



Kotlin Language Documentation

Table of Contents

Getting Started	5
Basic Syntax	5
Idioms	10
Coding Conventions	14
What's New in Kotlin 1.1	16
Basics	25
Basic Types	25
Packages	30
Control Flow	32
Returns and Jumps	35
Classes and Objects	37
Classes and Inheritance	37
Properties and Fields	42
Interfaces	45
Visibility Modifiers	47
Extensions	49
Data Classes	53
Sealed Classes	54
Generics	55
Nested Classes	60
Enum Classes	61
Object Expressions and Declarations	63
Delegation	67
Delegated Properties	68
Functions and Lambdas	73
Functions	73
Higher-Order Functions and Lambdas	78
Inline Functions	82
Coroutines	85

Other	90
Destructuring Declarations	90
Collections	92
Ranges	94
Type Checks and Casts	97
This Expression	99
Equality	100
Operator overloading	101
Null Safety	104
Exceptions	107
Annotations	109
Reflection	113
Type-Safe Builders	117
Reference	123
Grammar	123
Notation	123
Semicolons	123
Syntax	123
Lexical structure	131
Compatibility	133
Java Interop	137
Calling Java code from Kotlin	137
Calling Kotlin from Java	144
JavaScript	151
Kotlin JavaScript Overview	151
Dynamic Type	152
Calling JavaScript from Kotlin	153
Calling Kotlin from JavaScript	156
JavaScript Modules	158
JavaScript Reflection	161
Tools	162
Documenting Kotlin Code	162

Using Gradle	165
Using Maven	170
Using Ant	175
Kotlin and OSGi	178
Compiler Plugins	179
FAQ	183
FAQ	183
Comparison to Java	185
Comparison to Scala	186

Getting Started

Basic Syntax

Defining packages

Package specification should be at the top of the source file:

```
package my.demo

import java.util.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

See [Packages](#).

Defining functions

Function having two `Int` parameters with `Int` return type:

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

Function with an expression body and inferred return type:

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

Function returning no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

`Unit` return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

See [Functions](#).

Defining local variables

Assign-once (read-only) local variable:

```

val a: Int = 1 // immediate assignment
val b = 2      // `Int` type is inferred
val c: Int    // Type required when no initializer is provided
c = 3         // deferred assignment

```

Mutable variable:

```

var x = 5 // `Int` type is inferred
x += 1

```

See also [Properties And Fields](#).

Comments

Just like Java and JavaScript, Kotlin supports end-of-line and block comments.

```

// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */

```

Unlike Java, block comments in Kotlin can be nested.

See [Documenting Kotlin Code](#) for information on the documentation comment syntax.

Using string templates

```

var a = 1
// simple name in template:
val s1 = "a is $a"

a = 2
// arbitrary expression in template:
val s2 = "${s1.replace("is", "was")}, but now is $a"

```

See [String templates](#).

Using conditional expressions

```

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

```

Using **if** as an expression:

```

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

```

See [if-expressions](#).

Using nullable values and checking for **null**

A reference must be explicitly marked as nullable when **null** value is possible.

Return **null** if `str` does not hold an integer:

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

Use a function returning nullable value:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("either '$arg1' or '$arg2' is not a number")
    }
}
```

or

```
// ...
if (x == null) {
    println("Wrong number format in arg1: '${arg1}'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '${arg2}'")
    return
}

// x and y are automatically cast to non-nullable after null check
println(x * y)
```

See [Null-safety](#).

Using type checks and automatic casts

The `is` operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
```

or

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

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}
```

or even

```

fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand side of `&&`
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}

```

See [Classes](#) and [Type casts](#).

Using a for loop

```

val items = listOf("apple", "banana", "kiwi")
for (item in items) {
    println(item)
}

```

or

```

val items = listOf("apple", "banana", "kiwi")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}

```

See [for loop](#).

Using a while loop

```

val items = listOf("apple", "banana", "kiwi")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}

```

See [while loop](#).

Using when expression

```

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

```

See [when expression](#).

Using ranges

Check if a number is within a range using `in` operator:


```

val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}

```

Check if a number is out of range:

```

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

if (-1 !in 0..list.lastIndex) {
    println("-1 is out of range")
}
if (list.size !in list.indices) {
    println("list size is out of valid list indices range too")
}

```

Iterating over a range:

```

for (x in 1..5) {
    print(x)
}

```

or over a progression:

```

for (x in 1..10 step 2) {
    print(x)
}
for (x in 9 downTo 0 step 3) {
    print(x)
}

```

See [Ranges](#).

Using collections

Iterating over a collection:

```

for (item in items) {
    println(item)
}

```

Checking if a collection contains an object using `in` operator:

```

when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}

```

Using lambda expressions to filter and map collections:

```

fruits
    .filter { it.startsWith("a") }
    .sortedBy { it }
    .map { it.toUpperCase() }
    .forEach { println(it) }

```

See [Higher-order functions and Lambdas](#).

Idioms

A collection of random and frequently used idioms in Kotlin. If you have a favorite idiom, contribute it. Do a pull request.

Creating DTOs (POJOs/POCOs)

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

provides a `Customer` class with the following functionality:

- getters (and setters in case of `vars`) for all properties
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- `component1()`, `component2()`, ..., for all properties (see [Data classes](#))

Default values for function parameters

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

Filtering a list

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

Or alternatively, even shorter:

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

String Interpolation

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

Instance Checks

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

Traversing a map/list of pairs

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

`k`, `v` can be called anything.

Using ranges

```

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) { ... }

```

Read-only list

```

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

```

Read-only map

```

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

```

Accessing a map

```

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

```

Lazy property

```

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

```

Extension Functions

```

fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()

```

Creating a singleton

```

object Resource {
    val name = "Name"
}

```

If not null shorthand

```

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

println(files?.size)

```

If not null and else shorthand

```

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

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

```

Executing a statement if null

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

Execute if not null

```
val data = ...

data?.let {
    ... // execute this block if not null
}
```

Return on when statement

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

'try/catch' expression

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

    // Working with result
}
```

'if' expression

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

Builder-style usage of methods that return Unit

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

Single-expression functions

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

This is equivalent to

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

This can be effectively combined with other idioms, leading to shorter code. E.g. with the [when-expression](#):

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

Calling multiple methods on an object instance ('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's 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
}
```

Coding Conventions

This page contains the current coding style for the Kotlin language.

Naming Style

If in doubt default to the Java Coding Conventions such as:

- use of camelCase for names (and avoid underscore in names)
- types start with upper case
- methods and properties start with lower case
- use 4 space indentation
- public functions should have documentation such that it appears in Kotlin Doc

Colon

There is a space before colon where colon separates type and supertype and there's no space where colon separates instance and type:

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

Lambdas

In lambda expressions, spaces should be used around the curly braces, as well as around the arrow which separates the parameters from the body. Whenever possible, a lambda should be passed outside of parentheses.

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

In lambdas which are short and not nested, it's recommended to use the `it` convention instead of declaring the parameter explicitly. In nested lambdas with parameters, parameters should be always declared explicitly.

Class header formatting

Classes with a few arguments can be written in a single line:

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

Classes with longer headers should be formatted the way, that each primary constructor argument is in a separate line with indentation. Also, the closing parenthesis should be on a new line. If we use inheritance, then the superclass constructor call or list of implemented interfaces should be located on the same line as the parenthesis:

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

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

```

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

```

Constructor parameters can use either the regular indent or the continuation indent (double the regular indent).

Unit

If a function returns Unit, the return type should be omitted:

```

fun foo() { // ": Unit" is omitted here

}

```

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

What's New in Kotlin 1.1

Table of Contents

- [Coroutines](#)
- [Other language features](#)
- [Standard library](#)
- [JVM backend](#)
- [JavaScript backend](#)

JavaScript

Starting with Kotlin 1.1, the JavaScript target is no longer considered experimental. All language features are supported, and there are many new tools for integration with the front-end development environment. See [below](#) for a more detailed list of changes.

Coroutines (experimental)

The key new feature in Kotlin 1.1 is *coroutines*, bringing the support of `async / await`, `yield` and similar programming patterns. The key feature of Kotlin's design is that the implementation of coroutine execution is part of the libraries, not the language, so you aren't bound to any specific programming paradigm or concurrency library.

A coroutine is effectively a light-weight thread that can be suspended and resumed later. Coroutines are supported through [suspending functions](#): a call to such a function can potentially suspend a coroutine, and to start a new coroutine we usually use an anonymous suspending functions (i.e. suspending lambdas).

Let's look at `async / await` which is implemented in an external library, [kotlinx.coroutines](#):

```
// runs the code in the background thread pool
fun asyncOverlay() = async(CommonPool) {
    // start two async operations
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // and then apply overlay to both results
    applyOverlay(original.await(), overlay.await())
}

// launches new coroutine in UI context
launch(UI) {
    // wait for async overlay to complete
    val image = asyncOverlay().await()
    // and then show it in UI
    showImage(image)
}
```

Here, `async { ... }` starts a coroutine and, when we use `await()`, the execution of the coroutine is suspended while the operation being awaited is executed, and is resumed (possibly on a different thread) when the operation being awaited completes.

