

Kotlin Part 4: Idiomatic Kotlin

CS 346 Application
Development

Credits

- This section summarizes a talk by Urs Peters @Kotlin Dev Day 2022: [Idiomatic Kotlin: the key to unlocking Kotlin's true potential](#).

Why Idiomatic Kotlin?

It's possible to use Kotlin as a “better Java”, but you would be missing out on some of the features that make Kotlin unique and interesting.

- It has a number of unique language features that are worth learning!
- These can lead you to more efficient and effective use of the language and its libraries.

We'll discuss Kotlin design principles along the way.

1. Favour immutability over mutability

Kotlin favors immutability with various immutable constructs and defaults.

What is so good about immutability?

- **Immutability**: exactly one state that will never change.
- **Mutable**: an infinite amount of potential states.

```
data class Programmer(val name: String,  
                      val languages: List<String>)  
fun known(language:String) = languages.contains(language)  
  
val urs = Programmer("Urs", listOf("Kotlin", "Scala", "Java"))  
val joe = urs.copy(name = "Joe")
```

Criteria	Immutable	Mutable
Reasoning	Simple: one state only	Hard: many possible states
Safety	Safer: state remains the same and valid	Unsafe: accidental errors due to state changes
Testability	No side effects which makes tests deterministic	Side effects: can lead to unexpected failures
Thread-safety	Inherently thread-safe	Manual synchronization required

How to leverage it?

- prefer vals over vars
- prefer read-only collections (listOf instead of mutableListOf)
- use immutable value objects instead of mutable ones (e.g. data classes over classes)

Local mutability that does not leak outside is ok (e.g. a var within a function is ok if nothing external to the function relies on it).

2. Use nullability appropriately

Think twice before using !!

```
val uri = URI("...")
val res = loadResource(uri)
val lines = res!!read() // bad!
val lines = res?.read() ?: throw IAE("$uri invalid") // more reasonable
```

Stick to nullable types only

```
public Optional<Goody> findGoodyForAmount(amount:Double)

val goody = findGoodyForAmount(100)
if(goody.isPresent()) goody.get() ... else ... // bad

val goody = findGoodyForAmount(100).orElse(null)
if(goody != null) goody ... else ... // good uses null consistently
```

Use nullability where applicable but don't overuse it.

```
data class Order(  
    val id: Int? = null,  
    val items: List<LineItem>? = null,  
    val state: OrderState? = null,  
    val goody: Goody? = null  
) // too much!  
  
data class Order(  
    val id: Int? = null,  
    val items: List<LineItem> = emptyList(),  
    val state: OrderState = UNPAID,  
    val goody: Goody? = null  
) // some types made more sense as not-null values
```

Avoid using nullable types in Collections

```
val items: List<LineItem?> = emptyList()  
val items: List<LineItem>? = null,  
val items: List<LineItem?> = null // all terribad  
  
val items: List<LineItem> = emptyList() // that's what this is for
```

3. Get The Most Out Of Classes and Objects

Use immutable data classes for value classes, config classes etc.

You'll be surprised how many classes you create that are just data classes.

```
class Person(val name: String, val age: Int)
val p1 = Person("Joe", 42)
val p2 = Person("Joe", 42)
p1 == p2 // false

data class Person(val name: String, val age: Int)
val p1 = Person("Joe", 42)
val p2 = Person("Joe", 42)
p1 == p2 // true
```



Use normal classes instead of data classes for services etc.

```
class PersonService(val dao: PersonDao) {  
    fun create(p: Person) {  
        if (p.age >= MAX_AGE)  
            LOG.warn("$p ${bornInYear(p.age)} too old")  
        dao.save(p)  
    }  
    companion object {  
        val LOG = LoggerFactory.getLogger()  
        val MAX_AGE = 120  
        fun bornInYear(age: Int) = ...  
    }  
}
```



Use value classes for domain specific types instead of common types.

```
value class Email(val value: String)
value class Password(val value: String)

fun login(email: Email, pwd: Password) // no performance impact! type erased in bytecode
```

Value denotes an inline value-based class. They have restrictions compared to regular classes, but are inline and type-erased. Useful for high performing code.

Seal classes for exhaustive branch checks.

```
// problematic
data class Square(val length: Double)
data class Circle(val radius: Double)

when (shape) {
  is Circle -> "..."
  is Rectangle -> "..."
  else -> throw IAE("unknown shape $shape") // annoying
}

// fixed
sealed interface Shape // prevents additions
data class Square(val length: Double): Shape
data class Circle(val radius: Double): Shape

when (shape) {
  is Circle -> "..."
  is Rectangle -> "..."
}
```

Sealed means “no more classes can implement this interface than exist in this file. This is why there is no ‘else’ clause required.

