevUser: User? = null

    fun getUser(): User {
        // ...
    }

    private fun connectWithDb(): Connection {
        // ...
    }
}
```

Item 28: Define contract with documentation

```
abstract class LoggedView {

    /**
    *  Method defining how logout dialog should look like.
    *  Used internally when session expires.
    *  Should not be used directly.
    */
    protected fun showLogoutDialog() {
        //...
    }
}
```


I FIND YOUR LACK OF DOCUMENTATION
DISTURBING
quickmeme.com

Item 29: Respect abstraction contract

```kotlin
/**
* Returns a new read-only list of given elements.
* The returned list is serializable (JVM).
*/
public fun <T> listOf(vararg elements: T): List<T> = ...
```

```kotlin
val list = listOf(1,2,3)

if (list is MutableList) {
    list.add(4)
}
```

```kotlin
val list = listOf(1,2,3)
val mutable = list.toMutableList()
mutable.add(4)
```

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(AbstractList.java:148)
    at java.util.AbstractList.add(AbstractList.java:108)
    at Simplest_versionKt.main(Simplest version.kt:6)
```

Item 37: Respect the contract of `equals`

```
/**
 * Indicates whether some other object is "equal to" this one. Implementations must fulfil the f
 * requirements:
 *
 * * Reflexive: for any non-null value `x`, `x.equals(x)` should return true.
 * * Symmetric: for any non-null values `x` and `y`, `x.equals(y)` should return true if and onl
 * * Transitive:  for any non-null values `x`, `y`, and `z`, if `x.equals(y)` returns true and `
 * * Consistent:  for any non-null values `x` and `y`, multiple invocations of `x.equals(y)` con
 * * Never equal to null: for any non-null value `x`, `x.equals(null)` should return false.
 *
 * Read more about [equality](https://kotlinlang.org/docs/reference/equality.html) in Kotlin.
 */
public open operator fun equals(other: Any?): Boolean
```

Item 37: Respect the contract of equals

```
class Time(
    val millisArg: Long = -1,
    val isNow: Boolean = false
) {
    val millis: Long
        get() = if (isNow) System.currentTimeMillis()  else millisArg

    override fun equals(other: Any?): Boolean =
        other is Time && millis == other.millis
}

val now = Time(isNow = true)
now == now // Sometimes true, sometimes false
List(100000) { now }.all { it == now } // Most likely false
```

Not reflexive
Not consistent

You cannot trust:

- `contains`
- `Set`
- `assertEquals`
- ...

Item 36: Prefer class hierarchies to tagged classes

```
sealed class Time
data class TimePoint(val millis: Long): Time()
object Now: Time()
```

Item 37: Respect the contract of `equals`

```
class Complex(val real: Double, val img: Double) {
    override fun equals(other: Any?): Boolean =
        if (other is Double) img == 0.0 && real == other
        else other is Complex && real == other.real && img == other.img
}
```

Not symmetric

```
Complex(1.0, 0.0).equals(1.0) // true
1.0.equals(Complex(1.0, 0.0)) // false
```

```
val list = listOf<Any>(Complex(1.0, 0.0))
list.contains(1.0) // ?
```

You cannot trust:

- `contains`
- `Set`
- `assertEquals`
- ...

Item 38: Respect the contract of `hashCode`

```
31    class FullName(
      Class has 'equals()' defined but does not define 'hashCode()' more... (⌘F1)
33        var surname: String
34    ) {
35        override fun equals(other: Any?): Boolean = other is FullName &&
36                other.name == name && other.surname == surname
37    }
```

```
val name1 = FullName("Marcin", "Moskała")
val name2 = FullName("Marcin", "Moskała")
print(name1 == name2) // true


val set = mutableSetOf<FullName>()
set.add(name1)
print(set.contains(name2)) // false
```

You cannot trust:

- `Set`
- `Map`
- ...

Item 38: Respect the contract of `hashCode`

```
class NameOk(val name: String) {
    override fun equals(o: Any?): Boolean = o is NameOk && o.name == name
    override fun hashCode(): Int = name.hashCode()
}
class NameNope(val name: String) {
    override fun equals(o: Any?): Boolean = o is NameNope && o.name == name
    override fun hashCode(): Int = 0
}


val okSet = List(10000) { NameOk("$it") }.toSet() // Equals used 0 times
val nopeSet = List(10000) { NameNope("$it") }.toSet() // Equals used 50116683 times

NameOk("9999") in properSet // Equals used 1 times
NameOk("Not there") in properSet // Equals used 0 times
NameNope("9999") in nopeSet // Equals used 4324 times
NameNope("Not there") in nopeSet // Equals used 10001 times
```

Item 16: Properties should represent state, not behavior

```
val Tree<Int>.sum: Int
    get() = when (this) {
        is Leaf -> value
        is Node -> left.sum + right.sum
    }
```

Item 16: Properties should represent state, not behavior

```
fun Tree<Int>.sum(): Int = when (this) {
    is Leaf -> value
    is Node -> left.sum() + right.sum()
}
```

Item 16: Properties should represent state, not behavior

```
class UserIncorrect {
    private var name: String = ""
    fun getName() = name
    fun setName(name: String) {
        this.name = name
    }
}
```

Item 35: Use function types instead of interfaces to pass operations and actions

```
class CalendarView {
    var onDateClicked: ((date: Date) -> Unit)? = null
    var onPageChanged: ((date: Date) -> Unit)? = null
}

calendarView.onDateClicked = { /*...*/ }
calendarView.onPageChanged = fun(date) { /*...*/ }
calendarView.onDateClicked = DateHandler::onDateChanged
calendarView.onPageChanged = handler::onPageChanged

class OnDateClicked: (Date) -> Unit { /*...*/ }
calendarView.onDateClicked = OnDateClicked()
```

Biggest sins of Kotlin developers:

- Hiding too much
- Choosing short over readable
- Disrespecting contracts

Hiding too much

Item 14: Specify the variable type when it is not clear
Item 4: Do not expose inferred types
Item 3: Eliminate platform types as soon as possible
Item 15: Consider referencing receiver explicitly
Item 41: Avoid member extensions

Choosing short over readable

Item 11: Design for readability
Item 13: Avoid returning or operating on Unit?
Item 12: Operator meaning should be consistent with its function name

Choosing short over readable

Item 27: Minimize elements visibility
Item 29: Respect abstraction contract
Item 28: Define contract with documentation
Item 37: Respect the contract of equals
Item 38: Respect the contract of hashCode
Item 16: Properties should represent state, not behavior
Item 35: Use function types instead of interfaces to pass operations and actions

Kotlin not-to-do list

WHAT YOU SHOULD AVOID DOING IN KOTLIN

@MarcinMoskala