The standard library uses coroutines to support *lazily generated sequences* with `yield` and `yieldAll` functions. In such a sequence, the block of code that returns sequence elements is suspended after each element has been retrieved, and resumed when the next element is requested. Here's an example:


```

val seq = buildSequence {
    for (i in 1..5) {
        // yield a square of i
        yield(i * i)
    }
    // yield a range
    yieldAll(26..28)
}

// print the sequence
println(seq.toList())

```

Run the code above to see the result. Feel free to edit it and run again!

For more information, please refer to the [coroutine documentation](#) and [tutorial](#).

Note that coroutines are currently considered an **experimental feature**, meaning that the Kotlin team is not committing to supporting the backwards compatibility of this feature after the final 1.1 release.

Other Language Features

Type aliases

A type alias allows you to define an alternative name for an existing type. This is most useful for generic types such as collections, as well as for function types. Here is an example:

```

typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
    oscarWinners.count { it.value.contains("La La Land") }

// Note that the type names (initial and the type alias) are interchangeable:
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
    oscarWinners["Best picture"] == "La La Land"

```

See the [documentation](#) and [KEEP](#) for more details.

Bound callable references

You can now use the `::` operator to get a [member reference](#) pointing to a method or property of a specific object instance. Previously this could only be expressed with a lambda. Here's an example:

```

val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)

```

Read the [documentation](#) and [KEEP](#) for more details.

Sealed and data classes

Kotlin 1.1 removes some of the restrictions on sealed and data classes that were present in Kotlin 1.0. Now you can define subclasses of a top-level sealed class on the top level in the same file, and not just as nested classes of the sealed class. Data classes can now extend other classes. This can be used to define a hierarchy of expression classes nicely and cleanly.

```
sealed class Expr

data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

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

val e = eval(Sum(Const(1.0), Const(2.0)))
```

Read the [documentation](#) or [sealed class](#) and [data class](#) KEEPs for more detail.

Destructuring in lambdas

You can now use the [destructuring declaration](#) syntax to unpack the arguments passed to a lambda. Here's an example:

```
val map = mapOf(1 to "one", 2 to "two")
// before
println(map.mapValues { entry ->
    val (key, value) = entry
    "$key -> $value!"
})
// now
println(map.mapValues { (key, value) -> "$key -> $value!" })
```

Read the [documentation](#) and [KEEP](#) for more details.

Underscores for unused parameters

For a lambda with multiple parameters, you can use the `_` character to replace the names of the parameters you don't use:

```
map.forEach { _, value -> println("$value!") }
```

This also works in [destructuring declarations](#):

```
val (_, status) = getResult()
```

Read the [KEEP](#) for more details.

Underscores in numeric literals

Just as in Java 8, Kotlin now allows to use underscores in numeric literals to separate groups of digits:

```
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Read the [KEEP](#) for more details.

Shorter syntax for properties

For properties with the getter defined as an expression body, the property type can now be omitted:

```
data class Person(val name: String, val age: Int) {
    val isAdult get() = age >= 20 // Property type inferred to be 'Boolean'
}
```

Inline property accessors

You can now mark property accessors with the `inline` modifier if the properties don't have a backing field. Such accessors are compiled in the same way as [inline functions](#).

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

You can also mark the entire property as `inline` - then the modifier is applied to both accessors.

Read the [documentation](#) and [KEEP](#) for more details.

Local delegated properties

You can now use the [delegated property](#) syntax with local variables. One possible use is defining a lazily evaluated local variable:

```
val answer by lazy {
    println("Calculating the answer...")
    42
}
if (needAnswer()) {                // returns the random value
    println("The answer is $answer.") // answer is calculated at this point
}
else {
    println("Sometimes no answer is the answer...")
}
```

Read the [KEEP](#) for more details.

Interception of delegated property binding

For [delegated properties](#), it is now possible to intercept delegate to property binding using the `provideDelegate` operator. For example, if we want to check the property name before binding, we can write something like this:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, prop: KProperty<*>): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        ... // property creation
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

The `provideDelegate` method will be called for each property during the creation of a `MyUI` instance, and it can perform the necessary validation right away.

Read the [documentation](#) for more details.

Generic enum value access

It is now possible to enumerate the values of an enum class in a generic way.

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
```

Scope control for implicit receivers in DSLs

The [@DslMarker](#) annotation allows to restrict the use of receivers from outer scopes in a DSL context. Consider the canonical [HTML builder example](#):

```
table {
    tr {
        td { +"Text" }
    }
}
```

In Kotlin 1.0, code in the lambda passed to `td` has access to three implicit receivers: the one passed to `table`, to `tr` and to `td`. This allows you to call methods that make no sense in the context - for example to call `tr` inside `td` and thus to put a `<tr>` tag in a `<td>`.

In Kotlin 1.1, you can restrict that, so that only methods defined on the implicit receiver of `td` will be available inside the lambda passed to `td`. You do that by defining your annotation marked with the `@DslMarker` meta-annotation and applying it to the base class of the tag classes.

Read the [documentation](#) and [KEEP](#) for more details.

rem operator

The `mod` operator is now deprecated, and `rem` is used instead. See [this issue](#) for motivation.

Standard library

String to number conversions

There is a bunch of new extensions on the `String` class to convert it to a number without throwing an exception on invalid number: `String.toIntOrNull(): Int?`, `String.toDoubleOrNull(): Double?` etc.

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

Also integer conversion functions, like `Int.toString()`, `String.toInt()`, `String.toIntOrNull()`, each got an overload with `radix` parameter, which allows to specify the base of conversion (2 to 36).

onEach()

`onEach` is a small, but useful extension function for collections and sequences, which allows to perform some action, possibly with side-effects, on each element of the collection/sequence in a chain of operations. On iterables it behaves like `forEach` but also returns the iterable instance further. And on sequences it returns a wrapping sequence, which applies the given action lazily as the elements are being iterated.

```
inputDir.walk()
    .filter { it.isFile && it.name.endsWith(".txt") }
    .onEach { println("Moving $it to $outputDir") }
    .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also(), takeIf() and takeUnless()

These are three general-purpose extension functions applicable to any receiver.

`also` is like `apply`: it takes the receiver, does some action on it, and returns that receiver. The difference is that in the block inside `apply` the receiver is available as `this`, while in the block inside `also` it's available as `it` (and you can give it another name if you want). This comes handy when you do not want to shadow `this` from the outer scope:

```
fun Block.copy() = Block().also {
    it.content = this.content
}
```

`takeIf` is like `filter` for a single value. It checks whether the receiver meets the predicate, and returns the receiver, if it does or `null` if it doesn't. Combined with an elvis-operator and early returns it allows to write constructs like:

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// do something with existing outDirFile
```

```
val index = input.indexOf(keyword).takeIf { it >= 0 } ?: error("keyword not found")
// do something with index of keyword in input string, given that it's found
```

`takeUnless` is the same as `takeIf`, but it takes the inverted predicate. It returns the receiver when it *doesn't* meet the predicate and `null` otherwise. So one of the examples above could be rewritten with `takeUnless` as following:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

It is also convenient to use when you have a callable reference instead of the lambda:

```
val result = string.takeUnless(String::isEmpty)
```

groupingBy()

This API can be used to group a collection by key and fold each group simultaneously. For example, it can be used to count the number of words starting with each letter:

```
val frequencies = words.groupingBy { it.first() }.eachCount()
```

Map.toMap() and Map.toMutableMap()

These functions can be used for easy copying of maps:

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

The operator `plus` provides a way to add key-value pair(s) to a read-only map producing a new map, however there was not a simple way to do the opposite: to remove a key from the map you have to resort to less straightforward ways to like `Map.filter()` or `Map.filterKeys()`. Now the operator `minus` fills this gap. There are 4 overloads available: for removing a single key, a collection of keys, a sequence of keys and an array of keys.

```
val map = mapOf("key" to 42)
val emptyMap = map - "key"
```

minOf() and maxOf()

These functions can be used to find the lowest and greatest of two or three given values, where values are primitive numbers or `Comparable` objects. There is also an overload of each function that take an additional `Comparator` instance, if you want to compare objects that are not comparable themselves.

```

val list1 = listOf("a", "b")
val list2 = listOf("x", "y", "z")
val minSize = minOf(list1.size, list2.size)
val longestList = maxOf(list1, list2, compareBy { it.size })

```

Array-like List instantiation functions

Similar to the `Array` constructor, there are now functions that create `List` and `MutableList` instances and initialize each element by calling a lambda:

```

val squares = List(10) { index -> index * index }
val mutable = MutableList(10) { 0 }

```

Map.getValue()

This extension on `Map` returns an existing value corresponding to the given key or throws an exception, mentioning which key was not found. If the map was produced with `withDefault`, this function will return the default value instead of throwing an exception.

```

val map = mapOf("key" to 42)
// returns non-nullable Int value 42
val value: Int = map.getValue("key")

val mapWithDefault = map.withDefault { k -> k.length }
// returns 4
val value2 = mapWithDefault.getValue("key2")

// map.getValue("anotherKey") // <- this will throw NoSuchElementException

```

Abstract collections

These abstract classes can be used as base classes when implementing Kotlin collection classes. For implementing read-only collections there are `AbstractCollection`, `AbstractList`, `AbstractSet` and `AbstractMap`, and for mutable collections there are `AbstractMutableCollection`, `AbstractMutableList`, `AbstractMutableSet` and `AbstractMutableMap`. On JVM these abstract mutable collections inherit most of their functionality from JDK's abstract collections.