4. Use Available Extensions

```
// bad  
val fis = FileInputStream("path")  
val text = try {  
    val sb = StringBuilder()  
    var line: String?  
    while(fis.readLine().apply {line = this} != null) {  
        sb.append(line).append(System.lineSeparator())  
    }  
    sb.toString()  
} finally {  
    try { fis.close() } catch (ex:Throwable) { }  
}  
  
// good, via extension functions  
val text = FileInputStream("path").use { it.reader().readText() }
```

5. Use control-flow appropriately

Use if/else for single branch conditions rather than when

```
// too verbose  
val reduction = when {  
    customer.isVip() -> 0.05  
    else -> 0.0  
}  
  
// better  
val reduction = if (customer.isVip()) 0.05 else 0.0
```



Use `when` for multi-branch conditions.

```
fun reduction(customerType: CustomerTypeEnum) = when (customerType) {  
    REGULAR -> 0  
    GOLD -> 0.1  
    PLATINUM -> 0.3  
}
```

6. Expression Oriented Programming

Imperative Programming

- Imperative programming relies on declaring variables that are mutated along the way.
 - i.e. var, loops, mutable collections, mutating data, side effects.

```
var kotlinDevs = mutableListOf<Person>()
for (person in persons) {
    if (person.langs.contains("Kotlin"))
        kotlinDevs.add(person)
}
kotlinDevs.sort()
```

Expression Oriented Programming

- Expression oriented programming relies on thinking in functions where every input results in an output.
 - i.e., val, functions, read-only collections, input/output, transforming data

```
val kotlinDevs = persons.filter { it.langs.contains("Kotlin") }.sorted()
```

This is better because it results in more concise, deterministic, more easily testable and clearly scoped code that is easy to reason about compared to the imperative style.

if/else is an expression returning a result.

```
// imperative style  
var result: String  
if(number % 2 == 0)  
    result = "EVEN"  
else  
    result = "ODD"  
  
// expression style, better  
val result = if(number % 2 == 0) "EVEN" else "ODD"
```

when is an expression too, returning a result.

```
// imperative style
var hi: String
when(lang) {
    "NL" -> hi = "Goede dag"
    "FR" -> hi = "Bonjour"
    else -> hi = "Good day"
}

// expression style, better
val hi = when(lang) {
    "NL" -> "Goede dag"
    "FR" -> "Bonjour"
    else -> "Good day"
}
```

try/catch also.

```
// imperative style  
var text: String  
try {  
    text = File("path").readText()  
} catch (ex: Exception) {  
    text = ""  
}  
  
// expression style, better  
val text = try {  
    File("path").readText()  
} catch (ex: IOException) {  
    ""  
}
```




Most functional collections return a result, so the `return` keyword is rarely needed!

```
fun firstAdult(ps: List<Person>, age: Int) =  
    ps.firstOrNull{ it.age >= 18 }
```

7. Functional Collections Over For-Loops

Program on a higher abstraction level with (chained) higher-order functions from the collection.

```
// bad!  
val kids = mutableSetOf<Person>()  
for(person in persons) {  
    if(person.age < 18) kids.add(person)  
}  
names.sorted()  
  
// better!  
val kids: mutableSetOf<Person> = persons.filter{ it.age < 18}
```



For readability, write multiple chained functions from *top-down* instead of *left-right*.

```
// bad!
val names = mutableSetOf<String>()
for(person in persons) {
    if(person.age < 18)
        names.add(person.name)
}
names.sorted()

// better!
val names = persons.filter{ it.age < 18 }
                    .map{ it.name }
                    .sorted()
```

Use intermediate variables when chaining more than ~3-5 operators.

```
// bad!  
val sortedAgeGroupNames = persons  
    .filter{ it.age >= 18 }  
    .groupBy{ it.age / 10 * 10 }  
    .mapValues{ it.value.map{ it.name } }  
    .toList()  
    .sortedBy{ it.first }  
  
// better, more readable  
val ageGroups = persons.filter{ it.age >= 18 }  
                        .groupBy{ it.age / 10 * 10 }  
val sortedNamesByAgeGroup = ageGroups  
    .mapValues{ (_, group) -> group.map(Person::name) }  
    .toList()  
    .sortedBy{ (ageGroup, _) -> ageGroup }
```

8. Scope Your Code

Use `apply/with` to configure a mutable object.

```
// old way
fun client(): RestClient {
    val client = RestClient()
    client.username = "xyz"
    client.secret = "secret"
    client.url = "https://....employees"
    return client
}

// better way
fun client() = RestClient().apply {
    username = "xyz"
    secret = "secret"
    url = "https://....employee"
}
```


Use `let/run` to manipulate the context object and return a different type.

```
// old way
val file = File("/path")
file.setReadOnly(true)
val created = file.createNewFile()

// new way
val created = File("/path").run {
  setReadOnly(true)
  createNewFile() // last expression so result from this function is returned
}
```

Use also to execute a side-effect.

```
// old way
if(amount <= 0) {
    val msg = "Payment amount is < 0"
    LOGGER.warn(msg)
    throw IAE(msg)
} else ...

// new way
require(amount > 0) {
    "Payment amount is < 0".also(LOGGER::warn)
}
```

Reference

- Leeds. 2025. [Dave Leeds on Kotlin](#). Online.
- Leeds. 2025. **Kotlin: An Illustrated Guide**. TypeAlias Studios LLC. ISBN 979-8992796605.
- JetBrains. 2025. [Kotlin Documentation](#). Online.
- Elizarov, et al. 2024. **Kotlin in Action**. 2nd edition. Manning Publications. ISBN 9781617299605.
- Vermeulen et al. 2021. **Functional Programming with Kotlin**. Manning. ISBN 978-1617297168.