Naming rules

Kotlin follows the Java naming conventions. In particular:

Names of packages are always lower case and do not use underscores (org.example.myproject).

Using multi-word names is generally discouraged, but if you do need to use multiple words, you can either simply concatenate them together or use camel humps (org.example.myProject).

Names of classes and **objects** start with an upper case letter and use camel humps:

```
open class DeclarationProcessor { ... }
object EmptyDeclarationProcessor : DeclarationProcessor()
{ ... }
```

Function names

Names of functions, properties and local variables start with a lower case letter and use camel humps and no underscores:

```
fun processDeclarations() { ... }
var declarationCount = ...
```

Exception: factory functions used to create instances of classes can have the same name as the class being created:

```
abstract class Foo { ... }
class FooImpl : Foo { ... }
fun Foo(): Foo { return FooImpl(...) }
```

Names for test methods

In tests (and only in tests), it's acceptable to *use method names with spaces enclosed in backticks*. (Note that such method names are currently not supported by the Android runtime.) Underscores in method names are also allowed in test code.

```
class MyTestCase {
    @Test fun `ensure everything works`() { ... }
    @Test fun ensureEverythingWorks_onAndroid() { ... }
}
```

Property names

Names of constants (properties marked with const, or top-level or object val properties with no custom get function that hold deeply immutable data) should use uppercase underscore-separated names:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

Formatting

In most cases, Kotlin follows the Java coding conventions.

Use 4 spaces for indentation. Do not use tabs.

For curly braces, put the opening brace in the end of the line where the construct begins, and the closing brace on a separate line aligned horizontally with the opening construct.

```
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

Horizontal whitespace

- Put spaces around binary operators (a + b).
- Put spaces between control flow keywords (if, when, for and while) and the corresponding opening parenthesis.

Exception:

- Don't put spaces around the "range to" operator (0..i).
- Do not put spaces around unary operators (a++).
- Do not put a space before an opening parenthesis in a primary constructor declaration, method declaration or method call.

```
class A(val x: Int)
fun foo(x: Int) { ... }
fun bar() {
    foo(1)
}
```

- Never put a space after (, [, or before],).
- Never put a space around . or ?.: foo.bar().filter { it > 2 }.joinToString(), foo?.bar()
- Put a space after //: // This is a comment
- Do not put spaces around angle brackets used to specify type parameters: class Map<K, V> { ... }
- Do not put spaces around ::: Foo::class, String::length
- Do not put a space before ? used to mark a nullable type: String?

Colon

- Put a space before : in the following cases:
- when it's used to separate a type and a supertype;
- when delegating to a superclass constructor or a different constructor of the same class;
- after the object keyword.
- Don't put a space before : when it separates a declaration and its type.
- Always put a space after :.

Class header formatting

- Classes with a few primary constructor parameters can be written in a single line:

```
class Person(id: Int, name: String)
```

- Classes with longer headers should be formatted so that each primary constructor parameter is in a separate line with indentation.

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { ... }
```

- For multiple interfaces, the superclass constructor call should be located first and then each interface should be located in a different line:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { ... }
```

Modifiers

If a declaration has multiple modifiers, always put them in the following order:

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
tailrec
vararg
suspend
inner
enum / annotation
companion
inline
infix
operator
data
```

Place all annotations before modifiers:

```
@Named("Foo")
private val foo: Foo
```

Annotation formatting

Annotations are typically placed on separate lines, before the declaration to which they are attached, and with the same indentation:

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

Annotations without arguments may be placed on the same line:

```
@JsonExclude @JvmField
var x: String
```

A single annotation without arguments may be placed on the same line as the corresponding declaration:

```
@Test fun foo() { ... }
```

File annotations

File annotations are placed after the file comment (if any), before the package statement, and are separated from package with a blank line (to emphasize the fact that they target the file and not the package).

```
/** License, copyright and whatever */
@file:JvmName("FooBar")
package foo.bar
```

Function formatting

If the function signature doesn't fit on a single line, use the following syntax:

```
fun longMethodName(
    argument1: ArgumentType = defaultValue,
    argument2: AnotherArgumentType
): ReturnType {
    // body
}
```

Coding conventions for libraries

Follow an additional set of rules to ensure API stability:

- Always explicitly specify member visibility (to avoid accidentally exposing declarations as public API)
- Always explicitly specify function return types and property types (to avoid accidentally changing the return type when the implementation changes)
- Provide KDoc comments for all public members, with the exception of overrides that do not require any new documentation (to support generating documentation for the library)

Refer:https://kotlinlang.org/docs/reference/coding-conventions.html

https://kotlinlang.org/docs/reference/basic-syntax.html

https://codelabs.developers.google.com/codelabs/build-your-first-android-app-kotlin/index.html#0