Kotlin Language Documentation

Table of Contents

□□

□□□□

□□□

□□□□□□□□□□□□□□□

```
package my.demo

import java.util.*

// ...
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ □.

□□□□

□□□□ `Int` □□□□□ `Int` □□□:

```
fun sum(a: Int, b: Int): Int {
  return a + b
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
fun sum(a: Int, b: Int) = a + b
```

□□□□□□□□□□:

```
fun printSum(a: Int, b: Int): Unit {
  print(a + b)
}
```

`Unit` □□□□□□□□□:

```
fun printSum(a: Int, b: Int) {
  print(a + b)
}
```

□□ □□.

□□□□□□□

□□□□□ `val` □□□□□□□:

```
val a: Int = 1
val b = 1  // `Int` □□□□□□□
val c: Int // □□□□□□□□□□□□□□□□□□□□□□□□□□□□
c = 1      // □□□□
```

□□□□□ `var` □□□□□□□:

```
var x = 5 // `□□□□□ Int` □□
x += 1
```

□□ □□□□□.

□□

□□ Java □ JavaScript□Kotlin □□□□□□□□□□

```
// □□□□□□□□

/* □□□□□□□□
   □□□□ */
```

□ Java □□□□□Kotlin □□□□□□□□□

□□ □□ Kotlin □□□□□□□□□□□□□□□□□□□

□□□□□□□

```
fun main(args: Array<String>) {
  if (args.size == 0) return

  print("First argument: ${args[0]}")
}
```

□□ □□□□□.

□□□□□□

```
fun max(a: Int, b: Int): Int {
  if (a > b)
    return a
  else
    return b
}
```

□□ if □□□□□:

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

□□ if □□□.

□□□□□□ null □□

□□□□□□□□□□null □□□□□□□□□□□□□□□□□ ? □□□□□□□□□□

□□ `str` □□□□□□□□□null:

```
fun parseInt(str: String): Int? {
  // ...
}
```

□□□□□□□□:

```
fun main(args: Array<String>) {
  if (args.size < 2) {
    print("Two integers expected")
    return
  }

  val x = parseInt(args[0])
  val y = parseInt(args[1])

  // □□□□ `x * y` □□□□□□□□□□□□□□ null
  if (x != null && y != null) {
    // □□□□□□□□x □ y □□□□□□□□□□non-nullable□
    print(x * y)
  }
}
```

□□

```
  // ...
  if (x == null) {
    print("Wrong number format in '${args[0]}'")
    return
  }
  if (y == null) {
    print("Wrong number format in '${args[1]}'")
    return
  }

  // □□□□□□□□x □ y □□□□□□□□□
  print(x * y)
```

□□ Null □□□.

□□□□□□□□□□□□□□

is □□□□□□□□□: □□□□□□□□□□□□□□ □□

```
fun getStringLength(obj: Any): Int? {
  if (obj is String) {
    // `obj` □□□□□□□□□□□□□□□ `String`
    return obj.length
  }

  // □□□□□□□□□□□□ `obj` □□□ `Any` □□
  return null
}
```

□□

```
fun getStringLength(obj: Any): Int? {
  if (obj !is String)
    return null

  // `obj` □□□□□□□□□□□□□□□ `String`
  return obj.length
}
```

□□

```
fun getStringLength(obj: Any): Int? {
  // `obj` □ `&&` □□□□□□□ `String` □□
  if (obj is String && obj.length > 0)
    return obj.length

  return null
}
```

□□ □ and □□□□.

□□ for □□

```
fun main(args: Array<String>) {
  for (arg in args)
    print(arg)
}
```

□□

```
for (i in args.indices)
  print(args[i])
```

□□ for□□.

Using a while loop

```
fun main(args: Array<String>) {
  var i = 0
  while (i < args.size)
    print(args[i++])
}
```

See while □□.

□□ when □□□

```
fun cases(obj: Any) {
  when (obj) {
    1          -> print("One")
    "Hello"    -> print("Greeting")
    is Long    -> print("Long")
    !is String -> print("Not a string")
    else       -> print("Unknown")
  }
}
```

□□ when□□□.

□□□□□range□

□□ in □□□□□□□□□□□□□□□□□□□□□□

```
if (x in 1..y-1)
  print("OK")
```

□□□□□□□□□□□□□□□□□:

```
if (x !in 0..array.lastIndex)
  print("Out")
```

□□□□□:

```
for (x in 1..5)
  print(x)
```

□□ □□ Range.

□□□□

□□□□□□□□:

```
for (name in names)
  println(name)
```

□□ in □□□□□□□□□□□□□□□□□□□□

```
if (text in names) // □□□ names.contains(text)
  print("Yes")
```

□□ lambda □□□□□□□□filter□□□□□map□□□□

```
names
    .filter { it.startsWith("A") }
    .sortedBy { it }
    .map { it.toUpperCase() }
    .forEach { print(it) }
```

□□ □□□□□Lambda□□□.

□□□□

□□□ Kotlin □□ pull request □□□□□□□□

□□ DTOs□POJOs/POCOs□

```
data class Customer(val name: String, val email: String)
```

□□ `Customer` □□□□□□□□□

— □□□□□□ getters □□□var □□□□□□ setters□

— `equals()`

— `hashCode()`

— `toString()`

— `copy()`

— □□□□□□ `component1()` , `component2()` , … □□ (□□□□□□)

□□□□□□□□□

```
fun foo(a: Int = 0, b: String = "") { ... }
```

□□ list

```
val positives = list.filter { x -> x > 0 }
```

□□□□□□□:

```
val positives = list.filter { it > 0 }
```

String □□

```
println("Name $name")
```

□□□□□

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

□□ map/pair□list

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

k 和 v 分别表示键和值

遍历区间（range）

```
for (i in 1..100) { ... }  // closed range: includes 100
for (i in 1 until 100) { ... } // half-open range: does not include 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

创建 list

```
val list = listOf("a", "b", "c")
```

创建 map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

访问 map

```
println(map["key"])
map["key"] = value
```

懒加载

```
val p: String by lazy {
    // compute the string
}
```

扩展函数

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

创建单

```
object Resource {
    val name = "Name"
}
```

If not null □□

```
val files = File("Test").listFiles()

println(files?.size)
```

If not null and else □□

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

if null □□□□□□

```
val data = ...
val email = data["email"] ?: throw IllegalStateException("Email is missing!")
```

if not null □□□□

```
val data = ...

data?.let {
    ... // □□□□□□□□□, □□data□□null
}
```

□□when□□□

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

‘try/catch’ □□□

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // Working with result
}
```

‘if’ □□□

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

□□□□□ **Unit** □□□□ **Builder** □□□□

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

□□□□□□

```
fun theAnswer() = 42
```

□□□

```
fun theAnswer(): Int {
    return 42
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□when □□□□□□□□□

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

□□□□□□□□□□□□□□ □**with**□

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Java 7 □ try with resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

Convenient form for a generic function that requires the generic type information

```
//  public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT) throws
JsonSyntaxException {
//     ...

inline fun <reified T: Any> Gson.fromJson(json): T = this.fromJson(json, T::class.java)
```

Consuming a nullable Boolean

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b` is false or null
}
```

□□□□

□□□□□□□ Kotlin □□□□□□□

□□□□

□□□□□□□□□□□□□□Java□□□□□□□□□□

— □□□□□□□□□□□□□□□□□□□□□

— □□□□□□□□□□□

— □□□□□□□□□□□□□

— □□ 4 □□□□□

— □□□□□□□□□□□□□□□□□□□□□□□□□ Kotlin Doc □

□□

□□□

```
interface Foo<out T : Any> : Bar {
    fun foo(a: Int): T
}
```

Lambda□□□

□lambda□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □lambda□□□□□□□□□□□□□□□□

```
list.filter { it > 10 }.map { element -> element * 2 }
```

□□□□□□lambda□□□□□□□□□□□□□□□□□□□□ `it` □□□□□□□□□□□ □□□□□□□□□lambda□□□□□□□□□□□□□□□□□□

Unit

□□□□□□ Unit □□□□□□□□□□□□□□

```
fun foo() { // □□□ ": Unit"

}
```

Functions vs Properties

In some cases functions with no arguments might be interchangeable with read-only properties. Although the semantics are similar, there are some stylistic conventions on when to prefer one to another.

Prefer a property over a function when the underlying algorithm:

— does not throw

— has a `O(1)` complexity

— is cheap to calculate (or cached on the first run)

— returns the same result over invocations

[illegible]

[illegible]

[illegible] Kotlin [illegible]

[illegible]

Kotlin [illegible] Java[illegible] Java `int` [illegible]`long`——[illegible])[illegible]

Kotlin [illegible] Java [illegible]

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

[illegible] Kotlin [illegible]

[illegible]

[illegible]:

- [illegible]: `123`
  - Long [illegible]`L` [illegible]: `123L`
- [illegible]: `0x0F`
- [illegible]: `0b00001011`

[illegible]: [illegible]

Kotlin [illegible]:

- [illegible] double[illegible]`123.5`[illegible]`123.5e10`
- Float [illegible] `f` [illegible] `F` [illegible]: `123.5f`

□□□□

□ Java □□□□□□□□□□ JVM □□□□□□□□□□□□□□□□□□□□□□□□ `Int?` □□□□□ □□□□□□□□□□□□□

□□□□□□□□□□□□□□□□:

```
val a: Int = 10000
print(a === a) // □□ 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!□□ 'false'!!!
```

□□□□□□□□□□□□□□□:

```
val a: Int = 10000
print(a == a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // Prints 'true'
```

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

```
// □□□□□□□□□□□□□□□□□□
val a: Int? = 1 // □□□□□ Int (java.lang.Integer)
val b: Long? = a // □□□□□□□□□□□□□ Long (java.lang.Long)
print(a == b) // □□□□□□□ "false" □□ Long □ equals() □□□□□□□□□ Long
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□ `Byte` □□□□□□□ `Int` □□□

```
val b: Byte = 1 // OK, □□□□□□□□□□
val i: Int = b // □□
```

□□□□□□□□□□□□□□□□

```
val i: Int = b.toInt() // OK: □□□□
```

□□□□□□□□□□□□□□□□:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`

— `toDouble(): Double`

— `toChar(): Char`

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val l = 1L + 3 // Long + Int => Long
```

□□

Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□:

```
val x = (1 shl 2) and 0x000FF000
```

□□□□□□□□□□ `Int` □ `Long` □□

— `shl(bits)` – □□□□□ (Java’s `<<`)

— `shr(bits)` – □□□□□ (Java’s `>>`)

— `ushr(bits)` – □□□□□ (Java’s `>>>`)

— `and(bits)` – □□

— `or(bits)` – □□

— `xor(bits)` – □□□

— `inv()` – □□

□□

□□□ `Char` □□□□□□□□□□□□□□

```
fun check(c: Char) {
  if (c == 1) { // □□□□□□□□
    // ...
  }
}
```

□□□□□□□□□□□□: `'1'` □ □□□□□□□□□□□□□ □□□□□□□□ `\t` □ `\b` □ `\n` □ `\r` □ `\'` □ `\"` □ `\\` □ `\$` □ □□□□□□□□
Unicode □□□□□□□ `'\uFF00'` □

□□□□□□□□□□□ `Int` □□□

```
fun decimalDigitValue(c: Char): Int {
  if (c !in '0'..'9')
    throw IllegalArgumentException("Out of range")
  return c.toInt() - '0'.toInt() // □□□□□□□
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□

□□□ Boolean □□□□□□□□□□□true □ false□

□□□□□□□□□□□□□□□□

□□□□□□□□□□□

— || – □□□□□

— && – □□□□□

— ! - □□□

□□

□□□ Kotlin □□□ Array □□□□□□□□□ get □ set □□□□□□□□□□□□□□□□□□[] □□ size □□□□□□□□□□□□□□□□□□□

```
class Array<T> private constructor() {
  val size: Int
  fun get(index: Int): T
  fun set(index: Int, value: T): Unit

  fun iterator(): Iterator<T>
  // ...
}
```

□□□□□□□□□□ arrayOf() □□□□□□□□□□□□□□□□□□□□arrayOf(1, 2, 3) □□□ array [1, 2, 3]□ □□□□□□
arrayOfNulls() □□□□□□□□□□□□□□□□□□□□□□□□□□□

□□ □□□□□□□□□□□□□□□□

```
// □□□□ Array<String> □□□□ ["0", "1", "4", "9", "16"]
val asc = Array(5, { i -> (i * i).toString() })
```

□□□□□□[] □□□□□□□□□□□□□get() □ set()□

□□: □ Java □□□□□Kotlin □□□□□□□□□□□invariant□□□□□□□ Kotlin □□□□□Array<String> □□□ Array<Any> □□□
□□□□□□□□□□□□□□□□□□□□□ Array<out Any>, □□□□□□□□

Kotlin □□□□□□□□□□□□□□□□□□□□□□□□□□□ByteArray□ ShortArray□IntArray □□□□□□□□ Array □□□□□□□□□□□□ □□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

□□□

□□□□ String □□□□□□□□□□□□□□□□□□ □□□□□□——□□□□□□□□□□□□□□□□□s[i] □ □□□ for □□□□□□□□:

```
for (c in str) {
  println(c)
}
```

□□□□□□

Kotlin □□□□□□□□□□□□□: □□ Java □□□:

```
val s = "Hello, world!\n"
```

□□□□□□□□□□□□□□□□□□[□□] □□□□□□□□□□

□□□□□ □□□□□□ `"""` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
val text = """
  for (c in "foo")
    print(c)
"""
```

You can remove leading whitespace with [trimMargin()] function:

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

By default `|` is used as margin prefix, but you can choose another character and pass it as a parameter, like `trimMargin(">")`.

□□□□□

□□□□□□□ *□□□□□* □□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ `$` □□□□□□□□□□□□□□:

```
val i = 10
val s = "i = $i" // □□□□□ "i = 10"
```

□□□□□□□□□□□□□□□:

```
val s = "abc"
val str = "$s.length is ${s.length}" // □□□□□ "abc.length is 3"
```

□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□ `$` □□□□□□□□□□□□□□□□□□□□□□□

```
val price = """
${'$'}9.99
"""
```

□

□□□□□□□□□□□□□□:

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□`baz()` □□□□`foo.bar.baz`□`Goo` □□□□`foo.bar.Goo`□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□

□□□□□□□□□□□□□□□□.

```
import foo.Bar // □□ Bar □□□□□□□□□□
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
import foo.* // 'foo' □□□□□□□□□
```

□□□□□□□□□□□□□□□as □□□□□□□□□□□□□□□□□□□□

```
import foo.Bar // Bar □□□
import bar.Bar as bBar // bBar □□ 'bar.Bar'
```

□□□ `import` □□□□□□□□□□□□□□□□□□□□□□□□□

— □□□□□□□

— □□□□□□□□□□□□□□□;

— □□□□

□ Java □□□Kotlin □□□□□ “import static” □□□ □□□□□□□□`import` □□□□□□

□□□□□□□

□□□□□□□private □□□□□□□□□□□□□□□□□□□□□□□□□□

□□□

If□□□

□ Kotlin □□if□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□ ? □□ : □□□□□□□□if □□□□□□□□□

```
// □□□□
var max = a
if (a < b)
  max = b

// □ else
var max: Int
if (a > b)
  max = a
else
  max = b

// □□□□□
val max = if (a > b) a else b
```

if□□□□□□□□□□□□□□□□□□□□□□□□□

```
val max = if (a > b) {
    print("Choose a")
    a
  }
  else {
    print("Choose b")
    b
  }
```

□□□□□ if □□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□ `else` □□□

□□ if □□□

When□□□

when □□□□ C □□□ switch □□□□□□□□□□□□□□□□

```
when (x) {
  1 -> print("x == 1")
  2 -> print("x == 2")
  else -> { // □□□□□
    print("x is neither 1 nor 2")
  }
}
```

when □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□when □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□ if □□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□

[illegible]else [illegible] [illegible] when [illegible]else [illegible] [illegible]

[illegible]

```
when (x) {
  0, 1 -> print("x == 0 or x == 1")
  else -> print("otherwise")
}
```

[illegible]

```
when (x) {
  parseInt(s) -> print("s encodes x")
  else -> print("s does not encode x")
}
```

[illegible]in[illegible]!in[illegible][illegible][illegible]

```
when (x) {
  in 1..10 -> print("x is in the range")
  in validNumbers -> print("x is valid")
  !in 10..20 -> print("x is outside the range")
  else -> print("none of the above")
}
```

[illegible]is[illegible]!is[illegible] [illegible][illegible][illegible] [illegible]

```
val hasPrefix = when(x) {
  is String -> x.startsWith("prefix")
  else -> false
}
```

when [illegible]if-else if[illegible] [illegible]

```
when {
  x.isOdd() -> print("x is odd")
  x.isEven() -> print("x is even")
  else -> print("x is funny")
}
```

[illegible] when [illegible][illegible]

For[illegible]

for [illegible]iterator[illegible]:

```
for (item in collection)
  print(item)
```

□□□□□□□□□□□□□□

```
for (item: Int in ints) {
  // ...
}
```

□□□□□□for □□□□□□□□□□□□□□□□□□□□□□□□□□□□

— □□□□□□□□□□□□□□□ iterator() □□□□□□□□

— □□□□□□□□□□□□□□□ next() □□□

— □□□□□□□□□□□□□□□ hasNext() □□ Boolean □

□□□□□□□□□□□□ operator □

□□□□ for □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□ list□□□□□□□□□