Array manipulation functions

The standard library now provides a set of functions for element-by-element operations on arrays: comparison (`contentEquals` and `contentDeepEquals`), hash code calculation (`contentHashCode` and `contentDeepHashCode`), and conversion to a string (`contentToString` and `contentDeepToString`). They're supported both for the JVM (where they act as aliases for the corresponding functions in `java.util.Arrays`) and for JS (where the implementation is provided in the Kotlin standard library).

```

val array = arrayOf("a", "b", "c")
println(array.toString()) // JVM implementation: type-and-hash gibberish
println(array.contentToString()) // nicely formatted as list

```

JVM Backend

Java 8 bytecode support

Kotlin has now the option of generating Java 8 bytecode (`-jvm-target 1.8` command line option or the corresponding options in Ant/Maven/Gradle). For now this doesn't change the semantics of the bytecode (in particular, default methods in interfaces and lambdas are generated exactly as in Kotlin 1.0), but we plan to make further use of this later.

Java 8 standard library support

There are now separate versions of the standard library supporting the new JDK APIs added in Java 7 and 8. If you need access to the new APIs, use `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8` maven artifacts instead of the standard `kotlin-stdlib`. These artifacts are tiny extensions on top of `kotlin-stdlib` and they bring it to your project as a transitive dependency.

Parameter names in the bytecode

Kotlin now supports storing parameter names in the bytecode. This can be enabled using the `-java-parameters` command line option.

Constant inlining

The compiler now inlines values of `const val` properties into the locations where they are used.

Mutable closure variables

The box classes used for capturing mutable closure variables in lambdas no longer have volatile fields. This change improves performance, but can lead to new race conditions in some rare usage scenarios. If you're affected by this, you need to provide your own synchronization for accessing the variables.

javax.script support

Kotlin now integrates with the [javax.script API](#) (JSR-223). The API allows to evaluate snippets of code at runtime:

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2")) // Prints out 5
```

See [here](#) for a larger example project using the API.

kotlin.reflect.full

To [prepare for Java 9 support](#), the extension functions and properties in the `kotlin-reflect.jar` library have been moved to the package `kotlin.reflect.full`. The names in the old package (`kotlin.reflect`) are deprecated and will be removed in Kotlin 1.2. Note that the core reflection interfaces (such as `KClass`) are part of the Kotlin standard library, not `kotlin-reflect`, and are not affected by the move.

JavaScript Backend

Unified standard library

A much larger part of the Kotlin standard library can now be used from code compiled to JavaScript. In particular, key classes such as collections (`ArrayList`, `HashMap` etc.), exceptions (`IllegalArgumentException` etc.) and a few others (`StringBuilder`, `Comparator`) are now defined under the `kotlin` package. On the JVM, the names are type aliases for the corresponding JDK classes, and on the JS, the classes are implemented in the Kotlin standard library.

Better code generation

JavaScript backend now generates more statically checkable code, which is friendlier to JS code processing tools, like minifiers, optimisers, linters, etc.

The external modifier

If you need to access a class implemented in JavaScript from Kotlin in a typesafe way, you can write a Kotlin declaration using the `external` modifier. (In Kotlin 1.0, the `@native` annotation was used instead.) Unlike the JVM target, the JS one permits to use external modifier with classes and properties. For example, here's how you can declare the DOM `Node` class:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}
```

Improved import handling

You can now describe declarations which should be imported from JavaScript modules more precisely. If you add the `@JsModule("<module-name>")` annotation on an external declaration it will be properly imported to a module system (either CommonJS or AMD) during the compilation. For example, with CommonJS the declaration will be imported via `require(...)` function. Additionally, if you want to import a declaration either as a module or as a global JavaScript object, you can use the `@JsNonModule` annotation.

For example, here's how you can import JQuery into a Kotlin module:

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

In this case, JQuery will be imported as a module named `jquery`. Alternatively, it can be used as a `$`-object, depending on what module system Kotlin compiler is configured to use.

You can use these declarations in your application like this:

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```


Basics

Basic Types

In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable. Some types are built-in, because their implementation is optimized, but to the user they look like ordinary classes. In this section we describe most of these types: numbers, characters, booleans and arrays.

Numbers

Kotlin handles numbers in a way close to Java, but not exactly the same. For example, there are no implicit widening conversions for numbers, and literals are slightly different in some cases.

Kotlin provides the following built-in types representing numbers (this is close to Java):

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

Note that characters are not numbers in Kotlin.

Literal Constants

There are the following kinds of literal constants for integral values:

- Decimals: `123`
 - Longs are tagged by a capital `L`: `123L`
- Hexadecimals: `0x0F`
- Binaries: `0b00001011`

NOTE: Octal literals are not supported.

Kotlin also supports a conventional notation for floating-point numbers:

- Doubles by default: `123.5`, `123.5e10`
- Floats are tagged by `f` or `F`: `123.5f`

Underscores in numeric literals (since 1.1)

You can use underscores to make number constants more readable:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Representation

On the Java platform, numbers are physically stored as JVM primitive types, unless we need a nullable number reference (e.g. `Int?`) or generics are involved. In the latter cases numbers are boxed.

Note that boxing of numbers does not preserve identity:

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

On the other hand, it preserves equality:

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

Explicit Conversions

Due to different representations, smaller types are not subtypes of bigger ones. If they were, we would have troubles of the following sort:

```
// Hypothetical code, does not actually compile:
val a: Int? = 1 // A boxed Int (java.lang.Integer)
val b: Long? = a // implicit conversion yields a boxed Long (java.lang.Long)
print(a == b) // Surprise! This prints "false" as Long's equals() check for other part to be Long as well
```

So not only identity, but even equality would have been lost silently all over the place.

As a consequence, smaller types are NOT implicitly converted to bigger types. This means that we cannot assign a value of type `Byte` to an `Int` variable without an explicit conversion

```
val b: Byte = 1 // OK, literals are checked statically
val i: Int = b // ERROR
```

We can use explicit conversions to widen numbers

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

Every number type supports the following conversions:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

Absence of implicit conversions is rarely noticeable because the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example

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

Operations

Kotlin supports the standard set of arithmetical operations over numbers, which are declared as members of appropriate classes (but the compiler optimizes the calls down to the corresponding instructions). See [Operator overloading](#).

As of bitwise operations, there're no special characters for them, but just named functions that can be called in infix form, for example:

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

Here is the complete list of bitwise operations (available for `Int` and `Long` only):

- `shl(bits)` - signed shift left (Java's `<<`)
- `shr(bits)` - signed shift right (Java's `>>`)
- `ushr(bits)` - unsigned shift right (Java's `>>>`)
- `and(bits)` - bitwise and
- `or(bits)` - bitwise or
- `xor(bits)` - bitwise xor
- `inv()` - bitwise inversion

Characters

Characters are represented by the type `Char`. They can not be treated directly as numbers

```
fun check(c: Char) {  
    if (c == 1) { // ERROR: incompatible types  
        // ...  
    }  
}
```

Character literals go in single quotes: `'1'`. Special characters can be escaped using a backslash. The following escape sequences are supported: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` and `\$`. To encode any other character, use the Unicode escape sequence syntax: `'\uFF00'`.

We can explicitly convert a character to an `Int` number:

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

Like numbers, characters are boxed when a nullable reference is needed. Identity is not preserved by the boxing operation.

Booleans

The type `Boolean` represents booleans, and has two values: `true` and `false`.

Booleans are boxed if a nullable reference is needed.

Built-in operations on booleans include

- `||` - lazy disjunction
- `&&` - lazy conjunction
- `!` - negation

Arrays

Arrays in Kotlin are represented by the `Array` class, that has `get` and `set` functions (that turn into `[]` by operator overloading conventions), and `size` property, along with a few other useful member functions:

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

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

To create an array, we can use a library function `arrayOf()` and pass the item values to it, so that `arrayOf(1, 2, 3)` creates an array `[1, 2, 3]`. Alternatively, the `arrayOfNulls()` library function can be used to create an array of a given size filled with null elements.

