Idiomatic Kotlin

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

What is Idiomatic Kotlin?

`Idiomatic` in this context means *using the language as the designers intended, and taking advantage of advanced language features*. [1]

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. Leveraging its unique language features can lead you to more efficient and effective use of the language and its libraries.

> These notes summarize a talk by Urs Peters @ Kotlin Dev Day 2022 Idiomatic Kotlin: the key to unlocking Kotlin's true potential.

Principles

1. Favor immutability

Kotlin favors immutability with various immutable constructs and defaults.

What is so good about immutability?

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

Immutable state has some helpful qualities:

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 states deterministic	Side effects: can lead to unexpected failures
Thread-safety	Inherently thread-safe	Manual synchronization

1. Favor immutability

```
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")
```

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! what if it fails?
val lines = res?.read() ?: throw IAE("$uri invalid")
```

Stick to nullable types

```
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
```

2. Use nullability appropriately

Use nullability where applicable but don’t overruse it

```
// this is a lot of nulls
data class Order(
  val id: Int? = null,
  val items: List<LineItem> = null,
  val state: OrderState = null,
  val goody: Goody? = null
)

// sometimes other options make more sense!
data class Order(
  val id: Int? = null,
  val items: List<LineItem> = emptyList(), // more clear
  val state: OrderState = UNPAID, // more clear
  val goody: Goody? = null
)
```

2. Use nullability appropriately

Avoid nullable types in collections

```
// these are all terrible
val items: List<LineItem?>> = emptyList()
val items: List<LineItem>? = null
val items: List<LineItem?>? = null

// empty replaces nulls
val items: List<LineItem> = emptyList()
```

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. Use normal classes instead of data classes for services etc.

```
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
```

3. Get the most out of classes and objects

Use value classes for domain specific types instead of common types! They are type-erased in bytecode and have zero performance overhead.

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

fun login(email: Email, pwd: Password) // no performance impact
```

3. Get the most out of classes and objects

`Sealed` means “no more classes can implement this interface than already exist in this file.

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

when (shape) {
  is Circle → ...
  is Rectangle → ...
  else → throw IllegalArgumentException("Unknown shape") // annoying
}

// sealed removes the need for an else statement
sealed interface Shape
data class Square(val length: Double): Shape
data class Circle(val radius: Double): Shape

when (shape) {
  is Circle → ...
  is Rectangle → ...
}
```

4. Use available extensions

Become familiar with the Kotlin Standard Library.

```
// old-school
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) { }
}

// built-in extensions
val text = FileInputStream("path").use { it.reader().teadText() }
```

Use control-flow appropriately

Use `if/else` for single-branch conditions rather than `when`.

```
// lengthy
val reduction = when {
  customer.isVip() → 0.05
  else → 0.0
}

// better
val reduction = if (customer.isVip()) 0.05 else 0.00
```

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 relies on declaring variables that are mutated along the way e.g., vars, loop.

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

Expression oriented programming relies on thinking in functions where every input results in an output.

```
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.

7. Functional collections over for-loops

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

```
val kids = mutableSetOf<Person>()
for (person in persons) {
  if (person.age < 18) kids.add(person)
}
names.sorted()

// better
val kids = persons.filter {it.age < 18}.sorted()
```

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

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

8. Scope your code

Use apply/with to configure a mutable object.

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

// better way creates, then applies changes
fun create() = RestClient().apply {
  username = "xyz"
  secret = "secret"
  url = "https:// ... /employees"
}
```

8. Scope your code

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() // result from this function is returned
}
```

8. Scope your code

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 {
  // do something else
}

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

Bibliography

[1] Urs Peters, “Idiomatic Kotlin: the key to unlocking Kotlin's true potential.” [Online]. Available: https://www.youtube.com/watch?v=zYH6zTtl-nc