```
for (i in array.indices)
  print(array[i])
```

□□□□“□□□□□□□”□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□ withIndex □

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

□□ for □□□

While□□

while □ do..while □□□□

```
while (x > 0) {
  x--
}

do {
  val y = retrieveData()
} while (y != null) // y □□□□□
```

□□ while □□.

□□□□Break□continue

□□□□ Kotlin □□□□□□break □ continue □□□□□□□□□□□□

[illegible]

Kotlin [illegible]

- return.[illegible][illegible][illegible]
- break.[illegible]
- continue.[illegible]

Break[illegible]Continue[illegible]

[illegible] Kotlin [illegible]label[illegible] [illegible] `@` [illegible] `abc@`[illegible]`fooBar@`[illegible][illegible][illegible] [illegible]

```
loop@ for (i in 1..100) {
  // ...
}
```

[illegible]break [illegible]continue[illegible]

```
loop@ for (i in 1..100) {
  for (j in 1..100) {
    if (...)
      break@loop
  }
}
```

[illegible] break [illegible]continue [illegible]

[illegible]

Kotlin [illegible] Kotlin [illegible] [illegible]return [illegible] [illegible] lambda [illegible]

```
fun foo() {
  ints.forEach {
    if (it == 0) return
    print(it)
  }
}
```

[illegible] return [illegible]`foo` [illegible] [illegible][illegible] lambda [illegible] [illegible] lambda [illegible] return[illegible]

```
fun foo() {
  ints.forEach lit@ {
    if (it == 0) return@lit
    print(it)
  }
}
```

□□□□□□□□ lambda □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ lambda □□□□□□□

```
fun foo() {
  ints.forEach {
    if (it == 0) return@forEach
    print(it)
  }
}
```

□□□□□□□□□□□□□□ lambda □□□□□ □□□□□□□□□return □□□□□□□□□□□□□□□□

```
fun foo() {
  ints.forEach(fun(value: Int) {
    if (value == 0) return
    print(value)
  })
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ return□□

```
return@a 1
```

□□“□□□ `@a` □□ 1”□□□□“□□□□□□□□□□□□□□□(`@a 1`)”□

□□□□

□□□□

□

Kotlin □□□□□class □□□

```
class Invoice {
}
```

□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

```
class Empty
```

□□□□

□ Kotlin □□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
class Person constructor(firstName: String) {
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□constructor □□□□

```
class Person(firstName: String) {
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□init □□□□□□□□□□□□□□□□**initializer blocks**□□□

```
class Customer(name: String) {
    init {
        logger.info("Customer initialized with value ${name}")
    }
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

[illegible]Kotlin [illegible]

```
class Person(val firstName: String, val lastName: String, var age: Int) {
  // ...
}
```

[illegible] [illegible]var[illegible]val[illegible]

[illegible]constructor [illegible] [illegible]

```
class Customer public @Inject constructor(name: String) { ... }
```

[illegible][illegible]

[illegible]

[illegible]constructor** [illegible]**[illegible]

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

[illegible] [illegible]this [illegible]

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

[illegible] [illegible] public[illegible] [illegible]
[illegible]

```
class DontCreateMe private constructor () {
}
```

> [illegible] JVM [illegible] [illegible] Kotlin [illegible] Jackson [illegible] JPA [illegible]
>
> ```
> class Customer(val customerName: String = "")
> ```

□□□□□

□□□□□□□□□□□□□□□□□□□□□□□

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

□□ Kotlin □□□new □□□□

Creating instances of nested, inner and anonymous inner classes is described in Nested classes.

□□□

□□□□□

— □□□□□□□□□

— □□

— □□

— □□□□□□□

— □□□□

□□

□ Kotlin □□□□□□□□□□□□□ `Any` □□□□□□□□□□□□□□□□□□□□

```
class Example // □ Any □□□□
```

`Any` □□ `java.lang.Object` □□□□□□□□ `equals()` □ `hashCode()` □ `toString()` □□□□□□□□ □□□□□□□Java□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□ □□*super*{□.keyword} □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
class MyView : View {
    constructor(ctx: Context) : super(ctx) {
    }

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs) {
    }
}
```

□□□ *open*{□.keyword} □□□ Java □ *final*{□.keyword} □□□□□□□□□ □□□□□□□□□□□□□□ Kotlin □□□□□□□ final□□□□ [Effective Java]□□□ □ 17 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□

□□□□□□□□Kotlin □□□□□□□□ Java □□□Kotlin □□□□ □□□□□□□□□□□□□□*□□*□□□□□□□□□□

```
open class Base {
  open fun v() {}
  fun nv() {}
}
class Derived() : Base() {
  override fun v() {}
}
```

Derived.v() □□□□□□□**override**□□□□□□□□□□□□□□□□□ □□□□□□□**open** □ `Base.nv()` □□□□□□□□□□□□□□□□□□□ □□□□□ **override**□□□□ **final** □□□□□□ **open** □□□□□□□□□□□□□□□□

□□□ *override*{□.keyword} □□□*final*{□.keyword} □□□□

```
open class AnotherDerived() : Base() {
  final override fun v() {}
}
```

Overriding properties works in a similar way to overriding methods. Note that you can use the `override` keyword as part of the property declaration in a primary constructor:

```
open class Foo {
    open val x: Int get { ... }
}

class Bar1(override val x: Int) : Foo() {

}
```

You can also override a `val` property with a `var` property, but not vice versa. This is allowed because a `val` property essentially declares a getter method, and overriding it as a `var` additionally declares a setter method in the derived class.

□□□□□□□□hack□□□□

□□□□□□□□□□□□□□(□□□□□**final**)□□□□□□□□□□□□□□□□□□□□□hack□

□□□□□□□□□□□□□□□□□

— □□□□□□□□□□□□□□□hacks

— □□□□□□□□□□□□C++□C#□□□□□□□□□

— □□□□□□□ hack□□□□□□□□□□□□□□ Java □□ hack □□ Kotlin □□□*□□[Java □□□]*□□□□□□Aspect□□□□□□□□□□□□□□

□□□□

□ Kotlin □□□ □□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ super□□ `super<Base>`□

```
open class A {
  open fun f() { print("A") }
  fun a() { print("a") }
}

interface B {
  fun f() { print("B") } // □□□□□□□□□□ 'open' □
  fun b() { print("b") }
}

class C() : A(), B {
  // □□□□□□□□□ f()□
  override fun f() {
    super<A>.f() // □□ A.f()
    super<B>.f() // □□ B.f()
  }
}
```

□□□□ `A` □ `B` □□□□□□□ `a()` □ `b()` □□□□□□ `C` □□□□□□□□□□□□□□□□ □□ `f()` □ `C` □□□□□□□□□□□□□□□□ `C` □□□ `f()` □□□□□□□□□□□□□□□□□□□

□□□

□□□□□□□□□□□□□□□□abstract{□.keyword}□ □□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□ `open` □□□□□□□□□□□□□——□□ □□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□

```
open class Base {
  open fun f() {}
}

abstract class Derived : Base() {
  override abstract fun f()
}
```

□□□□

□ Java □ C# □□□□ Kotlin □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□ Java/C# □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□

□□□

[illegible] [illegible] [illegible]
[illegible] [illegible]

[illegible] `sealed` [illegible] [illegible]

```
sealed class Expr {
    class Const(val number: Double) : Expr()
    class Sum(val e1: Expr, val e2: Expr) : Expr()
    object NotANumber : Expr()
}
```

[illegible] [illegible]

[illegible] [when [illegible]] [illegible] [illegible] `else` [illegible]

```
fun eval(expr: Expr): Double = when(expr) {
    is Expr.Const -> expr.number
    is Expr.Sum -> eval(expr.e1) + eval(expr.e2)
    Expr.NotANumber -> Double.NaN
    // [illegible] else [illegible]
}
```

□□□□□

□□□□

Kotlin□□□□□□□□. □□□□□□□□□,□□□□var□□□□□□□□□□val.

```
public class Address {
  public var name: String = ...
  public var street: String = ...
  public var city: String = ...
  public var state: String? = ...
  public var zip: String = ...
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Java□□□□□□□□

```
fun copyAddress(address: Address): Address {
  val result = Address() // there's no 'new' keyword in Kotlin
  result.name = address.name // accessors are called
  result.street = address.street
  // ...
  return result
}
```

Getters □ Setters

□□□□□□□□□□□□

```
var <propertyName>: <PropertyType> [= <property_initializer>]
  [<getter>]
  [<setter>]
```

□□□□□□□□□□□(initializer)□getter □ setter □□□□□□□□□□□(PropertyType)□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□:

```
var allByDefault: Int? // error: explicit initializer required, default getter and setter implied
var initialized = 1 // has type Int, default getter and setter
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□1□□□□□□`val`□□□□`var` 2□□□□□□□□setter

```
val simple: Int? // has type Int, default getter, must be initialized in constructor
val inferredType = 1 // has type Int and a default getter
```

□□□□□□□□□□□□□□□□,□□□□□□□□,□□□□□□□□□□□□□□□□□□□□□□□□getter□□□:

```
val isEmpty: Boolean
  get() = this.size == 0
```

□□□□setter□□□:

```
var stringRepresentation: String
  get() = this.toString()
  set(value) {
    setDataFromString(value) // parses the string and assigns values to other properties
  }
```

□□□□,setter□□□□□□“value”,□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□,□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□:

```
var setterVisibility: String = "abc"
  private set // the setter is private and has the default implementation

var setterWithAnnotation: Any? = null
  @Inject set // annotate the setter with Inject
```

□□□□

□Kotlin□□□□□□□□□,□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□,Kotlin□□ □□□□,□□□□□□□ `field` □□□□

```
var counter = 0 // the initializer value is written directly to the backing field
  set(value) {
    if (value >= 0)
      field = value
  }
```

`field` □□□□□□□□□□□□□□□□□

A backing field will be generated for a property if it uses the default implementation of at least one of the accessors, or if a custom accessor references it through the `field` identifier.

□□□□□□□□□□ □□□□□□□□□

```
val isEmpty: Boolean
  get() = this.size == 0
```

□□□□

□□□□□□□□□□□□“□□□□□□□□“□□□□□□□□□□□“□□□□□□□”(backing property)□□□□

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
  get() {
    if (_table == null)
      _table = HashMap() // Type parameters are inferred
    return _table ?: throw AssertionError("Set to null by another thread")
  }
```

□□□□□□□□□□Java□□□Bean□□□□□□□□□□□□□□□□□□□□getter□setter□□□□□□□□□□□□□

Compile-Time Constants

Properties the value of which is known at compile time can be marked as*compile time constants* using the `const` modifier. Such properties need to fulfil the following requirements:

— Top-level or member of an `object`

— Initialized with a value of type `String` or a primitive type

— No custom getter

Such properties can be used in annotations:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Late-Initialized Properties

Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient. For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle this case, you can mark the property with the `lateinit` modifier:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```

The modifier can only be used on `var` properties declared inside the body of a class (not in the primary constructor), and only when the property does not have a custom getter or setter. The type of the property must be non-null, and it must not be a primitive type.

Accessing a `lateinit` property before it has been initialized throws a special exception that clearly identifies the property being accessed and the fact that it hasn't been initialized.

□□□□

□□ Overriding Members

□□□□

□□□□□□□□□□□□□□□□(□□)□ □□□□,□□□□□□getter□setter□□□□□□□□□□□□□□ □□□□□□□,□□□□□□□□□□□□□□□□□□□□□□□□□□□:lazy values,□□□□□□□□□,□□□□□□□□,□□□□□□□□,□□□

□□□□□□□□□□□□□□□□□*delegated properties*□

□□

Kotlin □□□□ Java 8 □□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□ abstract□□□□□□□□□

□□□□□ interface □□□□□□

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

□□□□

□□□□□□□□□□□□□□□□□

```
class Child : MyInterface {
   override fun bar() {
      // body
   }
}
```

□□□□

□□□

```
interface MyInterface {
    val property: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

□□□□□Override□□□

□□□□□□□□□□□□□□□□□□□□□□

```
interface A {
  fun foo() { print("A") }
  fun bar()
}

interface B {
  fun foo() { print("B") }
  fun bar() { print("bar") }
}

class C : A {
  override fun bar() { print("bar") }
}

class D : A, B {
  override fun foo() {
    super<A>.foo()
    super<B>.foo()
  }
}
```

□□□□□□□ *A* □ *B* □□□□□□□ *foo()* □ *bar()*□ □□□□□□□ *foo()*, □□□□ *B* □□□ *bar()* (*bar()* □ *A* □□□□□□□ abstract□ □□□□□□□ □□□□□□ abstract□□□□ *C* □□□□□□□ *A* □□□□□□□□□□□□□□ *bar()* □ □□□□□□□□□□□ *D* □□□□□□□ *bar()*□□□□□□□□ *A* □ *B*□□□□□□□ □□□ *B* □ *bar()* □□□□ □□□□□□□□□□□□□□□ *foo()*□□□□□□□□□□ *D* □□□□□□□□□□□□□□ *D* □□□ *foo()*□

□□□□□

□□□□□□□□□□□□□□□□□setter□□□□□□_visibility modifiers_□□□□□ □getter □□□□□□□□□□□□□□ □Kotlin□□□□□□□□□□□□ `private` □ `protected` □ `internal` □ `public` □ □□□□□□□□□□□□□□□□□□□ `public` □

□□□□□□□□□□□□□□□

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
// file name: example.kt
package foo

fun baz() {}
class Bar {}
```

— □□□□□□□□□□□□□□□□□□□□□□□□□ `public` □□□□□□□□□□□□□ □□□□;
— □□□□□ `private` □□□□□□□□□□□□□□□
— □□□□□ `internal` □□□□□□□□□□□□□□□
— `protected` □□□□□□□□□□

□□:

```
// file name: example.kt
package foo

private fun foo() {} // visible inside example.kt

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in example.kt

internal val baz = 6    // visible inside the same module
```

□□□□

□□□□□□□□□

— `private` □□□□□□□□□□□□□□(□□□□□□□).
— `protected` — □ `private` □□+□□□□□□□
— `internal` — □□□□□□ *inside this module* □□□□□□□□ `internal` □□□□□;
— `public` — □□□□□□□□□□□□□□ `public` □□□

□□ □□Java□□:□□□□□□□Kotlin□□□□private□□□

If you override a `protected` member and do not specify the visibility explicitly, the overriding member will also have `protected` visibility.

□□:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4  // public by default

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a is not visible
    // b, c and d are visible
    // Nested and e are visible

    override val b = 5   // 'b' is protected
}

class Unrelated(o: Outer) {
    // o.a, o.b are not visible
    // o.c and o.d are visible (same module)
    // Outer.Nested is not visible, and Nested::e is not visible either
}
```

□□□□

□□□□□□□□□□□□□□□□,□□□□□□□(□□□□□□□□□ □*□□□□*{:.keyword} keyword)□

```
class C private constructor(a: Int) { ... }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□`public`□□□□□ □□□□□□□□□□□□□□□□□(□□□□□□□□□□□□ □□□□□□□□).

□□□□

□□□□□□□□□□□□□□□□□□□

Modules

The `internal` visibility modifier means that the member is visible with the same module. More specifically, a module is a set of Kotlin files compiled together:

— an IntelliJ IDEA module;

— a Maven or Gradle project;

— a set of files compiled with one invocation of theAnt task.

□□

Kotlin□c#□Gosu□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□_extensions_□Kotlin□□ _extension functions_ □ *extension properties*.

□□□□

□□□□□□□□□□□□□□□□□□□*□□□□□*□□□□□□□□□□□□□□□□□□□□□ □□□ `MutableList<Int>` □□□□`swap` □□□

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
  val tmp = this[index1] // 'this' corresponds to the list
  this[index1] = this[index2]
  this[index2] = tmp
}
```

□□this□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□`MutableList<Int>` :

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'
```

□□□□□□□□□□□□`MutableList<T>` □□□□□□□□□□□

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
  val tmp = this[index1] // 'this' corresponds to the list
  this[index1] = this[index2]
  this[index2] = tmp
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□Generic functions.

□□□□□□□

□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ This means that the extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result of evaluating that expression at runtime. For example:

```
open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())
```

This example will print “c”, because the extension function being called depends only on the declared type of the parameter `c`, which is the `C` class.