Another option is to use a factory function that takes the array size and the function that can return the initial value of each array element given its index:

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

As we said above, the `[]` operation stands for calls to member functions `get()` and `set()`.

Note: unlike Java, arrays in Kotlin are invariant. This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure (but you can use `Array<out Any>`, see [Type Projections](#)).

Kotlin also has specialized classes to represent arrays of primitive types without boxing overhead: `ByteArray`, `ShortArray`, `IntArray` and so on. These classes have no inheritance relation to the `Array` class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

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

Strings

Strings are represented by the type `String`. Strings are immutable. Elements of a string are characters that can be accessed by the indexing operation: `s[i]`. A string can be iterated over with a `for`-loop:

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

String Literals

Kotlin has two types of string literals: escaped strings that may have escaped characters in them and raw strings that can contain newlines and arbitrary text. An escaped string is very much like a Java string:

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

Escaping is done in the conventional way, with a backslash. See [Characters](#) above for the list of supported escape sequences.

A raw string is delimited by a triple quote (`"""`), contains no escaping and can contain newlines and any other characters:

```
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(">")`.

String Templates

Strings may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign (\$) and consists of either a simple name:

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

or an arbitrary expression in curly braces:

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

Templates are supported both inside raw strings and inside escaped strings. If you need to represent a literal \$ character in a raw string (which doesn't support backslash escaping), you can use the following syntax:

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

Packages

A source file may start with a package declaration:

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

All the contents (such as classes and functions) of the source file are contained by the package declared. So, in the example above, the full name of `baz()` is `foo.bar.baz`, and the full name of `Goo` is `foo.bar.Goo`.

If the package is not specified, the contents of such a file belong to "default" package that has no name.

Default Imports

A number of packages are imported into every Kotlin file by default:

- `kotlin.*`
- `kotlin.annotation.*`
- `kotlin.collections.*`
- `kotlin.comparisons.*` (since 1.1)
- `kotlin.io.*`
- `kotlin.ranges.*`
- `kotlin.sequences.*`
- `kotlin.text.*`

Additional packages are imported depending on the target platform:

- JVM:
 - `java.lang.*`
 - `kotlin.jvm.*`
- JS:
 - `kotlin.js.*`

Imports

Apart from the default imports, each file may contain its own import directives. Syntax for imports is described in the [grammar](#).

We can import either a single name, e.g.

```
import foo.Bar // Bar is now accessible without qualification
```

or all the accessible contents of a scope (package, class, object etc):

```
import foo.* // everything in 'foo' becomes accessible
```

If there is a name clash, we can disambiguate by using `as` keyword to locally rename the clashing entity:

```
import foo.Bar // Bar is accessible
import bar.Bar as bBar // bBar stands for 'bar.Bar'
```

The `import` keyword is not restricted to importing classes; you can also use it to import other declarations:

- top-level functions and properties;

- functions and properties declared in [object declarations](#);
- [enum constants](#)

Unlike Java, Kotlin does not have a separate "import static" syntax; all of these declarations are imported using the regular `import` keyword.

Visibility of Top-level Declarations

If a top-level declaration is marked `private`, it is private to the file it's declared in (see [Visibility Modifiers](#)).

Control Flow

If Expression

In Kotlin, `if` is an expression, i.e. it returns a value. Therefore there is no ternary operator (condition ? then : else), because ordinary `if` works fine in this role.

```
// Traditional usage
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

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

`if` branches can be blocks, and the last expression is the value of a block:

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

If you're using `if` as an expression rather than a statement (for example, returning its value or assigning it to a variable), the expression is required to have an `else` branch.

See the [grammar for if](#).

When Expression

`when` replaces the switch operator of C-like languages. In the simplest form it looks like this

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

`when` matches its argument against all branches sequentially until some branch condition is satisfied. `when` can be used either as an expression or as a statement. If it is used as an expression, the value of the satisfied branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. (Just like with `if`, each branch can be a block, and its value is the value of the last expression in the block.)

The `else` branch is evaluated if none of the other branch conditions are satisfied. If `when` is used as an expression, the `else` branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions.

If many cases should be handled in the same way, the branch conditions may be combined with a comma:


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

We can use arbitrary expressions (not only constants) as branch conditions

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

We can also check a value for being `in` or `!in` a [range](#) or a collection:

```
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")
}
```

Another possibility is to check that a value `is` or `!is` of a particular type. Note that, due to [smart casts](#), you can access the methods and properties of the type without any extra checks.

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

`when` can also be used as a replacement for an `if-else if` chain. If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true:

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

See the [grammar for when](#).

For Loops

`for` loop iterates through anything that provides an iterator. The syntax is as follows:

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

The body can be a block.

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

As mentioned before, `for` iterates through anything that provides an iterator, i.e.

- has a member- or extension-function `iterator()`, whose return type
- has a member- or extension-function `next()`, and
- has a member- or extension-function `hasNext()` that returns `Boolean`.

All of these three functions need to be marked as `operator`.

A `for` loop over an array is compiled to an index-based loop that does not create an iterator object.

If you want to iterate through an array or a list with an index, you can do it this way:

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

Note that this "iteration through a range" is compiled down to optimal implementation with no extra objects created.

Alternatively, you can use the `withIndex` library function:

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

See the [grammar for for](#).

While Loops

`while` and `do..while` work as usual

```
while (x > 0) {  
    x--  
}  
  
do {  
    val y = retrieveData()  
} while (y != null) // y is visible here!
```

See the [grammar for while](#).

Break and continue in loops

Kotlin supports traditional `break` and `continue` operators in loops. See [Returns and jumps](#).

Returns and Jumps

Kotlin has three structural jump expressions:

- **return**. By default returns from the nearest enclosing function or [anonymous function](#).
- **break**. Terminates the nearest enclosing loop.
- **continue**. Proceeds to the next step of the nearest enclosing loop.

All of these expressions can be used as part of larger expressions:

```
val s = person.name ?: return
```

The type of these expressions is the [Nothing type](#).

Break and Continue Labels

Any expression in Kotlin may be marked with a [label](#). Labels have the form of an identifier followed by the `@` sign, for example: `abc@`, `fooBar@` are valid labels (see the [grammar](#)). To label an expression, we just put a label in front of it

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

Now, we can qualify a **break** or a **continue** with a label:

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

A **break** qualified with a label jumps to the execution point right after the loop marked with that label. A **continue** proceeds to the next iteration of that loop.

Return at Labels

With function literals, local functions and object expression, functions can be nested in Kotlin. Qualified **returns** allow us to return from an outer function. The most important use case is returning from a lambda expression. Recall that when we write this:

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

The **return**-expression returns from the nearest enclosing function, i.e. `foo`. (Note that such non-local returns are supported only for lambda expressions passed to [inline functions](#).) If we need to return from a lambda expression, we have to label it and qualify the **return**:

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

Now, it returns only from the lambda expression. Oftentimes it is more convenient to use implicit labels: such a label has the same name as the function to which the lambda is passed.

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

Alternatively, we can replace the lambda expression with an [anonymous function](#). A `return` statement in an anonymous function will return from the anonymous function itself.

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

When returning a value, the parser gives preference to the qualified return, i.e.

```
return@a 1
```

means "return 1 at label @a" and not "return a labeled expression (@a 1)".

Classes and Objects

Classes and Inheritance

Classes

Classes in Kotlin are declared using the keyword `class`:

```
class Invoice {  
}
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor etc.) and the class body, surrounded by curly braces. Both the header and the body are optional; if the class has no body, curly braces can be omitted.

```
class Empty
```

Constructors

A class in Kotlin can have a **primary constructor** and one or more **secondary constructors**. The primary constructor is part of the class header: it goes after the class name (and optional type parameters).

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

If the primary constructor does not have any annotations or visibility modifiers, the `constructor` keyword can be omitted:

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

The primary constructor cannot contain any code. Initialization code can be placed in **initializer blocks**, which are prefixed with the `init` keyword:

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

Note that parameters of the primary constructor can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

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

In fact, for declaring properties and initializing them from the primary constructor, Kotlin has a concise syntax:

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

Much the same way as regular properties, the properties declared in the primary constructor can be mutable ([var](#)) or read-only ([val](#)).

If the constructor has annotations or visibility modifiers, the [constructor](#) keyword is required, and the modifiers go before it:

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

For more details, see [Visibility Modifiers](#).

Secondary Constructors

The class can also declare **secondary constructors**, which are prefixed with [constructor](#):

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

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the [this](#) keyword:

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

If a non-abstract class does not declare any constructors (primary or secondary), it will have a generated primary constructor with no arguments. The visibility of the constructor will be public. If you do not want your class to have a public constructor, you need to declare an empty primary constructor with non-default visibility.

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

NOTE: On the JVM, if all of the parameters of the primary constructor have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors.