If a class has a member function, and an extension function is defined which has the same receiver type, the same name and is applicable to given arguments, the **member always wins**. For example:

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }
```

□□□□□□C□□□c□c.foo()□□□□□”member”□□□□”extension”.

However, it’s perfectly OK for extension functions to overload member functions which have the same name but a different signature:

```
class C {
    fun foo() { println("member") }
}

fun C.foo(i: Int) { println("extension") }
```

The call to `C().foo(1)` will print “extension”.

Nullable□□□

□□ □□□□□□□null□□□□□□□□□□this == null□□□□□□□ □□□□□□null□□□□□□Kotlin□□toString()□□□□□□□□□□□□□□□□□□□□

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // after the null check, 'this' is autocast to a non-null type, so the toString() below
    // resolves to the member function of the Any class
    return toString()
}
```

□□□□

□□□□□□Kotlin□□□□□□

```
val <T> List<T>.lastIndex: Int
  get() = size - 1
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□[backing field]. □□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□ getters/setters.

□□:

```
val Foo.bar = 1 // error: initializers are not allowed for extension properties
```

□□□□□□□□

□□□□□□□□□[□□□□] □□□□□□□□□□□□ □□□□□□□

```
class MyClass {
  companion object { }  // will be called "Companion"
}

fun MyClass.Companion.foo() {
  // ...
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
MyClass.foo()
```

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□

```
package foo.bar

fun Baz.goo() { ... }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□

```
package com.example.usage

import foo.bar.goo // importing all extensions by name "goo"
                   // or
import foo.bar.*   // importing everything from "foo.bar"

fun usage(baz: Baz) {
  baz.goo()
)
```

□□□□□□Imports

Declaring Extensions as Members

Inside a class, you can declare extensions for another class. Inside such an extension, there are multiple *implicit receivers* - objects members of which can be accessed without a qualifier. The instance of the class in which the extension is declared is called *dispatch receiver*, and the instance of the receiver type of the extension method is called *extension receiver*.

```
class D {
    fun bar() { ... }
}

class C {
    fun baz() { ... }

    fun D.foo() {
        bar()   // calls D.bar
        baz()   // calls C.baz
    }

    fun caller(d: D) {
        d.foo()   // call the extension function
    }
}
```

In case of a name conflict between the members of the dispatch receiver and the extension receiver, the extension receiver takes precedence. To refer to the member of the dispatch receiver you can use the qualified this syntax.

```
class C {
    fun D.foo() {
        toString()         // calls D.toString()
        this@C.toString()  // calls C.toString()
    }
```

Extensions declared as members can be declared as `open` and overridden in subclasses. This means that the dispatch of such functions is virtual with regard to the dispatch receiver type, but static with regard to the extension receiver type.

```
open class D {
}

class D1 : D() {
}

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo()   // call the extension function
    }
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}

C().caller(D())   // prints "D.foo in C"
C1().caller(D())  // prints "D.foo in C1" - dispatch receiver is resolved virtually
C().caller(D1())  // prints "D.foo in C" - extension receiver is resolved statically
```

Motivation

□□

□Java□□□□□□□□□”*Utils”: `FileUtils`, `StringUtils` □□□□□ `java.util.Collections` □□□□□□□□□□□ □□□□Utils-classes□□□□□□□□□□□□□□□□

```
// Java
Collections.swap(list, Collections.binarySearch(list, Collections.max(otherList)),
Collections.max(list))
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
// Java
swap(list, binarySearch(list, max(otherList)), max(list))
```

□□□□□□□□□□□□□□□□□□IDE□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
// Java
list.swap(list.binarySearch(otherList.max()), list.max())
```

□□□□□□□□□□List□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□Kotlin□□□□*□□□* □□□`data`□

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

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

— `equals()` / `hashCode()` □□

— `toString()` □□□ `"User(name=John, age=42)"`□

— [componentN() functions](#) □□□□□□□□□□□□□□□□□□□

— `copy()` □□□□□□□□

□□□

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfil the following requirements:

— The primary constructor needs to have at least one parameter;

— All primary constructor parameters need to be marked as `val` or `var`;

— Data classes cannot be abstract, open, sealed or inner;

— Data classes may not extend other classes (but may implement interfaces).

> □JVM□□□ (□□[Constructors](#)).
>
> ```
> data class User(val name: String = "", val age: Int = 0)
> ```

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□`copy()`□□□□□□□□□□□□□□□□`User`□□□□□□□□□□□□□□□□□□□

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

□□□□□□□

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

□□□□□□□□□

_□□□□_□□□□□□□□□[□□□]□

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age"
```

[illegible]

[illegible]`Pair`[illegible]`Triple`[illegible]

□□

□Java□□□Kotlin□□□□□□□□□□□□□

```
class Box<T>(t: T) {
  var value = t
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val box: Box<Int> = Box<Int>(1)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val box = Box(1) // 1 has type Int, so the compiler figures out that we are talking about
Box<Int>
```

□□Variance

Java□□□□□□□□□□□□□□□□□wildcards□□□(□□Java Generics FAQ)□ □□Kotlin□□□□□□□□□□□□□□□□□□□□□□□declaration-site variance□□□□□□□

□□□□□□□□□Java□□□□□□wildcards□□□□□□□□□□□Effective Java, Item 28: *Use bounded wildcards to increase API flexibility*□□□□□□□□□□ □□□Java□□□□□□□□□□□`List<String>`□□□`List<Object>`□□□□□ □□□□□□□List□□□□□□□□ □□Java□□□□□□□□□□□□□□□□□□□□□□

```
// Java
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // !!! The cause of the upcoming problem sits here. Java prohibits
this!
objs.add(1); // Here we put an Integer into a list of Strings
String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String
```

□□□Java□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□`Collection`□□□□`addAll()` □□□□□□□□□□□□□□□□□□□□□□□
□□

```
// Java
interface Collection<E> ... {
  void addAll(Collection<E> items);
}
```

□□□□□□□□□□□□□□□□□□□□

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
  to.addAll(from); // !!! Would not compile with the naive declaration of addAll:
                   //       Collection<String> is not a subtype of Collection<Object>
}
```

[illegible]Effective Java, Item 25: *Prefer lists to arrays*[illegible]

[illegible] `addAll()` [illegible]

```
// Java
interface Collection<E> ... {
  void addAll(Collection<? extends E> items);
}
```

[illegible]**wildcard**[illegible] `? extends T` [illegible] `T` [illegible] `T` [illegible] T[illegible] `T` [illegible] `T` [illegible] `Collection<String>` [illegible] `Collection<? extends Object>` [illegible]**extends**[illegible]wildcard[illegible]**covariant**[illegible]

[illegible] `String` [illegible] `Object` [illegible]_[illegible]_[illegible] `Object` [illegible] `String` [illegible]Java[illegible] `List<? super String>` [illegible] `List<Object>` [illegible]

[illegible]**“[illegible]”**[illegible]**contravariance**[illegible]String[illegible] `List<? super String>` [illegible] `add(String)` [illegible] `set(int, String)`),[illegible] `List<T>` [illegible] `T` [illegible] `String` [illegible] `Object` [illegible]

Joshua Bloch[illegible]**Producers(**[illegible]**)**[illegible]**Consumers**[illegible]“*[illegible]*”[illegible]

PECS [illegible]roducer-Extends, Consumer-Super[illegible]

[illegible][illegible] `List<? extends Foo>` [illegible] `add()` [illegible] `set()` [illegible] `clear()` [illegible] `clear()` [illegible]

[illegible]

[illegible] `Source<T>` [illegible] `T` [illegible] `T` [illegible]

```
// Java
interface Source<T> {
  T nextT();
}
```

[illegible] `Source<Object>` [illegible] `Source<String>` [illegible]Java[illegible]

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!! Not allowed in Java
  // ...
}
```

□□□□□□□□□□□□□□□□□□□□□□`Source<? extends Object>`□□

□Kotlin□□`T` □□□□□□`Source<T>`□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□

```
abstract class Source<out T> {
  abstract fun nextT(): T
}

fun demo(strs: Source<String>) {
  val objects: Source<Any> = strs // This is OK, since T is an out-parameter
  // ...
}
```

□□□□□□□□□`C` □□□□`T` □□□□□□□□□□□□□□□□□□□□□□`C` □□□□□-□□□□□□□□`C<Base>` □□□□□□□□`C<Derived>` □□□□

□□□□□□□□`C` □□□`T` □□□□□□□□`T` □□□□□□□□□□□□□□ □□□□□□`C` □`T` □□□□□□□□□□□□`T` □□□□□

out□□□ □Java□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□**out**□Kotlin□□□□□□□□□□□□□□□**in**□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□`Comparable`□

```
abstract class Comparable<in T> {
  abstract fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
  x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
  // Thus, we can assign x to a variable of type Comparable<Double>
  val y: Comparable<Double> = x // OK!
}
```

□□□□**in**□**out**□□□□□□□□□□□□□□□□□□□□□C#□□□ □□□□□□□□□□□□□□□□□□□□□□□□□

[The Existential] **Transformation: Consumer in, Producer out!** □□□□□□□□□□□:)□

□□□□

□□□□□□□□□□□□

□□□□□□□T□*out*□□`T` □□□□□□□□□□ □□□□□□□□□ Array□

```
class Array<T>(val size: Int) {
  fun get(index: Int): T { /* ... */ }
  fun set(index: Int, value: T) { /* ... */ }
}
```

□□□□`T` □□□□□□□co- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun copy(from: Array<Any>, to: Array<Any>) {
  assert(from.size == to.size)
  for (i in from.indices)
    to[i] = from[i]
}
```

□□□□□□□□□□□copy□□□□□□□□□□□□□□□□□□□□□□□

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3)
copy(ints, any) // Error: expects (Array<Any>, Array<Any>)
```

□□□□□□□□□□□□□□□□`Array<T>` □□ `T` □□□□□□□□□□□ `Array<Int>` □ `Array<Any>` □□□□□□□□□□□□□□□□□□□copy□□□□□□□□□□□□□□□□□□□□□□□□□□String□ □□□□□□□□□□□ `Int` □□□□□ `ClassCastException` □□□□□□□□□

□□□□□□□□□□□□□ `copy()` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun copy(from: Array<out Any>, to: Array<Any>) {
 // ...
}
```

□□□□□□□□□□□□□ □ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `T` □□□□□□□□□□□□□ `get()` □□□□□□□□□□□□□□□□□□Java□□ `Array<? extends Object>` □ □□□□□□□□□□

□□□□□□□□□□**in**□□□□□□□

```
fun fill(dest: Array<in String>, value: String) {
  // ...
}
```

`Array<in String>` □□□Java□□ `Array<? super String>`, □□□□□□□ `fill()` □□□□□□ `CharSequence` □□□□□□ `Object` □□□

□-□□ Star-Projections

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□*out*-□□

Kotlin□□□□□□□□□□□□□□□□

- For `Foo<out T>`, where `T` is a covariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>`. It means that when the `T` is unknown you can safely *read* values of `TUpper` from `Foo<*>`.
- For `Foo<in T>`, where `T` is a contravariant type parameter, `Foo<*>` is equivalent to `Foo<in Nothing>`. It means there is nothing you can *write* to `Foo<*>` in a safe way when `T` is unknown.
- For `Foo<T>`, where `T` is an invariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>` for reading values and to `Foo<in Nothing>` for writing values.

If a generic type has several type parameters each of them can be projected independently. For example, if the type is declared as `interface Function<in T, out U>` we can imagine the following star-projections:

- `Function<*, String>` means `Function<in Nothing, String>`;
- `Function<Int, *>` means `Function<Int, out Any?>`;
- `Function<*, *>` means `Function<in Nothing, out Any?>`.

[illegible][illegible]Java[illegible]raw[illegible]raw[illegible]

[illegible]

[illegible]

```
fun <T> singletonList(item: T): List<T> {
  // ...
}

fun <T> T.basicToString() : String {  // extension function
  // ...
}
```

To call a generic function, specify the type arguments at the call site**after** the name of the function:

```
val l = singletonList<Int>(1)
```

[illegible]

[illegible]

[illegible]

[illegible]Java[illegible]*extends*[illegible]

```
fun <T : Comparable<T>> sort(list: List<T>) {
  // ...
}
```

[illegible] `T` [illegible] `Comparable<T>` [illegible]

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype of
Comparable<HashMap<Int, String>>
```

[illegible] `Any?` [illegible] [illegible]**where**-[illegible]:

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T>
    where T : Comparable,
          T : Cloneable {
  return list.filter { it > threshold }.map { it.clone() }
}
```

□□□

□□□□□□□□□□□□□□□

```
class Outer {
  private val bar: Int = 1
  class Nested {
    fun foo() = 2
  }
}

val demo = Outer.Nested().foo() // == 2
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□“inner” □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
class Outer {
  private val bar: Int = 1
  inner class Inner {
    fun foo() = bar
  }
}

val demo = Outer().Inner().foo() // == 1
```

□□this-expressions.html□“this”□□□□□□□□□□□□□□□□□□□“this”□□□□□□□□

Anonymous inner classes

Anonymous inner class instances are created using an object expression:

```
window.addMouseListener(object: MouseAdapter() {
  override fun mouseClicked(e: MouseEvent) {
    // ...
  }

  override fun mouseEntered(e: MouseEvent) {
    // ...
  }
})
```

If the object is an instance of a functional Java interface (i.e. a Java interface with a single abstract method), you can create it using a lambda expression prefixed with the type of the interface:

```
val listener = ActionListener { println("clicked") }
```

□□□

□□□□□□□□□□□□□□□□□□□□

```
enum class Direction {
  NORTH, SOUTH, WEST, EAST
}
```

□□□□□□□NORTH□SOUTH……□□□□□□□□□□□□□□□“,”□□□

□□□

□□□□□□□RED□GREEN……□□□□□□□□□□□□□□□□□□□

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
enum class ProtocolState {
  WAITING {
    override fun signal() = TALKING
  },

  TALKING {
    override fun signal() = WAITING
  };

  abstract fun signal(): ProtocolState
}
```

with their corresponding methods, as well as overriding base methods. Note that if the enum class defines any members, you need to separate the enum constant definitions from the member definitions with a semicolon, just like in Java.

Working with Enum Constants

Just like in Java, enum classes in Kotlin have synthetic methods allowing to list the defined enum constants and to get an enum constant by its name. The signatures of these methods are as follows (assuming the name of the enum class is `EnumClass`):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

[illegible] `valueOf()` [illegible] `IllegalArgumentException`[illegible]

[illegible]

```
val name: String
val ordinal: Int
```

[illegible] [Comparable] [illegible]

□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Java□□□□□*□□□□*□ □Kotlin□□□□□□□□□□*□□ □□□*□*□□□□*□

□□□□□

□□□□□□□□□□□□□□□□□□□□

```
window.addMouseListener(object : MouseAdapter() {
  override fun mouseClicked(e: MouseEvent) {
    // ...
  }

  override fun mouseEntered(e: MouseEvent) {
    // ...
  }
})
```

□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□

```
open class A(x: Int) {
  public open val y: Int = x
}

interface B {...}

val ab: A = object : A(1), B {
  override val y = 15
}
```

□□□□□□□□□□□□□□□□□□□□□□□□

```
val adHoc = object {
  var x: Int = 0
  var y: Int = 0
}
print(adHoc.x + adHoc.y)
```

□□Java□□□□□□□□□□□□□□□□□□□□ □□□Java□□□□□□□□final□□□□□

```
fun countClicks(window: JComponent) {
  var clickCount = 0
  var enterCount = 0

  window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
      clickCount++
    }

    override fun mouseEntered(e: MouseEvent) {
      enterCount++
    }
  })
  // ...
}
```

□□□□

□□□□□□□□□□□□□□□□□□Kotilin□□Scala□□□□□□□□□□□□□□□□□

```
object DataProviderManager {
  fun registerDataProvider(provider: DataProvider) {
    // ...
  }

  val allDataProviders: Collection<DataProvider>
    get() = // ...
}
```

□□□□*□□□□*□□□□□□object□□□□□□□□□□□□□□□□□□□_□□□_□ □□□
□□

```
object DefaultListener : MouseAdapter() {
  override fun mouseClicked(e: MouseEvent) {
    // ...
  }

  override fun mouseEntered(e: MouseEvent) {
    // ...
  }
}
```

NOTE: □□□

□□□□

□□□□□□□□□□□□□□□□companion□□□□□□

```
class MyClass {
  companion object Factory {
    fun create(): MyClass = MyClass()
  }
}
```

[illegible]

```
val instance = MyClass.create()
```

[illegible] `companion` [illegible]

```
class MyClass {
  companion object {
  }
}

val x = MyClass.Companion
```

[illegible] [illegible]

```
interface Factory<T> {
  fun create(): T
}


class MyClass {
  companion object : Factory<MyClass> {
    override fun create(): MyClass = MyClass()
  }
}
```

[illegible]JVM[illegible] `@JvmStatic` [illegible] [illegible]Java interoperability [illegible] [illegible]

[illegible]

[illegible]

— [illegible]

— [illegible],[illegible]**lazily**[illegible]

— a companion object is initialized when the corresponding class is loaded (resolved), matching the semantics of a Java static initializer

□□

□□□

□□□□□□□□□□□□□□□□. Kotlin□□□□□□□ □□□ `Derived` □□□□□□□□ `Base` □□□□□□□□□□□□□□□□□□□□□□□□□□

```
interface Base {
  fun print()
}

class BaseImpl(val x: Int) : Base {
  override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
  val b = BaseImpl(10)
  Derived(b).print() // prints 10
}
```

□□□ `Derived` □□ by-□□□□ `b` □□□ □□□ `Derived` □□□□□□□ □□□□□□□□□□□□□□□□□ `b` □ `base` □□□

□□□□

□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

— □□□□□lazy properties□: □□□□□□□□□□□□□□□□□□

— □□□□□□observable properties□: □□□□□□□□□□□□□□□□□□□□

— □□□□□□□□□□□map□□□□□□□□□□□□□□□□□□

□□□□□□(□□□□)□□□Kotlin □□*□□□□*:

```
class Example {
  var p: String by Delegate()
}
```

□□□: `val/var <property name>: <Type> by <expression>`.□by□□□□□□□□*□□*, □□ `get()` (□ `set()`) □□□□ □□□□□□□□ `getValue()` □ `setValue()` □□□ □□□□□□□□□□□□□□□□□□□□□□□`getValue()` □□□□ `setValue()` — □□ var's□□ □□:

```
class Delegate {
  operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
    return "$thisRef, thank you for delegating '${property.name}' to me!"
  }

  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
    println("$value has been assigned to '${property.name} in $thisRef.'")
  }
}
```

□□□□□□□`Delegate` □□□□□ `p` , `Delegate` □□ `getValue()` □□□□, □□□□□□□□□□□□ `p` □□□□□,□□□□□□□ `p` □□□□ □□ (□□□□□□□□□□□□). □□□□□:

```
val e = Example()
println(e.p)
```

□□□□□

Example@33a17727, thank you for delegating ‘p’ to me!

□□□□□□□□ `p` □□, `setValue()` □□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
e.p = "NEW"
```

□□□□□

NEW has been assigned to ‘p’ in Example@33a17727.

□□□□□□

□□□□□□□□□□□□□□□

□□□□ □□ □□ (□ val), □□□□□□□□□□□ getValue □□□□□□□□□□:

— □□□ — □□□_□□□□□_□□□□□□□□□□□(□□□□□□□□□□□□□□□□□),
— □□□□ — □□□□ KProperty<*> □□□□□□□□□,

□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ □□ □□ (□ var), □□□□□□□□□□□□□□□ setValue □□□□□□□□:

— □□□ — □ getValue(),
— □□□□ — □ getValue(),
— □□□ — □□□□□□□□□□□□□□□□□□

getValue() □/□ setValue() □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□ □□ □□□□□□□□ operator □□□□□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□factory□□□□

□□□□ Lazy

□□ lazy() □□□□ lambda □□□□□□□□□□□□□□□□□□□□□ Lazy<T> □□□: □□□□□ get() □□□□□□□□□□□□□□ lazy() □ lamda□□□□□□□□, □□□ get() □□□□□□□□□□□□□□

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

□□□□□□lazy□□□□□□□□□□synchronized□ □: □□□ □□□□□□□□□□□□□ LazyThreadSafetyMode.PUBLICATION □□□□□□□□□ lazy() □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□ LazyThreadSafetyMode.NONE □□, □□□□□□□□□□□□□□□□□□□□□

□□□□□ Observable

Delegates.observable() □□□□□□□□□□□□handler□ □□ handler □□□□□□□□□□□□□□□□□ (□□□□□□□). □□□□□□:□□□□□ □□□□□□□□□□□□□

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

□□□□□□□□

```
<no name> -> first
first -> second
```

□□□□□□□□□□□□“□□”□□□□□□□□□□□`vetoable()` □□ `observable()`. □□□□ `vetoable` □handler□□□□□□□□□□_□□_□□

□□□□□□ Map □

□□□□□□□□□□□□map□□□□□□□□□ □□□□□□□□□JSON□□□□□□□“□□”□□□□□□□□□ □□□□□□□□□□□□□□□map□□□□□□□□□□□□□□□□□

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

□□□□□□□□□□□□□□□□□□□□map□□□

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

□□□□□□□map□□□ (□□string□□□key□□□□□□□□□):

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

□□ *var*□□□□□□□□□□□□□`Map` □□ `MutableMap` □□□□

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

□□□Lambdas□□□

□□

□□□□

□Kotlin□□□□□□□□□□□□fun

```
fun double(x: Int): Int {
}
```

□□□□

□□□□□□□□□□□□

```
val result = double(2)
```

□□□□□□□□□□□□

```
Sample().foo() // create instance of class Sample and calls foo
```

□□□□

□□□□□□□□□□□□□□

— □□□□□□□□ □□□□□□
— □□□□□□□□□
— They are marked with the `infix` keyword

```
// Define extension to Int
infix fun Int.shl(x: Int): Int {
...
}

// call extension function using infix notation

1 shl 2

// is the same as

1.shl(2)
```

□□

□□□□□□□Pascal□□□□□*name*: *type*□□□□□□□□□□□□□□□□□□□□□

```
fun powerOf(number: Int, exponent: Int) {
...
}
```

□□□□(□□□□)

□□□□□□□□,□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size()) {
...
}
```

□□□□□□□□* * = * *□□□□□

Overriding methods always use the same default parameter values as the base method. When overriding a method with default parameters values, the default parameter values must be omitted from the signature:

```
open class A {
    open fun foo(i: Int = 10) { ... }
}

class B : A() {
    override fun foo(i: Int) { ... }  // no default value allowed
}
```

□□□□

□□□

□□□□□□□:

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
...
}
```

□□□□□□□□□□□□□□□□□

```
reformat(str)
```

□□□□□□□□□□□□□□□□□

```
reformat(str, true, true, false, '_')
```

□□□□□□□□□□□□□□□□□□□□□□□

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
  )
```

□□□□□□□□□□□□□

```
reformat(str, wordSeparator = '_')
```

Note that the named argument syntax cannot be used when calling Java functions, because Java bytecode does not always preserve names of function parameters.

□□**Unit**□□□

□□□□□□□□□□□□□□□□□□□□□□□□□ `Unit` □Unit □□□□□□□□ - Unit`□□□ □□□□□□□□□□

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

`Unit` □□□□□□□□□□□□□□□□□□□□□□□□

```
fun printHello(name: String?) {
    ...
}
```

□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□** =**□□□□□□

```
fun double(x: Int): Int = x * 2
```

□□□□□□□□□[□□]□,□□□□□□□□□

```
fun double(x: Int) = x * 2
```

□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□ `Unit` □ [□□□□□□]□□□□□□□ Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□(□□□□□)

□□□□□□□□□□□□□□□□□'vararg`□□□

```
fun <T> asList(vararg ts: T): List<T> {
  val result = ArrayList<T>()
  for (t in ts) // ts is an Array
    result.add(t)
  return result
}
```

□□□□□□□□□□□:

```
  val list = asList(1, 2, 3)
```

□□□□ `vararg` □□ `T` □□□□array `T` ,□□□□□□□□ `ts` □□□ `Array<out T>` □□□

□□□□□□□□□□□ `vararg` .If a `vararg` parameter is not the last one in the list, values for the following parameters can be passed using the named argument syntax, or, if the parameter has a function type, by passing a lambda outside parentheses.

□□□□□ `vararg` □□□□□□□□□□□□□□□□□□□ `asList(1, 2, 3)` □□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□**spread** □□□
□□□□□□□ `*` □□

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

□□□□□(□□□□)

□Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Java□C#□Scala□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□Kotlin□□□□□□□□ □□□□□□□□□□□□□□□□.

□□□□

Kotlin□□□□□□□,□□□□□□□□□□□□□□

```
fun dfs(graph: Graph) {
  fun dfs(current: Vertex, visited: Set<Vertex>) {
    if (!visited.add(current)) return
    for (v in current.neighbors)
      dfs(v, visited)
  }

  dfs(graph.vertices[0], HashSet())
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□the*visited*□□□□□.

```
fun dfs(graph: Graph) {
  val visited = HashSet<Vertex>()
  fun dfs(current: Vertex) {
    if (!visited.add(current)) return
    for (v in current.neighbors)
      dfs(v)
  }

  dfs(graph.vertices[0])
}
```

Member Functions

□□□□

□□□□□□□□□□,□□□□□□□□□□□

```
class Sample() {
  fun foo() { print("Foo") }
}
```

□□□□□□□□□

```
Sample().foo() // creates instance of class Sample and calls foo
```

□□□□□□□□□□□Classes □ Inheritance

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun <T> singletonList(item: T): List<T> {
  // ...
}
```

□□□□□□□□□□□□□□□Generics

□□□□

□□□□□□□here

□□□□

□□□□□□□their own section

□□□□□**Lambdas**□□□

□□□□□Lambdas□□□□□□□□□□their own section

Tail recursive functions

Kotlin supports a style of functional programming known astail recursion. This allows some algorithms that would normally be written using loops to instead be written using a recursive function, but without the risk of stack overflow. When a function is marked with the `tailrec` modifier and meets the required form the compiler optimises out the recursion, leaving behind a fast and efficient loop based version instead.

```
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

This code calculates the fixpoint of cosine, which is a mathematical constant. It simply calls Math.cos repeatedly starting at 1.0 until the result doesn't change any more, yielding a result of 0.7390851332151607. The resulting code is equivalent to this more traditional style:

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
```

To be eligible for the `tailrec` modifier, a function must call itself as the last operation it performs. You cannot use tail recursion when there is more code after the recursive call, and you cannot use it within try/catch/finally blocks. Currently tail recursion is only supported in the JVM backend.

□□□□□lambda□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□`lock()`□□□□□□□□□□□□□□□□□□□lock□□□□□□□□□□□□□□□□□□□□□□□

```
fun <T> lock(lock: Lock, body: () -> T): T {
  lock.lock()
  try {
    return body()
  }
  finally {
    lock.unlock()
  }
}
```

□□□□□□□□□□□□□`body` □□□□□□: `() -> T` □□body□□□□□□□□□□□□`T` □□□□□□□□ □□try□□□□□□□□□`lock` □□□□ □`lock()` □□□□□□□□□□□□□

□□□□□□□`lock()` □□□□□□□□□□□□□□□□□□□□□□(□□□□□□):

```
fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized)
```

□□□□□□□□□□□□□□□lambda □□□:

```
val result = lock(lock, { sharedResource.operation() })
```

lambda □□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□

- □□ lambda □□□□□□□□□□□□□□
- □□□□□□□□□□□□□□`->` □□□□□□□□□□□□
- □□□□ `->` □□ (□□□□□□□).

□Kotlin□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
lock (lock) {
  sharedResource.operation()
}
```

□□□□□□□□□□□`map()` (MapReduce):

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
  val result = arrayListOf<R>()
  for (item in this)
    result.add(transform(item))
  return result
}
```

□□□□□□□□□□□:

```
val doubled = ints.map { it -> it * 2 }
```

Note that the parentheses in a call can be omitted entirely if the lambda is the only argument to that call.

□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□ `->` □□□ `it` □□□

```
ints.map { it * 2 }
```

□□□□□□□□□LINQ-□□ □□□:

```
strings filter {it.length == 5} sortBy {it} map {it.toUpperCase()}
```

□□□□

□□□□□□□□□□□□□□□□□□□□

Lambda □□□□□□□□□

□□ lambda □□□□□□□□□□□□“□□□□□□”, □ □□□□□□□□□□ □□□□□□□□□□□□□□□□□□□

```
max(strings, { a, b -> a.length < b.length })
```

`max` □□□□□□□□□□, □□□□□ □□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□.□□□□□□□□□□□□□□□

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

□□□□

□□ □□□ `max` □□□□□□□□

```
fun <T> max(collection: Collection<T>, less: (T, T) -> Boolean): T? {
  var max: T? = null
  for (it in collection)
    if (max == null || less(max, it))
      max = it
  return max
}
```

□□ `less` □□□ `(T, T) -> Boolean` □□□□□, □□□□ `less` □□□□□ `T` □□□□□□□□□ `Boolean` □: □□□□□□□□□□□□
□ `True`.

□□□□□□□, `less` □□□□□□□□: □□□□ `T` □□□□□.

□□□□□□□□□□□, □□□□□□□□, □□□□□□□□□□□□□□□□

```
val compare: (x: T, y: T) -> Int = ...
```

Lambda□□□□□□

Lambda □□□□□□□□□□, □□□□□□□□□□□, □□□□□□□□:

```
val sum = { x: Int, y: Int -> x + y }
```

□□□Lambda□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□ `->` □□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

□□□□□□□□□□ lambda □□□□□□□□□□ □□Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `it` □□□□□□□

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□ lambda □□□□□□□□□ □□□□□□□□□ □□□□□□callSuffix.

□□□□

□□ lambda □□□□□□□□□□□□□□ □□□□□□□□□ □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□
□□□□□□:*□□□□*.

```
fun(x: Int, y: Int): Int = x + y
```

□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□:

```
fun(x: Int, y: Int): Int {
  return x + y
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
ints.filter(fun(item) = item > 0)
```

□□□`Unit`□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□ lambda □□□

□□□ lambda □□□□□□□□□□□□□non-local returns□□□□□□□□□□□□□□□*return*{□. keyword} □□ □□□□fun □□□□□□□□□□□ □□□□□□□ lambda □□□□□return *{: .keyword } □□□□□□□□□□ □□□□□□□*return*{: .keyword}□□□□□□□□□□□□□□□□

□□

□□ lambda □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□_□□_,□□□□□□□□□□□□□□□□java□□□□□□□□□□□□□□□□□□□□□□

```
var sum = 0
ints.filter { it > 0 }.forEach {
  sum += it
}
print(sum)
```

□□□□□□□□□□

kotlin□□□□□□□□□□□□*receiver□□* □□□□□□□□□□□. □□□□□□□□□□□□□ □□□□□□ □□□□□□□□□□□□□□□□□□□ □□ □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□Type-safe Groovy-style builders□□□□

□□□□□□□□□□□□□□□receiver□□□□□

```
sum : Int.(other: Int) -> Int
```

□□□□□□□□receiver□□□□□□□□□□□□□□□□□□□□□

```
1.sum(2)
```

□□□□□□□□□□□□□□□□□receiver□□□ □□□□□□□□receiver□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val sum = fun Int.(other: Int): Int = this + other
```

□receiver□□□□□□□□□□□□□□□□□Lamda□□□□□□□□□□receiver□□□□□□

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()  // create the receiver object
  html.init()        // pass the receiver object to the lambda
  return html
}


html {       // lambda with receiver begins here
    body()   // calling a method on the receiver object
}
```

□□□□ inline

□□ □□□□□□□□□□□□□□□□ □□□□(□□□□□□□□)□□□□□□□□□□ □□□

□□□□□□□□□□□□□ lambda □□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□ `lock()` □□□□□□□□□□□□□□□□□□□□□□□□ □□

```
lock(l) { foo() }
```

□□

```
l.lock()
try {
  foo()
}
finally {
  l.unlock()
}
```

□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□ `lock()` □□□□□ `inline` □□□:

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
  // ...
}
```

`inline` □□□□□□□□□□□□□□□□□□lambdas: □□□□□□□□□□ □□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□(□□□□□□□□)□□□□ □□□□□□□□□□□□□□□(megamorphic)□□□□□□□□

□□□□□noinline□

□□□□ □□□□□□□□□□□□□□□□□□□□□□□□lamdas □□□□□□□□□□□ `noinline` □□□□□□□□□:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
  // ...
}
```

□□□□□lambdas□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□ `noinline` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□[□□□□], □□□□□□□□□□□□□□□□□□□□ □□□□□□□□(□□□supress□□□□□□□□□□□□ □□□).

□□□□Non-local□□□□

□Kotlin□□□□□□□□□□□□□□□□□□ `return` □□□□□□□□□□□□□□□□□□ □□□□□□□□□□□lambda,□□□□□□□[□□],□□ □□□ lambda□ `return` □□□□□□□□□□□□□□□lambda□□□□□□□□□□□□:

```
fun foo() {
  ordinaryFunction {
     return // ERROR: can not make `foo` return here
  }
}
```

□□□□lambda□□□□□□□□□□□□□□return□□□□□□□□□□□□□□□□

```
fun foo() {
  inlineFunction {
    return // OK: the lambda is inlined
  }
}
```

□□□□□(□lambda□□□□□□□□□□□□□)□□□*non-local*□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun hasZeros(ints: List<Int>): Boolean {
  ints.forEach {
    if (it == 0) return true // returns from hasZeros
  }
  return false
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□lambdas□ □□□ non-local□□□□□□ □□□lambdas□□□□□□□□□□□□□□lambda□□□□ □ `crossinline` □□:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

break □ continue □□□□lambdas□□□□□, □□□□□□□□□□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p?.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□

```
myTree.findParentOfType(MyTreeNodeType::class.java)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
myTree.findParentOfType<MyTreeNodeType>()
```

□□□□□□□□□□□□*□□□□*, □□□□□□□□□□

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p?.parent
    }
    return p as T
}
```

□□□□□□□`reified` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□`!is` □ `as` □□□ □□□□ □□□□□□□□□□□□□□: `myTree.findParentOfType<MyTreeNodeType>()`.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
  println(membersOf<StringBuilder>().joinToString("\n"))
}
```

□□□□□ (□□□□□□□□□) □□□□□□□□. □□□□□□□□□□□□□ (□□□□□□□□□□□□□□□`Nothing`) □□□□□□□□□□□□

□□□□□□□spec document.

□□

□□□□

□□□□□□□_□□_□□□□□□□□□□□□□□□:

```
val (name, age) = person
```

□□□□□□_□□□□_□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□`name` □ `age`,□□□□□□□□□□□□□

```
println(name)
println(age)
```

□□□□□□□□□□□□□□□□□□□□□□

```
val name = person.component1()
val age = person.component2()
```

`component1()` □ `component2()` □□□ *principle of conventions* widely □Kotlin □□□□□□□□□ (□□□□□□`+` □ `*`, for-loops □) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□`component3()` □ `component4()`.