```
class Customer(val customerName: String = "")
```

Creating instances of classes

To create an instance of a class, we call the constructor as if it were a regular function:

```
val invoice = Invoice()

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

Note that Kotlin does not have a [new](#) keyword.

Creating instances of nested, inner and anonymous inner classes is described in [Nested classes](#).

Class Members

Classes can contain

- Constructors and initializer blocks
- [Functions](#)
- [Properties](#)
- [Nested and Inner Classes](#)
- [Object Declarations](#)

Inheritance

All classes in Kotlin have a common superclass `Any`, that is a default super for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

`Any` is not `java.lang.Object`; in particular, it does not have any members other than `equals()`, `hashCode()` and `toString()`. Please consult the [Java interoperability](#) section for more details.

To declare an explicit supertype, we place the type after a colon in the class header:

```
open class Base(p: Int)

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

If the class has a primary constructor, the base type can (and must) be initialized right there, using the parameters of the primary constructor.

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the `super` keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

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

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

The `open` annotation on a class is the opposite of Java's `final`: it allows others to inherit from this class. By default, all classes in Kotlin are final, which corresponds to [Effective Java](#), Item 17: *Design and document for inheritance or else prohibit it*.

Overriding Methods

As we mentioned before, we stick to making things explicit in Kotlin. And unlike Java, Kotlin requires explicit annotations for overridable members (we call them *open*) and for overrides:

```
open class Base {
    open fun v() {}
    fun nv() {}
}

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

The `override` annotation is required for `Derived.v()`. If it were missing, the compiler would complain. If there is no `open` annotation on a function, like `Base.nv()`, declaring a method with the same signature in a subclass is illegal, either with `override` or without it. In a final class (e.g. a class with no `open` annotation), open members are prohibited.

A member marked `override` is itself open, i.e. it may be overridden in subclasses. If you want to prohibit re-overriding, use `final`:

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

Overriding Properties

Overriding properties works in a similar way to overriding methods; properties declared on a superclass that are then redeclared on a derived class must be prefaced with `override`, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a getter method.

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

class Bar1 : Foo() {
    override val x: Int = ...
}
```

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.

Note that you can use the `override` keyword as part of the property declaration in a primary constructor.

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

class Bar2 : Foo {
    override var count: Int = 0
}
```

Overriding Rules

In Kotlin, implementation inheritance is regulated by the following rule: if a class inherits many implementations of the same member from its immediate superclasses, it must override this member and provide its own implementation (perhaps, using one of the inherited ones). To denote the supertype from which the inherited implementation is taken, we use `super` qualified by the supertype name in angle brackets, e.g. `super<Base>`:

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

interface B {
    fun f() { print("B") } // interface members are 'open' by default
    fun b() { print("b") }
}

class C() : A(), B {
    // The compiler requires f() to be overridden:
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}
```

It's fine to inherit from both `A` and `B`, and we have no problems with `a()` and `b()` since `C` inherits only one implementation of each of these functions. But for `f()` we have two implementations inherited by `C`, and thus we have to override `f()` in `C` and provide our own implementation that eliminates the ambiguity.

Abstract Classes

A class and some of its members may be declared [abstract](#). An abstract member does not have an implementation in its class. Note that we do not need to annotate an abstract class or function with `open` – it goes without saying.

We can override a non-abstract open member with an abstract one

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

Companion Objects

In Kotlin, unlike Java or C#, classes do not have static methods. In most cases, it's recommended to simply use package-level functions instead.

If you need to write a function that can be called without having a class instance but needs access to the internals of a class (for example, a factory method), you can write it as a member of an [object declaration](#) inside that class.

Even more specifically, if you declare a [companion object](#) inside your class, you'll be able to call its members with the same syntax as calling static methods in Java/C#, using only the class name as a qualifier.

Properties and Fields

Declaring Properties

Classes in Kotlin can have properties. These can be declared as mutable, using the `var` keyword or read-only using the `val` keyword.

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

To use a property, we simply refer to it by name, as if it were a field in 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 and Setters

The full syntax for declaring a property is

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

The initializer, getter and setter are optional. Property type is optional if it can be inferred from the initializer (or from the getter return type, as shown below).

Examples:

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

The full syntax of a read-only property declaration differs from a mutable one in two ways: it starts with `val` instead of `var` and does not allow a setter:

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

We can write custom accessors, very much like ordinary functions, right inside a property declaration. Here's an example of a custom getter:

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

A custom setter looks like this:

```

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

```

By convention, the name of the setter parameter is `value`, but you can choose a different name if you prefer.

Since Kotlin 1.1, you can omit the property type if it can be inferred from the getter:

```

val isEmpty get() = this.size == 0 // has type Boolean

```

If you need to change the visibility of an accessor or to annotate it, but don't need to change the default implementation, you can define the accessor without defining its body:

```

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

```

Backing Fields

Classes in Kotlin cannot have fields. However, sometimes it is necessary to have a backing field when using custom accessors. For these purposes, Kotlin provides an automatic backing field which can be accessed using the `field` identifier:

```

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

```

The `field` identifier can only be used in the accessors of the property.

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.

For example, in the following case there will be no backing field:

```

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

```

Backing Properties

If you want to do something that does not fit into this "implicit backing field" scheme, you can always fall back to having a *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")
    }

```

In all respects, this is just the same as in Java since access to private properties with default getters and setters is optimized so that no function call overhead is introduced.

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 Properties

See [Overriding Properties](#)

Delegated Properties

The most common kind of properties simply reads from (and maybe writes to) a backing field. On the other hand, with custom getters and setters one can implement any behaviour of a property. Somewhere in between, there are certain common patterns of how a property may work. A few examples: lazy values, reading from a map by a given key, accessing a database, notifying listener on access, etc.

Such common behaviours can be implemented as libraries using [delegated properties](#).

Interfaces

Interfaces in Kotlin are very similar to Java 8. They can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties but these need to be abstract or to provide accessor implementations.

An interface is defined using the keyword `interface`

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

Implementing Interfaces

A class or object can implement one or more interfaces

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

Properties in Interfaces

You can declare properties in interfaces. A property declared in an interface can either be abstract, or it can provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them.

```
interface MyInterface {  
    val prop: Int // abstract  
  
    val propertyWithImplementation: String  
    get() = "foo"  
  
    fun foo() {  
        print(prop)  
    }  
}  
  
class Child : MyInterface {  
    override val prop: Int = 29  
}
```

Resolving overriding conflicts

When we declare many types in our supertype list, it may appear that we inherit more than one implementation of the same method. For example

```

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()
    }

    override fun bar() {
        super<B>.bar()
    }
}

```

Interfaces *A* and *B* both declare functions *foo()* and *bar()*. Both of them implement *foo()*, but only *B* implements *bar()* (*bar()* is not marked abstract in *A*, because this is the default for interfaces, if the function has no body). Now, if we derive a concrete class *C* from *A*, we, obviously, have to override *bar()* and provide an implementation.

However, if we derive *D* from *A* and *B*, we need to implement all the methods which we have inherited from multiple interfaces, and to specify how exactly *D* should implement them. This rule applies both to methods for which we've inherited a single implementation (*bar()*) and multiple implementations (*foo()*).

Visibility Modifiers

Classes, objects, interfaces, constructors, functions, properties and their setters can have *visibility modifiers*. (Getters always have the same visibility as the property.) There are four visibility modifiers in Kotlin: `private`, `protected`, `internal` and `public`. The default visibility, used if there is no explicit modifier, is `public`.

Below please find explanations of these for different type of declaring scopes.

Packages

Functions, properties and classes, objects and interfaces can be declared on the "top-level", i.e. directly inside a package:

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

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

- If you do not specify any visibility modifier, `public` is used by default, which means that your declarations will be visible everywhere;
- If you mark a declaration `private`, it will only be visible inside the file containing the declaration;
- If you mark it `internal`, it is visible everywhere in the same [module](#);
- `protected` is not available for top-level declarations.

Examples:

```
// 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
```

Classes and Interfaces

For members declared inside a class:

- `private` means visible inside this class only (including all its members);
- `protected` — same as `private` + visible in subclasses too;
- `internal` — any client *inside this module* who sees the declaring class sees its `internal` members;
- `public` — any client who sees the declaring class sees its `public` members.

NOTE for Java users: outer class does not see private members of its inner classes in Kotlin.

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

Examples:

```

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
}

```

Constructors

To specify a visibility of the primary constructor of a class, use the following syntax (note that you need to add an explicit `constructor` keyword):

```

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

```

Here the constructor is private. By default, all constructors are `public`, which effectively amounts to them being visible everywhere where the class is visible (i.e. a constructor of an `internal` class is only visible within the same module).

Local declarations

Local variables, functions and classes can not have visibility modifiers.

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 the Ant task.

Extensions

Kotlin, similar to C# and Gosu, provides the ability to extend a class with new functionality without having to inherit from the class or use any type of design pattern such as Decorator. This is done via special declarations called *extensions*. Kotlin supports *extension functions* and *extension properties*.

Extension Functions

To declare an extension function, we need to prefix its name with a *receiver type*, i.e. the type being extended. The following adds a `swap` function to `MutableList<Int>`:

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

The `this` keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot). Now, we can call such a function on any `MutableList<Int>`:

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

Of course, this function makes sense for any `MutableList<T>`, and we can make it generic:

```
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
}
```

We declare the generic type parameter before the function name for it to be available in the receiver type expression. See [Generic functions](#).

Extensions are resolved **statically**

Extensions do not actually modify classes they extend. By defining an extension, you do not insert new members into a class, but merely make new functions callable with the dot-notation on variables of this type.

We would like to emphasize that extension functions are dispatched **statically**, i.e. they are not virtual by receiver type. 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") }
```

If we call `c.foo()` of any `c` of type `C`, it will print "member", not "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 Receiver

Note that extensions can be defined with a nullable receiver type. Such extensions can be called on an object variable even if its value is null, and can check for `this == null` inside the body. This is what allows you to call `toString()` in Kotlin without checking for null: the check happens inside the extension function.

```
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()
}
```

Extension Properties

Similarly to functions, Kotlin supports extension properties:

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

Note that, since extensions do not actually insert members into classes, there's no efficient way for an extension property to have a [backing field](#). This is why **initializers are not allowed for extension properties**. Their behavior can only be defined by explicitly providing getters/setters.

Example:

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

Companion Object Extensions

If a class has a [companion object](#) defined, you can also define extension functions and properties for the companion object:

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

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

Just like regular members of the companion object, they can be called using only the class name as the qualifier:

```
MyClass.foo()
```

Scope of Extensions

Most of the time we define extensions on the top level, i.e. directly under packages:

```
package foo.bar

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

To use such an extension outside its declaring package, we need to import it at the call site:

```
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()
}
```

See [Imports](#) for more information.

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

In Java, we are used to classes named `"*Utils"`: `FileUtils`, `StringUtils` and so on. The famous `java.util.Collections` belongs to the same breed. And the unpleasant part about these Utils-classes is that the code that uses them looks like this:

```

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

```

Those class names are always getting in the way. We can use static imports and get this:

```

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

```

This is a little better, but we have no or little help from the powerful code completion of the IDE. It would be so much better if we could say

```

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

```

But we don't want to implement all the possible methods inside the class `List`, right? This is where extensions help us.

Data Classes

We frequently create a class to do nothing but hold data. In such a class some standard functionality is often mechanically derivable from the data. In Kotlin, this is called a *data class* and is marked as `data`:

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

The compiler automatically derives the following members from all properties declared in the primary constructor:

- `equals()` / `hashCode()` pair,
- `toString()` of the form `"User(name=John, age=42)"`,
- [componentN\(\) functions](#) corresponding to the properties in their order of declaration,
- `copy()` function (see below).

If any of these functions is explicitly defined in the class body or inherited from the base types, it will not be generated.

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;
- (before 1.1) Data classes may only implement interfaces.

Since 1.1, data classes may extend other classes (see [Sealed classes](#) for examples).

On the JVM, if the generated class needs to have a parameterless constructor, default values for all properties have to be specified (see [Constructors](#)).

```
data class User(val name: String = "", val age: Int = 0)
```

Copying

It's often the case that we need to copy an object altering *some* of its properties, but keeping the rest unchanged. This is what `copy()` function is generated for. For the `User` class above, its implementation would be as follows:

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

This allows us to write

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

Data Classes and Destructuring Declarations

Component functions generated for data classes enable their use in [destructuring declarations](#):

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

Standard Data Classes

The standard library provides `Pair` and `Triple`. In most cases, though, named data classes are a better design choice, because they make the code more readable by providing meaningful names for properties.

Sealed Classes

Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type. They are, in a sense, an extension of enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances which can contain state.

To declare a sealed class, you put the `sealed` modifier before the name of the class. A sealed class can have subclasses, but all of them must be nested inside the declaration of the sealed class itself.

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

Note that classes which extend subclasses of a sealed class (indirect inheritors) can be placed anywhere, not necessarily inside the declaration of the sealed class.

The key benefit of using sealed classes comes into play when you use them in a [when expression](#). If it's possible to verify that the statement covers all cases, you don't need to add an `else` clause to the statement.

```
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  
    // the `else` clause is not required because we've covered all the cases  
}
```

Relaxed Rules for Sealed Classes (since 1.1)

Subclasses in the Same File

Since 1.1 you can declare the subclasses of the `sealed` class on the top-level, with only restriction that they should be located in the same file as the parent class.

Sealed Classes and Data Classes

Data classes can extend other classes, including `sealed` classes, which makes the hierarchy more usable.

With all the newly supported features, you can rewrite the `Expr` class hierarchy in the following way:

```
sealed class Expr  
data class Const(val number: Double) : Expr()  
data class Sum(val e1: Expr, val e2: Expr) : Expr()  
object NotANumber : Expr()  
  
fun eval(expr: Expr): Double = when (expr) {  
    is Const -> expr.number  
    is Sum -> eval(expr.e1) + eval(expr.e2)  
    NotANumber -> Double.NaN  
}
```

Generics

As in Java, classes in Kotlin may have type parameters:

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

In general, to create an instance of such a class, we need to provide the type arguments:

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

But if the parameters may be inferred, e.g. from the constructor arguments or by some other means, one is allowed to omit the type arguments:

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

Variance

One of the most tricky parts of Java's type system is wildcard types (see [Java Generics FAQ](#)). And Kotlin doesn't have any. Instead, it has two other things: declaration-site variance and type projections.

First, let's think about why Java needs those mysterious wildcards. The problem is explained in [Effective Java](#), Item 28: *Use bounded wildcards to increase API flexibility*. First, generic types in Java are **invariant**, meaning that `List<String>` is **not** a subtype of `List<Object>`. Why so? If List was not **invariant**, it would have been no better than Java's arrays, since the following code would have compiled and caused an exception at runtime:

```
// 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
```

So, Java prohibits such things in order to guarantee run-time safety. But this has some implications. For example, consider the `addAll()` method from `Collection` interface. What's the signature of this method? Intuitively, we'd put it this way:

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

But then, we would not be able to do the following simple thing (which is perfectly safe):

```
// 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>  
}
```

(In Java, we learned this lesson the hard way, see [Effective Java](#), Item 25: *Prefer lists to arrays*)

That's why the actual signature of `addAll()` is the following:

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

The **wildcard type argument** `? extends T` indicates that this method accepts a collection of objects of *some subtype of T*, not `T` itself. This means that we can safely **read** `T`'s from items (elements of this collection are instances of a subclass of `T`), but **cannot write** to it since we do not know what objects comply to that unknown subtype of `T`. In return for this limitation, we have the desired behaviour: `Collection<String>` is a subtype of `Collection<? extends Object>`. In "clever words", the wildcard with an **extends-bound** (**upper bound**) makes the type **covariant**.

The key to understanding why this trick works is rather simple: if you can only **take** items from a collection, then using a collection of `String`s and reading `Object`s from it is fine. Conversely, if you can only *put* items into the collection, it's OK to take a collection of `Object`s and put `String`s into it: in Java we have `List<? super String>` a **supertype** of `List<Object>`.

The latter is called **contravariance**, and you can only call methods that take `String` as an argument on `List<? super String>` (e.g., you can call `add(String)` or `set(int, String)`), while if you call something that returns `T` in `List<T>`, you don't get a `String`, but an `Object`.

Joshua Bloch calls those objects you only **read** from **Producers**, and those you only **write** to **Consumers**. He recommends: "*For maximum flexibility, use wildcard types on input parameters that represent producers or consumers*", and proposes the following mnemonic:

PECS stands for Producer-Extends, Consumer-Super.

NOTE: if you use a producer-object, say, `List<? extends Foo>`, you are not allowed to call `add()` or `set()` on this object, but this does not mean that this object is **immutable**: for example, nothing prevents you from calling `clear()` to remove all items from the list, since `clear()` does not take any parameters at all. The only thing guaranteed by wildcards (or other types of variance) is **type safety**. Immutability is a completely different story.

Declaration-site variance

Suppose we have a generic interface `Source<T>` that does not have any methods that take `T` as a parameter, only methods that return `T`:

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

Then, it would be perfectly safe to store a reference to an instance of `Source<String>` in a variable of type `Source<Object>` - there are no consumer-methods to call. But Java does not know this, and still prohibits it:

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

To fix this, we have to declare objects of type `Source<? extends Object>`, which is sort of meaningless, because we can call all the same methods on such a variable as before, so there's no value added by the more complex type. But the compiler does not know that.