□□□□□ for-loops□□□

Destructuring declarations also work in for-loops: when you say

```
for ((a, b) in collection) { ... }
```

□□ `a` □ `b` □□□□ `component1()` □ `component2()` □□□□□collection□□□□□

□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □Kotlin□□□□□□□□□□□□□□□□□*data class*□□□□□□□□□□

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // computations

    return Result(result, status)
}

// Now, to use this function:
val (result, status) = function(...)
```

□□□□□□□□□ componentN() □□□□□□□□□□□□□□□

NOTE: □□□□□□□□ Pair □□□ function() □□ Pair<Int, Status>, □□□□□□□□□□□□□□□□□□□□

□: □□□□□□

□□□□□□□□□□□□□□□□□□□□□□

```
for ((key, value) in map) {
  // do something with the key and the value
}
```

□□□□□□□□□□□

— □□□□□□ iterator() □□□□□□□□□□□□□□□□□□□□□

— □□□□□□□□□□□□ component1() □ component2().

□□□□□□□□□□□□□□□:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

□□□□□□□□□□□□□□□□for-loops □□□□□(□□□□□□□□□□□□□□□□)□

□□

□□□□□□□□□Kotlin □□□□□□□□□□□□□□lists□sets□maps □□□□□□□□□□□□□□□□□□□□□□□ bug □□□□□□ API□

□□□□□□□□□□□□□*□□□* □□

Kotlin □ `List<out T>` □□□□□□□□□□□□□□`size`□`get`□□□□□□ Java □□□□□□□□`Collection<T>` □□□□□□ `Iterable<T>`□□□ list □□□□□□`MutableList<T>` □□□□□□□□□□□□□□`Set<out T>/MutableSet<T>` □ `Map<K, out V>/MutableMap<K, V>`□

□□□□□□□ list □ set □□□□□□□□□□

```kotlin
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)        // □□ "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)   // □□ "[1, 2, 3, 4]"
readOnlyView.clear()    // -> □□□□

val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

Kotlin □□□□□□□□□□□□□ list □ set□ □□□□□□□□□□□`listOf()`□`mutableListOf()`□`setOf()`□`mutableSetOf()`□□□ map □□□□□□□□□□□□□□□□□□□□□□□□□□`mapOf(a to b, c to d)`

□□□□□□`readOnlyView` □□□□□□□□□□□□□□□□□□□`numbers`□□□□□□□□□ list □□□□□□□□ □□□□ list □□

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

□□ `listOf` □□□□□□ array list □□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□`List<Rectangle>` □□□ `List<Shape>` □□ Rectangle □□□ Shape□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□

```kotlin
class Controller {
    private val _items = mutableListOf<String>()
    val items: List<String> get() = _items.toList()
}
```

□□ `toList` □□□□□□□□□□□□□□□□□□□□□ list □□□□□□□□□□

List □ set □□□□□□□□□□□□□□□□□□□

```
val items = listOf(1, 2, 3, 4)
items.first() == 1
items.last() == 4
items.filter { it % 2 == 0 }   // □□ [2, 4]

val rwList = mutableListOf(1, 2, 3)
rwList.requireNoNulls()       // returns [1, 2, 3]
if (rwList.none { it > 6 }) println("No items above 6")  // prints "No items above 6"
val item = rwList.firstOrNull()
```

..... □□□□□□□□□□□□□□□□□ sort□zip□fold□reduce □□□

Map □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(readWriteMap["foo"])  // prints "1"
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

□□

□□□□□□□“rangeTo”□□□□□□□□□□□□`..`□in□!in□□□ □□□□□□□□□□□□□□,□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
if (i in 1..10) { // equivalent of 1 <= i && i <= 10
  println(i)
}
```

□□□□□`IntRange`, `LongRange`, `CharRange`□□□□□□□□□□:□□□□□□□□ □□□□□□□□□□□□□□□□□Java□□□for□□,□□□□□□□□□□□□

```
for (i in 1..4) print(i) // prints "1234"

for (i in 4..1) print(i) // prints nothing
```

□□□□□□□□□□□□□□?□□□□□□□□□□□□□□□□□□`downTo()`□□

```
for (i in 4 downTo 1) print(i) // prints "4321"
```

□□□□□□□□□□□□□□□,□□□□□□□□□□□1□?□□`step()`□□□□□□

```
for (i in 1..4 step 2) print(i) // prints "13"

for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

□□□□□□□

Ranges implement a common interface in the library: `ClosedRange<T>`.

`ClosedRange<T>` □□□□□□□□□□□□□□,□□□□□□□□□□□□ □□□□□□□:‘start’□‘endInclusive’,□□□□□□□□□□□ □□□□□□`contains`,□□□in / *!in* {: .keyword }□□□□□

Integral type progressions (`IntProgression`, `LongProgression`, `CharProgression`) denote an arithmetic progression. Progressions are defined by the `first` element, the `last` element and a non-zero `increment`. The first element is `first`, subsequent elements are the previous element plus `increment`. The `last` element is always hit by iteration unless the progression is empty.

A progression is a subtype of `Iterable<N>`, where `N` is `Int`, `Long` or `Char` respectively, so it can be used in for-loops and functions like `map`, `filter`, etc. □□`Progression`□Java/JavaScript□□□□□□for□□□□:

```
for (int i = first; i != last; i += increment) {
  // ...
}
```

□□□□, `..`□□□□□□□□□□□□□□`ClosedRange`□□□□`Progression`□ For example, `IntRange` implements `ClosedRange<Int>` and extends `IntProgression`, thus all operations defined for `IntProgression` are available for `IntRange` as well. `downTo()`□`step()`□□□□□□□□`Progression`□

Progressions are constructed with the `fromClosedRange` function defined in their companion objects:

```
IntProgression.fromClosedRange(start, end, increment)
```

The `last` element of the progression is calculated to find maximum value not greater than the `end` value for positive `increment` or minimum value not less than the `end` value for negative `increment` such that `(last - first) % increment == 0`.

□□□□□□

rangeTo()

□□□ `rangeTo()` □□□□□□□□□□□□□□□(`*Range` □,□□:

```
class Int {
  //...
  operator fun rangeTo(other: Long): LongRange = LongRange(this, other)
  //...
  operator fun rangeTo(other: Int): IntRange = IntRange(this, other)
  //...
}
```

Floating point numbers (`Double` , `Float`) do not define their `rangeTo` operator, and the one provided by the standard library for generic `Comparable` types is used instead:

```
public operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

The range returned by this function cannot be used for iteration.

downTo()

`downTo()` □□□□□□□□□□□□□□□□□□,□□□□□□□□:

```
fun Long.downTo(other: Int): LongProgression {
  return LongProgression.fromClosedRange(this, other, -1.0)
}

fun Byte.downTo(other: Int): IntProgression {
  return IntProgression.fromClosedRange(this, other, -1)
}
```

reversed()

□□ `reversed()` □□□□□□□□□ `*Progression` □□□□,□□□□□□□□□□

```
fun IntProgression.reversed(): IntProgression {
  return IntProgression.fromClosedRange(last, first, -increment)
}
```

step()

`step()` □□□□□□□□ `*Progression` □□□□, □□□□□□□□□□□ `step` □(□□□□)□ □□,step□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun IntProgression.step(step: Int): IntProgression {
  if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
  return IntProgression.fromClosedRange(first, last, if (increment > 0) step else -step)
}

fun CharProgression.step(step: Int): CharProgression {
  if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
  return CharProgression.fromClosedRange(first, last, step)
}
```

Note that the `last` value of the returned progression may become different from the `last` value of the original progression in order to preserve the invariant `(last - first) % increment == 0`. Here is an example:

```
(1..12 step 2).last == 11  // progression with values [1, 3, 5, 7, 9, 11]
(1..12 step 3).last == 10  // progression with values [1, 4, 7, 10]
(1..12 step 4).last == 9   // progression with values [1, 5, 9]
```

[illegible]

is [illegible] !is[illegible]

[illegible] `is` [illegible] `!is` [illegible]

```
if (obj is String) {
  print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
  print("Not a String")
}
else {
  print(obj.length)
}
```

[illegible]

[illegible]Kotlin[illegible] `is` [illegible]

```
fun demo(x: Any) {
  if (x is String) {
    print(x.length) // x is automatically cast to String
  }
}
```

[illegible]

```
  if (x !is String) return
  print(x.length) // x is automatically cast to String
```

[illegible] `&&` [illegible] `||` [illegible]

```
  // x is automatically cast to string on the right-hand side of `||`
  if (x !is String || x.length == 0) return

  // x is automatically cast to string on the right-hand side of `&&`
  if (x is String && x.length > 0)
      print(x.length) // x is automatically cast to String
```

[illegible] when-expressions [illegible] while-loops [illegible]

```
when (x) {
  is Int -> print(x + 1)
  is String -> print(x.length + 1)
  is IntArray -> print(x.sum())
}
```

Note that smart casts do not work when the compiler cannot guarantee that the variable cannot change between the check and the usage. More specifically, smart casts are applicable according to the following rules:

- val local variables - always;
- val properties - if the property is private or internal or the check is performed in the same module where the property is declared. Smart casts aren't applicable to open properties or properties that have custom getters;
- var local variables - if the variable is not modified between the check and the usage and is not captured in a lambda that modifies it;
- var properties - never (because the variable can be modified at any time by other code).

"[illegible]"[illegible]

[illegible]*[illegible]*[illegible]Kotlin[illegible]as (see operator precedence)[illegible]

```
val x: String = y as String
```

[illegible]null[illegible][illegible] String [illegible] y [illegible]Jave[illegible]

```
val x: String? = y as String?
```

"[illegible]"[illegible]

[illegible]*[illegible]*[illegible]——as? [illegible]null[illegible]

```
val x: String? = y as? String
```

[illegible]as?[illegible] String [illegible]

This[illegible]

[illegible]this[illegible]:

- [illegible][illegible][illegible], this[illegible]
- [illegible][illegible][illegible][illegible], this[illegible].

[illegible] this [illegible] this [illegible]

this[illegible]

[illegible]this([illegible][illegible], [illegible][illegible], [illegible][illegible][illegible] `this@label` [illegible]label[illegible] on the scope this is meant to be from:

```
class A { // implicit label @A
  inner class B { // implicit label @B
    fun Int.foo() { // implicit label @foo
      val a = this@A // A's this
      val b = this@B // B's this

      val c = this // foo()'s receiver, an Int
      val c1 = this@foo // foo()'s receiver, an Int

      val funLit = lambda@ fun String.() {
        val d = this // funLit's receiver
      }


      val funLit2 = { s: String ->
        // foo()'s receiver, since enclosing lambda expression
        // doesn't have any receiver
        val d1 = this
      }
    }
  }
}
```

□□□

Kotlin□□□□□□□□□□□:

— □□□□□(□□□□□□□□□□□□□□)

— □□□□□ (`equals()`)

□□□□□

□□□□□□□`===`□□□□□□(□□□□□□`!==`). `a === b` □□□`a`□`b`□□□□□□□□□□□□true□

□□□□□

□□□□□□□`==`□□□□□□(□□□□□□`!=`). □□,`a == b`□□□□□□□□□

```
a?.equals(b) ?: (b === null)
```

□□□□□□`a`□□`null`□□□`equals(Any?)`□□□□□□□□`a`□`null`□□□b□□□□`null`□□□

□□□□`null`□□□□□□□□□□□□□□□□□□□□□□□□□`a == null`□□`a === null`□□□□□□□□□□□□□

□□□□□□

Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □ `+` □□ `*` □□□□□□[□□□]□□□□□□□□□□□□□□□□□□□□[□□□□] □[□□□□]□□□□□□□□□□□□□□□□□□□□□□□ Functions that overload operators need to be marked with the `operator` modifier.

□□

□□□□□□□□□□□□□□□□□

□□□□□

□□□	□□□
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

□□□□□□□□□□□□□□□□□□□ `+a` □□□□□□□□

— □□ `a` □□□, □□□ `T` □

— □□□□□□ `T` □□ `operator` □□□□□□□ `unaryPlus()` ,□□□□□□□□□□□□□□□

— □□□□□□□□□□□□□□□□□

— □□□□□□□□□□□□□□ `R` □□□□ `+a` □ `R` □□□

□□ □□□□□□□□□□□, □□□□[□□□□]□□□□□□□□□□□□

□□□	□□□
a++	a.inc() + □□□
a--	a.dec() + □□□

□□□□□□□□□□□□□□□□□

> ⚠ **inc()/dec() □□□□□□□□□.**
> “□□□□□□”□□□□□□□□□□□□□□□

□□□□□□□□*□□*□□□□□□□ `a++` :

— □□ `a` □□□, □□□ `T` □

— □□□□□□□ `T` □□ `operator` □□□□□□□□`inc()□

— □□□□□□□ `R` ,□□ `R` □ `T` □□□.

□□□□□□□□□□

— □ `a` □□□□ `a0` □,

— □ `a.inc()` □□□□□ `a` ,

— 和 a0 □□□□□□□□□.

a– □□□□□□□□□□□□□

□□□□□□□□ ++a 和 --a □□□□□□□□□□□□, □□□:

— 对 a.inc() □□□□ a ,

— □□□□□ a □□□□□□□□□

□□□□□□

□□□	□□□
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Expression	**Translated to**
a in b	b.contains(a)
a !in b	!b.contains(a)

in 和 !in □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□	□□□
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

□□□□□□□□ get set □□□

□□	□□□
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

□□□□□□□□□□□□□□□□ invoke □□□

□□□	□□□
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)

□□□ a+= b□□□□□□□□□□□□□□□:

— □□□□□□□□□□□

— □□□□□□□□□□ ([plus()] [plusAssign()])□□□, □□□□□□□□□.

— □□□□□□□□□ Unit □□, □□□□□□□□

— □□ a.plusAssign(b) *□□□□□□ a = a + b □□ (□□□□□□□□□: a + b □□□□ a □□□□).

□□: assignments□Kotlin□□□□□□.

□□□	□□□
a == b	a?.equals(b) ?: b === null
a != b	!(a?.equals(b) ?: b === null)

□□: === □ !== (□□□□)□□□□, □□□□□□□□□□

The == □□□□□□□: □□□□□□□□□□□□□□□□□□□□ null □□□□ null == null □ true □

□□	□□□
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

□□□□□□□□□ compareTo □□□□□□□□□□□□□□□ Int □

□□□□□□□□□□

□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□

[illegible]

[illegible]Nullable[illegible]Non-Null[illegible]

Kotlin [illegible][illegible]

[illegible] Java [illegible] Java [illegible] `NullPointerException` [illegible] `NPE` [illegible]

Kotlin [illegible] `NullPointerException` [illegible] `NPE` [illegible]

— [illegible] `throw NullPointerException()`
— Usage of the `!!` operator that is described below
— [illegible] Java [illegible]
— [illegible] ([illegible] `this` [illegible])

[illegible] Kotlin [illegible] null [illegible]nullable references[illegible] [illegible]non-null references[illegible] [illegible] `String` [illegible] null[illegible]

```
var a: String = "abc"
a = null // [illegible]
```

[illegible] `null` [illegible] `String?` [illegible]

```
var b: String? = "abc"
b = null // ok
```

[illegible]/[illegible] `a` [illegible]/[illegible] `a` [illegible] `NPE` [illegible]

```
val l = a.length
```

[illegible] `b` [illegible]

```
val l = b.length // [illegible] b [illegible] null
```

[illegible]

[illegible] null

[illegible] `b` [illegible] null[illegible]

```
val l = if (b != null) b.length else -1
```

[illegible] if [illegible] `length`
[illegible]

```kotlin
if (b != null && b.length > 0)
  print("String of length ${b.length}")
else
  print("Empty string")
```

□□□□□□□□□□□□ b □□□□□□i.e. a local variable which is not modified between the check and the usage or a member val which has a backing field and is not overridable□□□□□□□ □□□□□□□□□□□□□□□

□□□□□□

□□□□□□□□□□□□□□□□□ ?. □

```kotlin
b?.length
```

□□ b □□□□□□□□□ b.length □□□□□null□□□□□□□□□□□□Int?.

□□□□□□□□□□□□□□□□□□□□□□Bob□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Bob □□□□□□□□□□□□□□□□□□□□□□□

```kotlin
bob?.department?.head?.name
```

□□□□□□□□□□□□□□□□□□□□□□□□□null□

To perform a certain operation only for non-null values, you can use the safe call operator together with let:

```kotlin
val listWithNulls: List<String?> = listOf("A", null)
for (item in listWithNulls) {
    item?.let { println(it) } // prints A and ignores null
}
```

Elvis □□□

□□□□□□□□□□□□□ r □□□□□□ □□□ r □□□□□□□□□□□□□□□□□□□□

```kotlin
val l: Int = if (b != null) b.length else -1
```

□□□□□ if-□□□, □□□□ Elvis □□□□□□, □□ ?: :

```kotlin
val l = b?.length ?: -1
```

□□ ?: □□□□□□□□□□□□ elvis □□□□□□□□□□□□□, □□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□

□□, □□ throw □ return □ Kotlin □□□□□□□□□□□□□□□□ Elvis □□□□□□□□□□□□□□□□□□□□□□

```kotlin
fun foo(node: Node): String? {
  val parent = node.getParent() ?: return null
  val name = node.getName() ?: throw IllegalArgumentException("name expected")
  // ...
}
```

!! □□□

□□□□□□□□□□□ NPE □□□□□□□□□□`b!!` □□□□□□□□□□□□□`b` □□□□□□□□□□□□□□`String` □□□□□`b` □□□□□□□□□`NPE` □□□

```kotlin
val l = b!!.length
```

□□□□□□□□□□□`NPE` □□□□□□□□□but you have to ask for it explicitly, and it does not appear out of the blue.

□□□□

□□□□□□□□□□□□□□□`ClassCastException` □ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□null□

```kotlin
val aInt: Int? = a as? Int
```

Collections of Nullable Type

If you have a collection of elements of a nullable type and want to filter non-null elements, you can do so by using `filterNotNull`.

```kotlin
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

□□

□□□

Kotlin□□□□□□□□ Throwable □□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ throw-expression □□□□□□□□

```
throw MyException("Hi There!")
```

□□ *try*{: .keyword }-expression □□□□□□□□

```
try {
  // some code
}
catch (e: SomeException) {
  // handler
}
finally {
  // optional finally block
}
```

□□□□□□□catch □□finally □□□□□□catch □ finally □□□□□□□□□□

Try□□□□□□

try □□□□□□□□□□□□□□□□□□□□

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

try-expression □□□□□try □ □□□□□□□□□□catch □□□□□□□□□□finally □□□□□□□□□□□□□□□□□

□□□□

Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□JDK□ StringBuilder □□□□□

```
Appendable append(CharSequence csq) throws IOException;
```

What does this signature say? It says that every time I append a string to something (a StringBuilder, some kind of a log, a console, etc.) I have to catch those IOExceptions. Why? Because it might be performing IO (Writer also implements Appendable)... So it results into this kind of code all over the place:

```
try {
  log.append(message)
}
catch (IOException e) {
  // Must be safe
}
```

[illegible]Effective Java,Item 65: *[illegible]*[illegible]

Bruce Eckel[illegible]Does Java need Checked Exceptions? [illegible]:

> [illegible] - [illegible]
> [illegible]

[illegible]

— Java’s checked exceptions were a mistake (Rod Waldhoff)

— The Trouble with Checked Exceptions (Anders Hejlsberg)

Java [illegible]

[illegible] Java Interoperability section [illegible]Java[illegible]

□□

□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□annotation □□□□□□□□□□□□

```
annotation class Fancy
```

Additional attributes of the annotation can be specified by annotating the annotation class with meta-annotations:

- @Target specifies the possible kinds of elements which can be annotated with the annotation (classes, functions, properties, expressions etc.);
- @Retention specifies whether the annotation is stored in the compiled class files and whether it’s visible through reflection at runtime (by default, both are true);
- @Repeatable allows using the same annotation on a single element multiple times;
- @MustBeDocumented specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
public annotation class Fancy
```

□□

```
@Fancy class Foo {
  @Fancy fun baz(@Fancy foo: Int): Int {
    return (@Fancy 1)
  }
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□constructor□□□□□□ □□□□□□□□□□□□

```
class Foo @Inject constructor(dependency: MyDependency) {
  // ...
}
```

□□□□□□□□□□□□□□

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

□□□□

□□□□□□□□□□□□□□

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

Allowed parameter types are:

— types that correspond to Java primitive types (Int, Long etc.);

— strings;

— classes (`Foo::class`);

— enums;

— other annotations;

— arrays of the types listed above.

If an annotation is used as a parameter of another annotation, its name is not prefixed with the @ character:

```
public annotation class ReplaceWith(val expression: String)

public annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

If you need to specify a class as an argument of an annotation, use a Kotlin class (KClass). The Kotlin compiler will automatically convert it to a Java class, so that the Java code will be able to see the annotations and arguments normally.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any?>)

@Ann(String::class, Int::class) class MyClass
```

Lambdas

□□□□□□□□lambda□□□□□□□□□□□□ lambda □□ `invoke()` □□ □□□Quasar□□□□□□□ □□□□□□□□□□□□□□□□□

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

Annotation Use-site Targets

When you're annotating a property or a primary constructor parameter, there are multiple Java elements which are generated from the corresponding Kotlin element, and therefore multiple possible locations for the annotation in the generated Java bytecode. To specify how exactly the annotation should be generated, use the following syntax:

```
class Example(@field:Ann val foo,    // annotate Java field
              @get:Ann val bar,      // annotate Java getter
              @param:Ann val quux)   // annotate Java constructor parameter
```

The same syntax can be used to annotate the entire file. To do this, put an annotation with the target `file` at the top level of a file, before the package directive or before all imports if the file is in the default package:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

If you have multiple annotations with the same target, you can avoid repeating the target by adding brackets after the target and putting all the annotations inside the brackets:

```
class Example {
    @set:[Inject VisibleForTesting]
    public var collaborator: Collaborator
}
```

The full list of supported use-site targets is:

— `file`
— `property` (annotations with this target are not visible to Java)
— `field`
— `get` (property getter)
— `set` (property setter)
— `receiver` (receiver parameter of an extension function or property)
— `param` (constructor parameter)
— `setparam` (property setter parameter)
— `delegate` (the field storing the delegate instance for a delegated property)

To annotate the receiver parameter of an extension function, use the following syntax:

```
fun @receiver:Fancy String.myExtension() { }
```

If you don't specify a use-site target, the target is chosen according to the `@Target` annotation of the annotation being used. If there are multiple applicable targets, the first applicable target from the following list is used:

— `param`

— `property`

— `field`

Jave□□

Java□□□□□□□□□Kotlin□

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
  // apply @Rule annotation to property getter
  @get:Rule val tempFolder = TemporaryFolder()

  @Test fun simple() {
    val f = tempFolder.newFile()
    assertEquals(42, getTheAnswer())
  }
}
```

□□□Java□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

□□□Jave□□□□□□□□□□□□□□’ value`□□;□□□□□□□□□□□□□□□□□

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

□□□Java□`value`□□□array□□□□Kotlin□□□□□`vararg`□□□□□

```java
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```kotlin
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

For other arguments that have an array type, you need to use `arrayOf` explicitly:

```java
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```kotlin
// Kotlin
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar")) class C
```

□□□□□□□□□Kotlin□□□□

```java
// Java
public @interface Ann {
    int value();
}
```

```kotlin
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□

> ⚠ □Java□□□□□□□□□□□□□□□□□□□□□□□□□ Jar □□(kotlin-reflect.jar).□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□,□□□□□□□□□□□□□□□□.

□□□

□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□:

```
val c = MyClass::class
```

□□□KClass□□.~~□□□□□KClass.properties □KClass.extensionProperties□□□□□□□□□□□□□□□□□~~

□□Kotlin□□□□□□□□Java□□□□□.□□Java interop section □□□□□

□□□□

□□□□□□□□□□□□□□□□□:

```
fun isOdd(x: Int) = x % 2 != 0
```

□□□□□□□□□□(`isOdd(5)`), □□□□□□□□□□□□□□□□□□. □□□ `::` □□□□□:

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // prints [1, 3]
```

□□ `::isOdd` □□□□□□□□□□ `(Int) -> Boolean`.

`::` can be used with overloaded functions when the expected type is known from the context. For example:

```
fun isOdd(x: Int) = x % 2 != 0
fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // refers to isOdd(x: Int)
```

Alternatively, you can provide the necessary context by storing the method reference in a variable with an explicitly specified type:

```
val predicate: (String) -> Boolean = ::isOdd   // refers to isOdd(x: String)
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□, □`String::toCharArray` □□□□`String`: `String.() -> CharArray` □□□□□□.

□□: □□□□

□□□□□□□□

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

```
fun length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // Prints "[a, abc]"
```

□□□□

□□□□□□□ :: □□□□□□□Kotlin□□□□□□□□□

```
var x = 1

fun main(args: Array<String>) {
    println(::x.get()) // prints "1"
    ::x.set(2)
    println(x)         // prints "2"
}
```

□□□ ::x □□□ KProperty<Int> □□□□□□□□,□□□□□ □□ get() □□□□□□□□□□□ name □□□□□□□□□□□□□ □□docs on the KProperty class.

□□□□□□□,□□ var y = 1 , ::y □□□□□□□KMutableProperty<Int>, □□□□ set() □□.

A property reference can be used where a function with no parameters is expected:

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length)) // prints [1, 2, 3]
```

To access a property that is a member of a class, we qualify it:

```
class A(val p: Int)

fun main(args: Array<String>) {
    val prop = A::p
    println(prop.get(A(1))) // prints "1"
}
```

□□□□□□□:

```
val String.lastChar: Char
  get() = this[length - 1]

fun main(args: Array<String>) {
  println(String::lastChar.get("abc")) // prints "c"
}
```

□Java□□□□

□ java □□□□□□□□□□□□□□□□□□□□□□ java □□ □□□□□(□ `kotlin.reflect.jvm` □)□ □□□□□□□□□□□□□□□ java getter □ □□□□□□□□□□

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main(args: Array<String>) {
    println(A::p.javaGetter) // prints "public final int A.getP()"
    println(A::p.javaField)  // prints "private final int A.p"
}
```

To get the Kotlin class corresponding to a Java class, use the `.kotlin` extension property:

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

□□□□□□□

□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□ `::` □□□□□□□□□.□□□□ □□□ □□□□□□□□□□□□□□□□ `Foo` :

```
class Foo

fun function(factory : () -> Foo) {
    val x : Foo = factory()
}
```

Using `::Foo` , the zero-argument constructor of the class Foo, □□□□□□□□□□□□□:

```
function(::Foo)
```

Type-Safe Builders

□□(builders)□□□*Groovy*□□□□□□□ □□□□□□□□□□□□□(semi-declarative)□□□□□□□□□□□□□□□□□□□XML□ □□UI□□, □□3D□□□□□□□□□□…

□□□□□Kotlin□□*□□□□*□□□□□□□Groovy□□□□□□□□□□□□□□

□□□□□Kotlin□□□*□□□□*□□□□

□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
import com.example.html.* // see declarations below

fun result(args: Array<String>) =
  html {
    head {
      title {+"XML encoding with Kotlin"}
    }
    body {
      h1 {+"XML encoding with Kotlin"}
      p  {+"this format can be used as an alternative markup to XML"}

      // an element with attributes and text content
      a(href = "http://kotlinlang.org") {+"Kotlin"}

      // mixed content
      p {
        +"This is some"
        b {+"mixed"}
        +"text. For more see the"
        a(href = "http://kotlinlang.org") {+"Kotlin"}
        +"project"
      }
      p {+"some text"}

      // content generated by
      p {
        for (arg in args)
          +arg
      }
    }
  }
```

□□□□□□□Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□

□□□□□□□Kotlin□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□HTML□□□□□ □□□□□□□□□□□ □□ `HTML` □□□□□□ `<html>` □□□□□□□□□ `<head>` □ `<body>` □ (□□□□□□□□.)

□□□□□□□□□□□□□□□□□□□□□□□□:

```
html {
 // ...
}
```

`html` □□□□□□□□□□□□□□□□[lambda □□□] □□□□□□□□□□

```
fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()
  html.init()
  return html
}
```

□□□□□□□□□□`init` □□□□□□□□□□□□ The type of the function is `HTML.() -> Unit`, which is a *function type with receiver*. This means that we need to pass an instance of type `HTML` (a *receiver*) to the function, and we can call members of that instance inside the function. The receiver can be accessed through the this keyword:

```
html {
  this.head { /* ... */ }
  this.body { /* ... */ }
}
```

(`head` □ `body` □□ `HTML` □□□□□□)

□□□□□□□□□□this□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

```
html {
  head { /* ... */ }
  body { /* ... */ }
}
```

□□□□□□□□□□□□□ □□□□□□□□□□□ `html` □□□□□□□□□□□□□□ `HTML` □□□□□□□□□□□□□□□□□□□□□□□□□□ `HTML` □□□□ □ `html` □□□□□□ `body()` □□□□□□□□□this□□□ □□□□□□□□□□□□

HTML□□□□□ `head` □ `body` □□□□□□□□ `html` □□□□□□□□□□□□□□□□□□□□□□□□html□children□□□□□□□□□

```
fun head(init: Head.() -> Unit) : Head {
  val head = Head()
  head.init()
  children.add(head)
  return head
}

fun body(init: Body.() -> Unit) : Body {
  val body = Body()
  body.init()
  children.add(body)
  return body
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `initTag` □

```
  protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
  }
```

□□□□□□□□□□□□:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

□□□□□□□□□□ `<head>` □ `<body>` □□.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
html {
  head {
    title {+"XML encoding with Kotlin"}
  }
  // ...
}
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `+` □□□ □□□□□□□□□□□□□□ `unaryPlus()` □□□□□ `unaryPlus()` □□□
□ `TagWithText` (`Title` □□□)□□□□□□

```
fun String.unaryPlus() {
  children.add(TextElement(this))
}
```

□□□□□ `+` □□□□□□□□□□ `TextElement` □□□□□□□□□□□children□□□□□□□□□□□□□□□□□□

□□□□□□□□ `com.example.html` □□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□

□com.example.html□□□□□

□□□□ `com.example.html` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□HTML□□ □□□□□□□□□□□□ □□□□□□lambda□□

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String? {
        val builder = StringBuilder()
        for (a in attributes.keys) {
            builder.append(" $a=\"${attributes[a]}\"")
        }
        return builder.toString()
    }


    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
```

```
    }
}

class HTML() : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head() : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title() : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body() : BodyTag("body")
class B() : BodyTag("b")
class P() : BodyTag("p")
class H1() : BodyTag("h1")

class A() : BodyTag("a") {
    public var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

□□□□

⚠ The dynamic type is not supported in code targeting the JVM

□□□□□□□,Kotlin□□□□□□□□□□□□□□□□ □□JavaScript,□□□□□□□dynamic□□□

```
val dyn: dynamic = ...
```

dynamic □□□□Kotlin□□□□:

- □□□□□□□□□□□□□□□□□□□□,
- □□□□□□□dynamic□□□□□□□□□□□□dynamic□□□□□□
- □□□□null□□□

dynamic □□□□□□□□dynamic□□□□□□□□□□□□□ □

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere
dyn.whatever(*arrayOf(1, 2, 3))
```

□JavaScript□□□□□□□□”as is”: dyn.whatever(1) □Kotlin□ dyn.whatever(1) □□□ JavaScript □□.

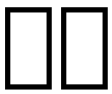
□□□□dynamic □□□□□□□□□□□□□□□□

```
dyn.foo().bar.baz()
```

□□□□□□□□ lambda □□□□□□□□□□dynamic □

```
dyn.foo {
  x -> x.bar() // x is dynamic
}
```

□□□□, □□.



Grammar

We are working on revamping the Grammar definitions and give it some style! Until then, please check the Grammar from the old site

□□□

□Kotlin□□□Java□□

Kotlin □□□□□□□□ java □□□□□□□□□□ Java □□□□□ kotlin □□□□□□ Kotlin □□□□□ □ Java □□□□□□□□□□□□□□□ kotlin □□□ Java □□□□□□

□□□□□ Java □□□□□□□□

```
import java.util.*

fun demo(source: List<Int>) {
  val list = ArrayList<Int>()
  // 'for'-loops work for Java collections:
  for (item in source)
    list.add(item)
  // Operator conventions work as well:
  for (i in 0..source.size() - 1)
    list[i] = source[i] // get and set are called
}
```

Getters □ Setters

□□□□□□getter/setter□□□□□□□□□□□□□□□(getter□□ `get` □□□□□□/setter□□□ `set` □□ □□□□□)□□□□Kotlin□□□□□□□□□□ □□□□□□□□□

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY      // call setFirstDayOfWeek()
    }
}
```

□□□□□□□□□□□□setter□□□□□□kotlin□□□□□□□□□□□□□□□

□□void□□□

□□□□Java□□□□void□□□□Kotlin□□□□□□(`Unit` □ □□□□□□□□□□□□□Kotlin□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□(□□ □□□ `Unit`)□

□Java□□□□Kotlin□□□□□□□□□□□□

□□Kotlin□□□□□Java□□□□□□□□:in, object, is, □□. □□□□Java□□□□□□□□Kotlin□□□,□□□□□□□□□□□□ □□□□(`)□□□□□□□

```
foo.`is`(bar)
```

Null□□□□□□□□

Java□□□□□□□□□□null□□□□□Kotlin□□□null□□□□□Java□□□□□□□□□□□□ □Kotlin□Java□□□□□□□□□□*platform types*.□□□□□Null□□□□□□□□□ □□□□□□□□□Java□□□□□□□□ (□□□□□)□

□□□□□□:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size() // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

□□□□□□□□□□□□□□□□□□Kotlin□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□kotlin□□□□null□□□□□□□□□□□

```
item.substring(1) // □□, □□item□□□□□□
```

□□□□□*□□□□*□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□kotlin□□□□□□□□□□□□□□□□□□□□□□□ □□□□□item □□□□□□□□□□□□□□□□□□□□□□□□□□□nullable□non-null□□□□□□□

```
val nullable: String? = item // □□□□□□□
val notNull: String = item // □□□□□□□□□□□
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Kotlin□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□kolin□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□

□□□□□□

□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□ □□□□□□IDE□□□□□□□□(□□□□□□□□□□□□□□)□□□□□□ □□□□□□□□□□□□□□□

- T! □□ "T □□ T? "
- (Mutable)Collection<T>! □□ "T□java□□□□□□□□□□□□□□□□□□□□□"
- Array<(out) T>! □□ "T (□ T □□□)□java□□□□□□□□□□□□"

□□□□□

Java types which have nullability annotations are represented not as platform types, but as actual nullable or non-null Kotlin types. The compiler supports several flavors of nullability annotations, including:

- JetBrains (@Nullable and @NotNull from the org.jetbrains.annotations package)

— Android (`com.android.annotations` and `android.support.annotations`)

— JSR-305 (`javax.annotation`)

— FindBugs (`edu.umd.cs.findbugs.annotations`)

— Eclipse (`org.eclipse.jdt.annotation`)

— Lombok (`lombok.NonNull`).

You can find the full list in the Kotlin compiler source code.

□□□□□

Kotlin□□□□□□□java□□□□□□□□□□□□as□is□□□□□□□□_□□_□□□□□kotlin□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□ java□ □□□□□□□kotlin□□□□□ □□□□□□

Java□□	**Kotlin□□**
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

□□□□□□□□□□□□□□□

Java□□	**Kotlin□□**
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.Void	kotlin.Nothing!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

□□□□□Kotlin□□□□□□□□□□□□□□□Java□□□□□□□□□□ □□□□□□Kotlin□□[`Kotlin`□□□

Java□□	**Kotlin□□□□**	**Kotlin□□□□**	□□□□□□□□
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Iterable<T>!

Java□□	Kotlin□□□□	Kotlin□□□	□□□□□
Collection<T>	Collection<T>	MutableCollection<T>	(Mutable)Collection<T>!
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set<T>!
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map.(Mutable)Entry<K, V>!

Java□□□□□□□□□□below□

Java□□	Kotlin□□
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

Kotlin□□Java□□

Kotlin□□□□Java□□□□□□□□Generics□□□□□java□□□□□□□□□□□□□□

- Java□□□□□□□□□□□□
  - `Foo<? extends Bar>` □□□ `Foo<out Bar!>!`
  - `Foo<? super Bar>` □□□ `Foo<in Bar!>!`
- Java□□□□□□□□□□□□□
  - `List` □□□ `List<*>!`, □□□ `List<out Any?>!`

□Java□□□Kotlin□□ □□□ `ArrayList<Integer>()` □ `ArrayList<Character>()` □□□□□□□□ □□□□□□□□□is-□□□□□□ Kotlin□□□□is-□□□□□□□□□□□□□:

```
if (a is List<Int>) // □□: □□□□□□□□□□Int□List
// but
if (a is List<*>) // □□□□□□list□□□□□□□
```

Java□□

□Java□□□Kotlin□□□□□□□□□□Kotlin□□□□□□ `Array<String>` □□□ `Array<Any>` □ □□□□□□□□□□□□□□□Kotlin□□□□□□□□□□□□□□□□□□□□□□□□□Kotlin□□□□□□ □□□Java□□□□□□□□□□□□□□□□□□□□□□platform types `Array<(out) String>!` □□

Java□□□□□□□□□□□□□□□□□□□□□□□/□□□□□□□□ □□Kotlin□□□□□□□□□□□□□□□□□□□□□□Java□□□□□□ □□□□□□□□□□□□□□□□□(specialized class)□□□□□□□(`IntArray`, `DoubleArray`, `CharArray` …)□ □□□□ `Array` □□□□□□□□java□□□□□□□□□□□□□□□□

□□□□□Java□□□□□□□□□□□□□□int□□□□□

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

□Kotlin□□□□□□□□□□□□□□:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // passes int[] to method
```

□□□□jvm□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
val array = arrayOf(1, 2, 3, 4)
array[x] = array[x] * 2 // □□□□□get() □ set()□□□
for (x in array) // □□□□□□□
  print(x)
```

□□□□□□□□□□□

```
for (i in array.indices) // □□□□□□□
  array[i] += 2
```

□□□in-□□□□□□□□□□

```
if (i in array.indices) { // □ (i >= 0 && i < array.size) □□
  print(array[i])
}
```

Java Varargs

Java□□□□□□□□□□□□□□□□□□□□

```
public class JavaArrayExample {

    public void removeIndices(int... indices) {
        // code here...
    }
}
```

□□□□□□□□□□□□□□`*` □□□ `IntArray` □

```
val javaObj = JavaArray()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

□□□□□□□null □□□□□□□□□□

□□□

□□Java□□□□□□□□□□□□□Kotlin□□□□ □□ `invoke()` □□□ □□□Java□□□□□□□□ □□□□□□(`infix call syntax`)□□□□□□□□

□□□□

□Kotlin□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□java□□□□□□□kotlin□□□□□□□□□□

```
fun render(list: List<*>, to: Appendable) {
  for (item in list)
    to.append(item.toString()) // Java□□□□□□□□□□IOException
}
```

□□□□

□java□□□□□□□kotlin□□□□□□[`java.lang.Object` □□□□□□□□□□□ `Any` □ □□ `Any` □□□□□□□□□□□□□□□□□□□□□□ □□ `toString()` , `hashCode()` □ `equals()` □ □□□□□□□[`java.lang.Object` □□□□□□□kotlin□□□□□□□□

wait()/notify()

Effective Java □69□□□□□□□□□□concurrency□□□□[`wait()` □ `notify()` □ □□□ `Any` □□□□□□□□□□ □□□□□□□□□□□□□□□□ □ `java.lang.Object` □

```
(foo as java.lang.Object).wait()
```

getClass()

□□□□□□□□□□□□□□□□□□□□□javaClass□□□□□□□□

```
val fooClass = foo.javaClass
```

□javaClass()□□□□java□□□□`Foo.class`□

```
val fooClass = javaClass<Foo>()
```

clone()

□□□ `clone()` □□□ `kotlin.Cloneable` □

```
class Example : Cloneable {
  override fun clone(): Any { ... }
}
```

□□□□ Effective Java,□11□: ////////

finalize()

□□□ `finalize()` , □□□□□□□□□□□□□□□□override □□□□

```
class C {
  protected fun finalize() {
    // □□□□
  }
}
```

□□ Java □□□□ `finalize()` □□□ private□

□ Java□□□□

□kotlin□□□□□□□□□□□□□□java□(java□□□□□□□)□□□java□□□□□□□□□□□□□□□□

□□□□□□□

java□□□□□□□□□□□□□ “□□□□□”□□□□□□□□□□“□□□□□”□□□□□□□□□ □□□□□□□□□□□□□□□□□□

```
if (Character.isLetter(a)) {
  // ...
}
```

Java □□

Java□□□□□□□kotlin□□□□□□□□□□□□□□□□□□ `instance.javaClass` □□ `ClassName::class.java` □□□□ `java.lang.Class` □java□□□□□

Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, a `KProperty` for a Java field, a Java method or constructor for a `KFunction` and vice versa.

SAM(□□□□□) □□

□□ Java 8 □□□Kotlin □□ SAM □□□□□□□□ Kotlin □□□□□□□□□□□□□□□□ □□□□□□□□□□□ Java □□□□□□□□□□□□□□□□□□□□ □□□□□ Kotlin □□□□□□□□□□□□□□

□□□□□□□□SAM□□□□□□□

```
val runnable = Runnable { println("This runs in a runnable") }
```

...□□□□□□:

```
val executor = ThreadPoolExecutor()
// Java□□: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

□□ Java □□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□ SAM □□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

□□SAM□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□

□□□□□□□□□□□□□□ Java □□□□□□□□ Kotlin □□□□□□□□□□□□□□□□□□□□ Kotlin □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□Kotlin□□□JNI

□□□□□□□□□□□□□□□□(C □□ C++)□□□□□□□□□□□□□`external`□□□(□□□Java□□`native`)

```
external fun foo(x: Int): Double
```

□□□□□□□Java□□□□

Java□□Kotlin□□

Java□□□□□□Kotlin□□□

□□

□□getters□□□□ *get*-□□□setters□□□*set*-□□□

□□□□□□

`example.kt` □□□ `org.foo.bar` □□□□□□□□□□□□□□□□□□□□□ `org.foo.bar.ExampleKt` □java□□□

```
// example.kt
package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

□□□□ `@JvmName` □□□□□□□□Java □□□□□

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

□□□□□□□□□□□□□Java□□□□□□□□□□□□□□□□□`@JvmName` □□□□□□□□□□□□□□□□□□□□□`@JvmMultifileClass` □
□□□

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() {
}
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() {
}
```

```
// Java
demo.Utils.foo();
demo.Utils.bar();
```

□□□□

□□□ Java □□□□□□□□□□□□ Kotlin □□□□□□□□ `@JvmField` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□(backing field)□□ □□□□□□ `open` , `override` □□ `const` □□□□□□□□□□□□□□□□□□□□□□□□ `@JvmField` □□□

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

□□□□□□ □□□□□□Java□□□□□□□□□□□□□□□□□□□□□□□□□ `lateinit` □□□□□□setter□□□□

□□□□

□□□□□□□□□□□□□□□□□□□□Koltin□□□□□□□□□□□□□(backing fields)□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□private□□□□□□□□□□□□□□□□□□□□□□□□

— `@JvmField` □□;

— `lateinit` □□□;

— `const` □□□.

□ `@JvmField` □□□□□□□□□□□□□□□□□□□□□□□□□□

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

□□□□□□□□□□□□□□□□[□□□□□] □□□□□□□□□□□□□□□□□□□□□□setter□□□□□□□□□□

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

□□ `const` □□□□□□□ Kotlin □□□□□□□ Java □□□□□□□□□□

```
// file example.kt

object Obj {
  const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

In Java:

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

□□□□

□□□□□□Kotlin □□□□□□□□□□□□□□ □Kotlin □□□□□@JvmStatic □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□

```
class C {
  companion object {
    @JustvmStatic fun foo() {}
    fun bar() {}
  }
}
```

□□□foo() □java□□□□□□□□bar() □□□

```
C.foo(); // □□□
C.bar(); // □□: □□□□□□□□
```

□□□□□□□□□

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

Java □□

```
Obj.foo(); // □□□
Obj.bar(); // □□
Obj.INSTANCE.bar(); // □□□□□□□□
Obj.INSTANCE.foo(); // □□
```

□□□□□@JvmStatic □□□□□□□□□□□□□□□□□getter □ setter □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□@JvmName□□□□□□

□□□□□□□□ Kotlin □□□□□□□□□□□□□□□□□ JVM □□□ □□□□□□□□*□□□□□*:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

□□□□□□□□□□□□□□□□□ JVM □□□□□□filterValid(Ljava/util/List;)Ljava/util/List; . □□□□□□□□□□□□ Kotlin□□□□□□ □□□□□□□@JvmName □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

□Kotlin□□□□□□□filterValid □□□□□□Java□□□□□filterValid □ filterValidInt .

□□□□□□□□□□□ `x` □□□ `getX()` □□□

```kotlin
val x: Int
  @JvmName("getX_prop")
  get() = 15

fun getX() = 10
```

□□□□

□□□□□□□□□□□□□□□□ Kotlin □□□ Java □□□□□□□□□□□□□□□□□□□□□□□□□□□java□□□□□□□□□ @JvmOverloads
□□

```kotlin
@JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
    ...
}
```

□□

```java
// Java
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□

□□□□□Secondary Constructors □□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□ □□@JvmOverloads
□□□□□

□□□□

□□□□□kotlin□□□□□□□ □□□□□□kotlin□□□java□□□□□□□□□□□ □□□□□□□□□kotlin□□□

```kotlin
// example.kt
package demo

fun foo() {
  throw IOException()
}
```

□□□□□□□java□□□□□□□□□□□□

```
// Java
try {
  demo.Example.foo();
}
catch (IOException e) { // □□: foo() □□□□ IOException
  // ...
}
```

□□ `foo()` □□□□ `IOException` □java□□□□□□□□□□ □□□□□□□□□□□□kotlin□□ `@throws` □□□

```
@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

Null□□□

□□ Java □□□ Kotlin □□□□□□□□□□□□□null □□□□□□□□ □□□□□□ Kotlin □□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ Java □□□□□□□□ `NullPointerException` □

□□□□

□Kotlin □□□□□declaration-site variance□□Java □□

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

□□□□□□□□□□□□□□□□□ Java □□□□□□□□□□□□□□□

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

□□□□□Kotlin □□□□□□□□□ `unboxBase(boxDerived("s"))` □□□□□□□□□□ Java □□□□□□□□□□□□ Java `Box` □□□□□ `T` □*□□□□*□ `Box<Derived>` □□□□□□□ `Box<Base>` □□□□□□□□ Java □□□□□□□□□□□□□□□□□□□□ `unboxBase` □ [//]:# (The problem is that in Kotlin we can say , but in Java that would be impossible, because in Java the class `Box` is *invariant* in its parameter `T`, and thus `Box<Derived>` is not a subtype of `Box<Base>`. To make it work in Java we'd have to define `unboxBase` as follows:)

```
Base unboxBase(Box<? extends Base> box) { ... }
```

□□□□□□□□□□ Java □*□□□□□*(`? extends Base`) □□□declaration-site variance□□□□ Java □□□□□□□□

□□□□□□□□□□□□□ Kotlin □ API □□□□□ `Box` □□□Koltin □□ `Box<Super>` □ Java □□□□ `Box<? extends Super>` (`Foo` □□□□ `Foo<? super Bar>`) □□□ Java □□□□□□□□□□□□□□(□□□□ Java □□□□)□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
// □□□□□□□ - □□□□
Box<Derived> boxDerived(Derived value) { ... }

// □□□□□ - □□□□
Base unboxBase(Box<? extends Base> box) { ... }
```

□□□□□□□□□□□ final □□□□□□□□□□□□□□□□□□□□□□ `Box<String>` □□□ Java □□□□□□□ `Box<String>` □

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `@JvmWildcard` □□□

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// □□□□□□
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `@JvmSuppressWildcards`

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// □□□□□□
// Base unboxBase(Box<Base> box) { ... }
```

□□□ `@JvmSuppressWildcards` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Java □□□□□

Nothing □□□□□

`Nothing` □□□□□□□□□□□□□□□ Java □□□□□□□□□□□□□□□□□□□ Java □□□□□□□□ `java.lang.Void` □□□□□□ `null` □□□□ Nothing □□□□□□ Java □□□□□□□□□□□□□□□□□□□□□□□□□Kotlin □□□□□□□□□□□□□□ `Nothing` □

```
fun emptyList(): List<Nothing> = listOf()
// □□□□
// List emptyList() { ... }
```

□□

□□kotlin□□□□

KDoc□□□□Kotlin□□□□□□□□java□ JavaDoc□□□□□□□□□□KDoc □□□JavaDoc□□□□□□□□Markdown□□□□□□□□□□ Kotlin□□□□□□□

Generating the Documentation

Kotlin’s documentation generation tool is called Dokka. See the Dokka README for usage instructions.

Dokka has plugins for Gradle, Maven and Ant, so you can integrate documentation generation into your build process.

KDoc □□

□JavaDoc□□□KDoc□□□ `/**` □□□□ `*/` □□,□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□ `@` □□□□

□□□ KDoc □□□□□□□□□□□

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

□□□

KDoc□□□□□□□□□□□

@param <name>

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□

```
@param name description.
@param[name] description.
```

@return

□□□□□□

@constructor

□□□□□

@receiver

Documents the receiver of an extension function.

@property <name>

Documents the property of a class which has the specified name. This tag can be used for documenting properties declared in the primary constructor, where putting a doc comment directly before the property definition would be awkward.

@throws <class>, @exception <class>

□□□□□□□□□□□□□□□□Kotlin□□□□□□□ □□□ □□□□□□

@sample <identifier>

□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□

@see <identifier>

□□□□□□□□□□□□□ □□□□□

@author

□□□□□□□□□

@since

□□□□□□□□□□□□□□□

@suppress

□□□□□□□□□□□□□□□□□□□□□API□ □□□□□□□□□□□□□□□□□□

⚠ KDoc □□□ @deprecated □□□□. □□□ @Deprecated□□

□□Markup□□

□□Markup□□□KDoc□□□□□□Markdown □□,□□□□ □□□□□□□□□□□□□□□□□□□□□□□□

□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Use the method [foo] for this purpose.

If you want to specify a custom label for the link, use the Markdown reference-style syntax:

Use [this method][foo] for this purpose.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□javadoc□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□

Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□KDoc□□□□□□□□□□□□

□□KDoc□□□□□□□□□□□□□□□□□□□□□□□□□□Kotlin□□□□□ □□□ □□□□□□□□□□□

Module and Package Documentation

Documentation for a module as a whole, as well as packages in that module, is provided as a separate Markdown file, and the paths to that file is passed to Dokka using the `-include` command line parameter or the corresponding parameters in Ant, Maven and Gradle plugins.

Inside the file, the documentation for the module as a whole and for individual packages is introduced by the corresponding first-level headings. The text of the heading must be “Module `<module name>`” for the module, and “Package `<package qualified name>`” for a package.

Here’s an example content of the file:

```
# Module kotlin-demo

The module shows the Dokka syntax usage.

# Package org.jetbrains.kotlin.demo

Contains assorted useful stuff.

## Level 2 heading

Text after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

Useful stuff in another package.
```

□□ Maven

□□□□□

kotlin-maven-plugin □□□□ Kotlin □□□□□□□□□□□□ Marven V3

□□ *kotlin.version* □□□□□□□□ Kotlin □□□The correspondence between Kotlin releases and versions is displayed below:

Milestone	Version
1.0.3	1.0.3
1.0.2 hotfix update	1.0.2-1
1.0.2	1.0.2
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

□□

Kotlin □□□□□□□□□□□□□□□□□□□□□□□□ pom □□□□□□□□□□□

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

□□□ Kotlin □□

□ <build> □□□□□□□□□□ Kotlin □□□□□

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

Maven □□□□□ Kotlin □□□□□□□□□

```
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals> <goal>compile</goal> </goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals> <goal>test-compile</goal> </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

□□□□ Kotlin □ Java □□

□□□□□□□ Kotlin □□□□□□ Java □□□□□□□□ □ Maven □□□□ kotlin-maven-plugin □□ maven-compiler-plugin □□□

It could be done by moving Kotlin compilation to previous phase, process-sources□□□□□□□□□□□□□□□□□□□□

```
<build>
   <plugins>
      <plugin>
         <artifactId>kotlin-maven-plugin</artifactId>
         <groupId>org.jetbrains.kotlin</groupId>
         <version>${kotlin.version}</version>

         <executions>
            <execution>
               <id>compile</id>
               <phase>process-sources</phase>
               <goals> <goal>compile</goal> </goals>
            </execution>

            <execution>
               <id>test-compile</id>
               <phase>process-test-sources</phase>
               <goals> <goal>test-compile</goal> </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

OSGi

OSGi□□□□ Kotlin OSGi page.

□□

Maven □□□□□□□Github □□□□

Using Ant

Getting the Ant Tasks

Kotlin provides three tasks for Ant:

— kotlinc: Kotlin compiler targeting the JVM

— kotlin2js: Kotlin compiler targeting JavaScript

— withKotlin: Task to compile Kotlin files when using the standard *javac* Ant task

These tasks are defined in the *kotlin-ant.jar* library which is located in the *lib* folder for the Kotlin Compiler

Targeting JVM with Kotlin-only source

When the project consists of exclusively Kotlin source code, the easiest way to compile the project is to use the *kotlinc* task

```
<project name="Ant Task Test" default="build">
   <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

   <target name="build">
      <kotlinc src="hello.kt" output="hello.jar"/>
   </target>
</project>
```

where ${kotlin.lib} points to the folder where the Kotlin standalone compiler was unzipped.

Targeting JVM with Kotlin-only source and multiple roots

If a project consists of multiple source roots, use *src* as elements to define paths

```
<project name="Ant Task Test" default="build">
   <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

   <target name="build">
      <kotlinc output="hello.jar">
         <src path="root1"/>
         <src path="root2"/>
      </kotlinc>
   </target>
</project>
```

Targeting JVM with Kotlin and Java source

If a project consists of both Kotlin and Java source code, while it is possible to use *kotlinc*, to avoid repetition of task parameters, it is recommended to use *withKotlin* task

```
<project name="Ant Task Test" default="build">
   <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

   <target name="build">
      <delete dir="classes" failonerror="false"/>
      <mkdir dir="classes"/>
      <javac destdir="classes" includeAntRuntime="false" srcdir="src">
         <withKotlin/>
      </javac>
      <jar destfile="hello.jar">
         <fileset dir="classes"/>
      </jar>
   </target>
</project>
```

To specify additional command line arguments for `<withKotlin>`, you can use a nested `<compilerArg>` parameter. The full list of arguments that can be used is shown when you run `kotlinc -help`. You can also specify the name of the module being compiled as the `moduleName` attribute:

```
<withKotlin moduleName="myModule">
   <compilerarg value="-no-stdlib"/>
</withKotlin>
```

Targeting JavaScript with single source folder

```
<project name="Ant Task Test" default="build">
   <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

   <target name="build">
      <kotlin2js src="root1" output="out.js"/>
   </target>
</project>
```

Targeting JavaScript with Prefix, PostFix and sourcemap options

```
<project name="Ant Task Test" default="build">
   <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

   <target name="build">
      <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="postfix"
sourcemap="true"/>
   </target>
</project>
```

Targeting JavaScript with single source folder and metaInfo option

The `metaInfo` option is useful, if you want to distribute the result of translation as a Kotlin/JavaScript library. If `metaInfo` was set to `true`, then during compilation additional JS file with binary metadata will be created. This file should be distributed together with the result of translation.

```
<project name="Ant Task Test" default="build">
   <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-
ant.jar"/>

   <target name="build">
      <!-- out.meta.js will be created, which contains binary descriptors -->
      <kotlin2js src="root1" output="out.js" metaInfo="true"/>
   </target>
</project>
```

References

Complete list of elements and attributes are listed below

Attributes common for kotlinc and kotlin2js

Name	Description	Required	Default Value
src	Kotlin source file or directory to compile	Yes	
nowarn	Suppresses all compilation warnings	No	false
noStdlib	Does not include the Kotlin standard library into the classpath	No	false
failOnError	Fails the build if errors are detected during the compilation	No	true

kotlinc Attributes

Name	Description	Required	Default Value
output	Destination directory or .jar file name	Yes	
classpath	Compilation class path	No	
classpathref	Compilation class path reference	No	
includeRuntime	If output is a .jar file, whether Kotlin runtime library is included in the jar	No	true
moduleName	Name of the module being compiled	No	The name of the target (if specified) or the project

kotlin2js Attributes

Name	Description	Required
output	Destination file	Yes
library	Library files (kt, dir, jar)	No

Name	Description	Required
outputPrefix	Prefix to use for generated JavaScript files	No
outputSuffix	Suffix to use for generated JavaScript files	No
sourcemap	Whether sourcemap file should be generated	No
metaInfo	Whether metadata file with binary descriptors should be generated	No
main	Should compiler generated code call the main function	No

□□ Gradle

In order to build Kotlin with Gradle you should set up the *kotlin-gradle* plugin, apply it to your project and add *kotlin-stdlib* dependencies. Those actions may also be performed automatically in IntelliJ IDEA by invoking the Tools | Kotlin | Configure Kotlin in Project action.

You can also enable incremental compilation to make your builds faster.

□□□□□

□□ *kotlin-gradle-plugin* □□Kotlin□□□□□□□.

□□□ Kotlin □□□□□□□□*kotlin.version*□□□□□:

```
buildscript {
  ext.kotlin_version = '<version to use>'

  repositories {
    mavenCentral()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
  }
}
```

The correspondence between Kotlin releases and versions is displayed below:

□□□	□□
1.0.3	1.0.3
1.0.2 hotfix update	1.0.2-1
1.0.2	1.0.2
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200

□□□	□□
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

□□□JVM

□□□JVM□□□, Kotlin□□□□□□□□

```
apply plugin: "kotlin"
```

Kotlin□□□□Java□□□□□□□□□□□□□□□□, □□□□□□□□□□□. □□□□□□□□□□□□□□:

```
project
    - src
        - main (root)
            - kotlin
            - java
```

□□□□□□□□□□□□□□□□□□□□□□□*sourceSets* □□

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

□□□ JavaScript

□□□□ JavaScript □□□, □□□□□□□□□□□□:

```
apply plugin: "kotlin2js"
```

□□□□□□□□Kotlin□□□□□□□□□□□□□□□□□□Kotlin□Java□□ (□□□□□□□□□□□□□□□□Java□□□□□□). □□ □□□□□□□□□□□□□□□□ JVM□□□□□□□□□□□□□□□ *sourceSets*

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

□□□□□□□□□□□□□□□□, □□ `kotlinOptions.metaInfo` □□□□□□□□□□□□□□□JS□□. □□□□□□□□□□□□□□□□□□□.

```
compileKotlin2Js {
 kotlinOptions.metaInfo = true
}
```

□□□ Android

Android□ Gradle□□□□□□□Gradle□□□□□, □□□□□□□□□□□□Kotlin□□□□□□Android□□, □□□□*kotlin-android* □□□□□□ *kotlin*:

```
buildscript {
    ...
}
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
```

Android Studio

□□□□□□□□Android Studio, □□□□□□□□□□□□□□□□□□:

```
android {
  ...

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

□□□□□□□□kotlin□□□Android Studio□□□□□□□□□□□□, □□□□□□□□□□□□IDE□□□□□□□□. Alternatively, you can put Kotlin classes in the Java source directory, typically located in `src/main/java`.

□□□□

In addition to the kotlin-gradle-plugin dependency shown above, you need to add a dependency on the Kotlin standard library:

```
buildscript {
   ext.kotlin_version = '<version to use>'
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
  }
}

apply plugin: "kotlin" // or apply plugin: "kotlin2js" if targeting JavaScript

repositories {
  mavenCentral()
}

dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

If your project uses Kotlin reflection or testing facilities, you need to add the corresponding dependencies as well:

```
compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
testCompile "org.jetbrains.kotlin:kotlin-test:$kotlin_version"
testCompile "org.jetbrains.kotlin:kotlin-test-junit:$kotlin_version"
```

Annotation processing

The Kotlin plugin supports annotation processors like *Dagger* or *DBFlow*. In order for them to work with Kotlin classes, add the respective dependencies using the `kapt` configuration in your `dependencies` block:

```
dependencies {
  kapt 'groupId:artifactId:version'
}
```

If you previously used the android-apt plugin, remove it from your `build.gradle` file and replace usages of the `apt` configuration with `kapt`. If your project contains Java classes, `kapt` will also take care of them. If you use annotation processors for your `androidTest` or `test` sources, the respective `kapt` configurations are named `kaptAndroidTest` and `kaptTest`.

Some annotation processing libraries require you to reference generated classes from within your code. For this to work, you'll need to add an additional flag to enable the *generation of stubs* to your build file:

```
kapt {
    generateStubs = true
}
```

Note, that generation of stubs slows down your build somewhat, which is why it's disabled by default. If generated classes are referenced only in a few places in your code, you can alternatively revert to using a helper class written in Java which can be seamlessly called from your Kotlin code.

For more information on `kapt` refer to the official blogpost.

Incremental compilation

Kotlin 1.0.2 introduced new experimental incremental compilation mode in Gradle. Incremental compilation tracks changes of source files between builds so only files affected by these changes would be compiled.

There are several ways to enable it:

1. add `kotlin.incremental=true` line either to a `gradle.properties` or a `local.properties` file;
2. add `-Pkotlin.incremental=true` to gradle command line parameters. Note that in this case the parameter should be added to each subsequent build (any build without this parameter invalidates incremental caches).

After incremental compilation is enabled, you should see the following warning message in your build log: `Using experimental kotlin incremental compilation`

Note, that the first build won't be incremental.

OSGi

OSGi □□□□ Kotlin OSGi page.

□□

Kotlin Repository □□□□□:

— Kotlin

— Mixed Java and Kotlin

— Android

— JavaScript

Kotlin and OSGi

To enable Kotlin OSGi support you need to include `kotlin-osgi-bundle` instead of regular Kotlin libraries. It is recommended to remove `kotlin-runtime`, `kotlin-stdlib` and `kotlin-reflect` dependencies as `kotlin-osgi-bundle` already contains all of them. You also should pay attention in case when external Kotlin libraries are included. Most regular Kotlin dependencies are not OSGi-ready, so you shouldn't use them and should remove them from your project.

Maven

To include the Kotlin OSGi bundle to a Maven project:

```
<dependencies>
    <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-osgi-bundle</artifactId>
      <version>${kotlin.version}</version>
    </dependency>
 </dependencies>
```

To exclude the standard library from external libraries (notice that "star exclusion" works in Maven 3 only)

```
    <dependency>
      <groupId>some.group.id</groupId>
      <artifactId>some.library</artifactId>
      <version>some.library.version</version>

      <exclusions>
        <exclusion>
          <groupId>org.jetbrains.kotlin</groupId>
          <artifactId>*</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
```

Gradle

To include `kotlin-osgi-bundle` to a gradle project:

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

To exclude default Kotlin libraries that comes as transitive dependencies you can use the following approach

```
dependencies {
 compile (
   [group: 'some.group.id', name: 'some.library', version: 'someversion'],
   .....) {
  exclude group: 'org.jetbrains.kotlin'
}
```

FAQ

Why not just add required manifest options to all Kotlin libraries

Even though it is the most preferred way to provide OSGi support, unfortunately it couldn’t be done for now due to so called “package split” issue that could’t be easily eliminated and such a big change is not planned for now. There is `Require-Bundle` feature but it is not the best option too and not recommended to use. So it was decided to make a separate artifact for OSGi.

□□□□

FAQ

□□□□

Kotlin□□□□

Kotlin □□□□□□ JVM □ JavaScript □□□□□□□□□□□□□□□□□□□□□□□□□

□□□ JetBrains □□□□□□□□□□□□□□□□OSS□□□□□□□□□□□□□□

□□□□□□□□□□□

□ JetBrains □□□□□ Java □□□□□□□□□□□□□□□□□Java□□□□□ □□□□□□□□□□□□□□□□□□ Java □□□□□□□□□□□□□□□□□□□□□□□□□□□□ Java □□□□□□□□□ □□□□□□□□□□□□□□□□□ JVM □□□□□□□□□□□ □□□□□□□□□□□□□□□□□□

The core values behind the design of Kotlin make it

- Interoperable: Kotlin can be freely mixed with Java,
- Safe: statically check for common pitfalls (e.g., null pointer dereference) to catch errors at compile time,
- Toolable: enable precise and performant tools such as IDEs and build systems,
- “Democratic”: make all parts of the langauge available to all developers (no policies are needed to restrict the use of some features to library writers or other groups of developers).

How is it licensed?

□□□□□

Kotlin □□□□□□□□□ Apache 2 □□□□□□□□□□□ IntelliJ □□□□□□□□□

□□□□ Github □□□□□□□□□□□□□□

Where can I get an HD Kotlin logo?

Logos can be downloaded here. Please follow simple rules in the `readme.txt` inside the archive.

□□□Java□

□□□□□□□□□□□ Java □□□□Kotlin □□□□ Java □□ Java □□□□□ Kotlin□□□□Java □□□□

[illegible]Kotlin[illegible]Java[illegible]

Kotlin [illegible] Java 6 [illegible] Kotlin [illegible] Android [illegible] Java 6 [illegible]

[illegible]

[illegible] Apache 2 [illegible] IntelliJ IDEA [illegible] [illegible] IntelliJ IDEA [illegible] Kotlin[illegible]

[illegible]Eclipse[illegible]

[illegible]

[illegible]

[illegible] Github [illegible]

Kotlin[illegible]

Kotlin [illegible] lambda [illegible] Kotlin [illegible] map[illegible] flatMap[illegible]reduce [illegible] Kotlin [illegible]

Kotlin[illegible]

Kotlin [illegible]Kotlin [illegible]

[illegible]

[illegible]

[illegible]

[illegible]Scala [illegible]

[illegible]

[illegible]

Kotlin[illegible]

[illegible]

[illegible]DSL[illegible]

[illegible]Kotlin [illegible]

Kotlin for JavaScript [illegible] ECMAScript[illegible]

[illegible] 5[illegible]

JavaScript[illegible]

□□□□□□□□□ CommonJS □ AMD □□□□□□

□□Java

Kotlin□□□□□Java□□□□□□

Kotlin □□□Java□□□□□□□□□□□□□□

— □□□□□□□□□□□□□□.

— □□□□□□□

— Kotlin□□□□□□□□

— □□□Java□SAM-conversions□Kotlin□□□□□□□□□□

— □□□□□□□□□site variance

— Kotlin □□□□□□□

Java□□□Kotlin□□□□□

— □□□□□

— □□□□□ □□□□

— □□□□□

— □□□□□□□

— □□□□□□□

Kotlin□□□Java□□□□□

— Lambda □□□ + □□□□□ = □□□□□□□□□

— □□□□□

— Null □□□

— □□□□□□□

— □□□□□□

— □□

— □□□□□□

— □□□□□

— □□□□□□□□□□□□□□□

— □□

— □□□□□□ & □□□□□

— □□□□□□

— □□□□□□

— □□□□□

— Data classes

— Separate interfaces for read-only and mutable collections

□□Scala

The main goal of the Kotlin team is to create a pragmatic and productive programming language, rather than to advance the state of the art in programming language research. □□□□, □□□□Scala□□□ □□□□,□□□□□□□□□Kotlin

Scala□□□Kotlin□□□

— □□□□, □□□□□, □□

— □Scala□, □□□□□□□□□□□□□□□□□□□□□□debugger,□□□□□code□□□□□□□□

— □□□□□□□□□□Kotlin□□□□□□.

— □□□□□□□

— □□□□□□□□

— □

— □□□□

— □□□□ □□□□□□□□□□□

— □□ □□□□

— □□□□□□□□

— □□ □□□□□

— □□□□

— □□□

— We plan to support Project Valhalla once it is released as part of the JDK

— Yield operator

— Actors

— Kotlin supports Quasar, a third-party framework for actor support on the JVM

— □□□□

— Kotlin supports Java 8 streams, which provide similar functionality

Kotlin□□□Scala□□□

— □□□ □□□

— Scala □ Option□□□□□□□□□□□

— □□□□

— Kotlin□□□□□□□□□□□□

— □□□ □□. Also implemented via 3rd party plugin: Autoproxy