In Kotlin, there is a way to explain this sort of thing to the compiler. This is called **declaration-site variance**: we can annotate the **type parameter** `T` of `Source` to make sure that it is only **returned** (produced) from members of `Source<T>`, and never consumed. To do this we provide the **out** modifier:

```
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
    // ...
}
```


The general rule is: when a type parameter `T` of a class `C` is declared **out**, it may occur only in **out**-position in the members of `C`, but in return `C<Base>` can safely be a supertype of `C<Derived>`.

In "clever words" they say that the class `C` is **covariant** in the parameter `T`, or that `T` is a **covariant** type parameter. You can think of `C` as being a **producer** of `T`'s, and NOT a **consumer** of `T`'s.

The **out** modifier is called a **variance annotation**, and since it is provided at the type parameter declaration site, we talk about **declaration-site variance**. This is in contrast with Java's **use-site variance** where wildcards in the type usages make the types covariant.

In addition to **out**, Kotlin provides a complementary variance annotation: **in**. It makes a type parameter **contravariant**: it can only be consumed and never produced. A good example of a contravariant class is `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!
}
```

We believe that the words **in** and **out** are self-explaining (as they were successfully used in C# for quite some time already), thus the mnemonic mentioned above is not really needed, and one can rephrase it for a higher purpose:

The Existential Transformation: Consumer in, Producer out! :-)

Type projections

Use-site variance: Type projections

It is very convenient to declare a type parameter `T` as *out* and have no trouble with subtyping on the use site. Yes, it is, when the class in question **can** actually be restricted to only return `T`'s, but what if it can't? A good example of this is `Array`:

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

This class cannot be either co- or contravariant in `T`. And this imposes certain inflexibilities. Consider the following function:

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

This function is supposed to copy items from one array to another. Let's try to apply it in practice:

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

Here we run into the same familiar problem: `Array<T>` is **invariant** in `T`, thus neither of `Array<Int>` and `Array<Any>` is a subtype of the other. Why? Again, because `copy` **might** be doing bad things, i.e. it might attempt to **write**, say, a `String` to `from`, and if we actually passed an array of `Int` there, a `ClassCastException` would have been thrown sometime later.

Then, the only thing we want to ensure is that `copy()` does not do any bad things. We want to prohibit it from **writing** to `from`, and we can:

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

What has happened here is called **type projection**: we said that `from` is not simply an array, but a restricted (**projected**) one: we can only call those methods that return the type parameter `T`, in this case it means that we can only call `get()`. This is our approach to **use-site variance**, and corresponds to Java's `Array<? extends Object>`, but in a slightly simpler way.

You can project a type with **in** as well:

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

`Array<in String>` corresponds to Java's `Array<? super String>`, i.e. you can pass an array of `CharSequence` or an array of `Object` to the `fill()` function.

Star-projections

Sometimes you want to say that you know nothing about the type argument, but still want to use it in a safe way. The safe way here is to define such a projection of the generic type, that every concrete instantiation of that generic type would be a subtype of that projection.

Kotlin provides so called **star-projection** syntax for this:

- 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?>`.

Note: star-projections are very much like Java's raw types, but safe.

Generic functions

Not only classes can have type parameters. Functions can, too. Type parameters are placed before the name of the function:

```
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)
```

Generic constraints

The set of all possible types that can be substituted for a given type parameter may be restricted by **generic constraints**.

Upper bounds

The most common type of constraint is an **upper bound** that corresponds to Java's *extends* keyword:

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

The type specified after a colon is the **upper bound**: only a subtype of `Comparable<T>` may be substituted for `T`. For example

```
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>>
```

The default upper bound (if none specified) is `Any?`. Only one upper bound can be specified inside the angle brackets. If the same type parameter needs more than one upper bound, we need a separate **where**-clause:

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

Nested Classes

Classes can be nested in other classes

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

Inner classes

A class may be marked as `inner` to be able to access members of outer class. Inner classes carry a reference to an object of an outer class:

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

See [Qualified this expressions](#) to learn about disambiguation of `this` in inner classes.

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 Classes

The most basic usage of enum classes is implementing type-safe enums

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

Each enum constant is an object. Enum constants are separated with commas.

Initialization

Since each enum is an instance of the enum class, they can be initialized

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

Anonymous Classes

Enum constants can also declare their own anonymous classes

```
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>
```

The `valueOf()` method throws an `IllegalArgumentException` if the specified name does not match any of the enum constants defined in the class.

Since Kotlin 1.1, it's possible to access the constants in an enum class in a generic way, using the `enumValues<T>()` and `enumValueOf<T>()` functions:

```
enum class RGB { RED, GREEN, BLUE }  
  
inline fun <reified T : Enum<T>> printAllValues() {  
    print(enumValues<T>().joinToString { it.name })  
}  
  
printAllValues<RGB>() // prints RED, GREEN, BLUE
```

Every enum constant has properties to obtain its name and position in the enum class declaration:

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

The enum constants also implement the [Comparable](#) interface, with the natural order being the order in which they are defined in the enum class.

Object Expressions and Declarations

Sometimes we need to create an object of a slight modification of some class, without explicitly declaring a new subclass for it. Java handles this case with *anonymous inner classes*. Kotlin slightly generalizes this concept with *object expressions* and *object declarations*.

Object expressions

To create an object of an anonymous class that inherits from some type (or types), we write:

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

If a supertype has a constructor, appropriate constructor parameters must be passed to it. Many supertypes may be specified as a comma-separated list after the colon:

```
open class A(x: Int) {  
    public open val y: Int = x  
}  
  
interface B {...}  
  
val ab: A = object : A(1), B {  
    override val y = 15  
}
```

If, by any chance, we need "just an object", with no nontrivial supertypes, we can simply say:

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

Note that anonymous objects can be used as types only in local and private declarations. If you use an anonymous object as a return type of a public function or the type of a public property, the actual type of that function or property will be the declared supertype of the anonymous object, or `Any` if you didn't declare any supertype. Members added in the anonymous object will not be accessible.

```

class C {
    // Private function, so the return type is the anonymous object type
    private fun foo() = object {
        val x: String = "x"
    }

    // Public function, so the return type is Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x          // Works
        val x2 = publicFoo().x    // ERROR: Unresolved reference 'x'
    }
}

```

Just like Java's anonymous inner classes, code in object expressions can access variables from the enclosing scope. (Unlike Java, this is not restricted to final variables.)

```

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++
        }
    })
    // ...
}

```

Object declarations

[Singleton](#) is a very useful pattern, and Kotlin (after Scala) makes it easy to declare singletons:

```

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

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

```

- This is called an *object declaration*, and it always has a name following the `object` keyword. Just like a variable declaration, an object declaration is not an expression, and cannot be used on the right hand side of an assignment statement.

To refer to the object, we use its name directly:

```

DataProviderManager.registerDataProvider(...)

```

Such objects can have supertypes:


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

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

NOTE: object declarations can't be local (i.e. be nested directly inside a function), but they can be nested into other object declarations or non-inner classes.

Companion Objects

An object declaration inside a class can be marked with the `companion` keyword:

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

Members of the companion object can be called by using simply the class name as the qualifier:

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

The name of the companion object can be omitted, in which case the name `Companion` will be used:

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

Note that, even though the members of companion objects look like static members in other languages, at runtime those are still instance members of real objects, and can, for example, implement interfaces:

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

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

However, on the JVM you can have members of companion objects generated as real static methods and fields, if you use the `@JvmStatic` annotation. See the [Java interoperability](#) section for more details.

Semantic difference between object expressions and declarations

There is one important semantic difference between object expressions and object declarations:

- object expressions are executed (and initialized) **immediately**, where they are used
- object declarations are initialized **lazily**, when accessed for the first time
- a companion object is initialized when the corresponding class is loaded (resolved), matching the semantics of a Java static initializer

Delegation

Class Delegation

The [Delegation pattern](#) has proven to be a good alternative to implementation inheritance, and Kotlin supports it natively requiring zero boilerplate code. A class `Derived` can inherit from an interface `Base` and delegate all of its public methods to a specified object:

```
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  
}
```

The `by`-clause in the supertype list for `Derived` indicates that `b` will be stored internally in objects of `Derived` and the compiler will generate all the methods of `Base` that forward to `b`.

Delegated Properties

There are certain common kinds of properties, that, though we can implement them manually every time we need them, would be very nice to implement once and for all, and put into a library. Examples include

- lazy properties: the value gets computed only upon first access,
- observable properties: listeners get notified about changes to this property,
- storing properties in a map, not in separate field each.

To cover these (and other) cases, Kotlin supports *delegated properties*:

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

The syntax is: `val/var <property name>: <Type> by <expression>`. The expression after `by` is the *delegate*, because `get()` (and `set()`) corresponding to the property will be delegated to its `getValue()` and `setValue()` methods. Property delegates don't have to implement any interface, but they have to provide a `getValue()` function (and `setValue()` — for `var`'s). For example:

```
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.")  
    }  
}
```

When we read from `p` that delegates to an instance of `Delegate`, the `getValue()` function from `Delegate` is called, so that its first parameter is the object we read `p` from and the second parameter holds a description of `p` itself (e.g. you can take its name). For example:

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

This prints

```
Example@33a17727, thank you for delegating 'p' to me!
```

Similarly, when we assign to `p`, the `setValue()` function is called. The first two parameters are the same, and the third holds the value being assigned:

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

This prints

```
NEW has been assigned to 'p' in Example@33a17727.
```

The specification of the requirements to the delegated object can be found [below](#).

Note that since Kotlin 1.1 you can declare a delegated property inside a function or code block, it shouldn't necessarily be a member of a class. Below you can find [the example](#).

Standard Delegates

The Kotlin standard library provides factory methods for several useful kinds of delegates.

Lazy

`lazy()` is a function that takes a lambda and returns an instance of `Lazy<T>` which can serve as a delegate for implementing a lazy property: the first call to `get()` executes the lambda passed to `lazy()` and remembers the result, subsequent calls to `get()` simply return the remembered result.

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

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

This example prints:

```
computed!
Hello
Hello
```

By default, the evaluation of lazy properties is **synchronized**: the value is computed only in one thread, and all threads will see the same value. If the synchronization of initialization delegate is not required, so that multiple threads can execute it simultaneously, pass `LazyThreadSafetyMode.PUBLICATION` as a parameter to the `lazy()` function. And if you're sure that the initialization will always happen on a single thread, you can use `LazyThreadSafetyMode.NONE` mode, which doesn't incur any thread-safety guarantees and the related overhead.

Observable

`Delegates.observable()` takes two arguments: the initial value and a handler for modifications. The handler gets called every time we assign to the property (*after* the assignment has been performed). It has three parameters: a property being assigned to, the old value and the new one:

```
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"
}
```

This example prints:

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

If you want to be able to intercept an assignment and "veto" it, use `vetoable()` instead of `observable()`. The handler passed to the `vetoable` is called *before* the assignment of a new property value has been performed.

Storing Properties in a Map

One common use case is storing the values of properties in a map. This comes up often in applications like parsing JSON or doing other "dynamic" things. In this case, you can use the map instance itself as the delegate for a delegated property.

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

In this example, the constructor takes a map:

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

Delegated properties take values from this map (by the string keys -- names of properties):

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

This works also for `var`'s properties if you use a `MutableMap` instead of read-only `Map`:

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

Local Delegated Properties (since 1.1)

You can declare local variables as delegated properties. For instance, you can make a local variable lazy:

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

The `memoizedFoo` variable will be computed on the first access only. If `someCondition` fails, the variable won't be computed at all.

Property Delegate Requirements

Here we summarize requirements to delegate objects.

For a **read-only** property (i.e. a `val`), a delegate has to provide a function named `getValue` that takes the following parameters:

- `thisRef` — must be the same or a supertype of the *property owner* (for extension properties — the type being extended),
- `property` — must be of type `KProperty<*>` or its supertype,

this function must return the same type as property (or its subtype).

For a **mutable** property (a `var`), a delegate has to *additionally* provide a function named `setValue` that takes the following parameters:

- `thisRef` — same as for `getValue()`,
- `property` — same as for `getValue()`,
- new value — must be of the same type as a property or its supertype.

`getValue()` and/or `setValue()` functions may be provided either as member functions of the delegate class or extension functions. The latter is handy when you need to delegate property to an object which doesn't originally provide these functions. Both of the functions need to be marked with the `operator` keyword.

The delegate class may implement one of the interfaces `ReadOnlyProperty` and `ReadWriteProperty` containing the required `operator` methods. These interfaces are declared in the Kotlin standard library:

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

Translation Rules

Under the hood for every delegated property the Kotlin compiler generates an auxiliary property and delegates to it. For instance, for the property `prop` the hidden property `prop$delegate` is generated, and the code of the accessors simply delegates to this additional property:

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler instead:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
    get() = prop$delegate.getValue(this, this::prop)
    set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

The Kotlin compiler provides all the necessary information about `prop` in the arguments: the first argument `this` refers to an instance of the outer class `C` and `this::prop` is a reflection object of the `KProperty` type describing `prop` itself.

Note that the syntax `this::prop` to refer a [bound callable reference](#) in the code directly is available only since Kotlin 1.1.

Providing a delegate (since 1.1)

By defining the `provideDelegate` operator you can extend the logic of creating the object to which the property implementation is delegated. If the object used on the right hand side of `by` defines `provideDelegate` as a member or extension function, that function will be called to create the property delegate instance.

One of the possible use cases of `provideDelegate` is to check property consistency when the property is created, not only in its getter or setter.

For example, if you want to check the property name before binding, you can write something like this:

```

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
        thisRef: MyUI,
        prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // create delegate
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}

```

The parameters of `provideDelegate` are the same as for `getValue`:

- `thisRef` — must be the same or a supertype of the *property owner* (for extension properties — the type being extended),
- `property` — must be of type `KProperty<*>` or its supertype.

The `provideDelegate` method is called for each property during the creation of the `MyUI` instance, and it performs the necessary validation right away.

Without this ability to intercept the binding between the property and its delegate, to achieve the same functionality you'd have to pass the property name explicitly, which isn't very convenient:

```

// Checking the property name without "provideDelegate" functionality
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
    id: ResourceID<T>,
    propertyName: String
): ReadOnlyProperty<MyUI, T> {
    checkProperty(this, propertyName)
    // create delegate
}

```

In the generated code, the `provideDelegate` method is called to initialize the auxiliary `prop$delegate` property. Compare the generated code for the property declaration `val prop: Type by MyDelegate()` with the generated code [above](#) (when the `provideDelegate` method is not present):

```

class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler
// when the 'provideDelegate' function is available:
class C {
    // calling "provideDelegate" to create the additional "delegate" property
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    val prop: Type
        get() = prop$delegate.getValue(this, this::prop)
}

```

Note that the `provideDelegate` method affects only the creation of the auxiliary property and doesn't affect the code generated for getter or setter.

Functions and Lambdas

Functions

Function Declarations

Functions in Kotlin are declared using the `fun` keyword

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

Function Usage

Calling functions uses the traditional approach

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

Calling member functions uses the dot notation

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

Infix notation

Functions can also be called using infix notations when

- They are member functions or [extension functions](#)
- They have a single parameter
- 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)
```

Parameters

Function parameters are defined using Pascal notation, i.e. *name: type*. Parameters are separated using commas. Each parameter must be explicitly typed.

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

Default Arguments

Function parameters can have default values, which are used when a corresponding argument is omitted. This allows for a reduced number of overloads compared to other languages.

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

Default values are defined using the = after type along with the value.

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  
}
```

Named Arguments

Function parameters can be named when calling functions. This is very convenient when a function has a high number of parameters or default ones.

Given the following function

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

we could call this using default arguments

```
reformat(str)
```

However, when calling it with non-default, the call would look something like

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

With named arguments we can make the code much more readable

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

and if we do not need all arguments

```
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-returning functions

If a function does not return any useful value, its return type is `Unit`. `Unit` is a type with only one value - `Unit`. This value does not have to be returned explicitly

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

The `Unit` return type declaration is also optional. The above code is equivalent to

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

Single-Expression functions

When a function returns a single expression, the curly braces can be omitted and the body is specified after a `=` symbol

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

Explicitly declaring the return type is [optional](#) when this can be inferred by the compiler

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

Explicit return types

Functions with block body must always specify return types explicitly, unless it's intended for them to return `Unit`, [in which case it is optional](#). Kotlin does not infer return types for functions with block bodies because such functions may have complex control flow in the body, and the return type will be non-obvious to the reader (and sometimes even for the compiler).

Variable number of arguments (Varargs)

A parameter of a function (normally the last one) may be marked with `vararg` modifier:

```
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
}
```

allowing a variable number of arguments to be passed to the function:

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

Inside a function a `vararg`-parameter of type `T` is visible as an array of `T`, i.e. the `ts` variable in the example above has type `Array<out T>`.

Only one parameter may be marked as `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.

When we call a `vararg`-function, we can pass arguments one-by-one, e.g. `asList(1, 2, 3)`, or, if we already have an array and want to pass its contents to the function, we use the **spread** operator (prefix the array with `*`):

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

Function Scope

In Kotlin functions can be declared at top level in a file, meaning you do not need to create a class to hold a function, like languages such as Java, C# or Scala. In addition to top level functions, Kotlin functions can also be declared local, as member functions and extension functions.

Local Functions

Kotlin supports local functions, i.e. a function inside another function

```
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())
}
```

Local function can access local variables of outer functions (i.e. the closure), so in the case above, the *visited* can be a local variable

```
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

A member function is a function that is defined inside a class or object

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

Member functions are called with dot notation

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

For more information on classes and overriding members see [Classes](#) and [Inheritance](#)

Generic Functions

Functions can have generic parameters which are specified using angle brackets before the function name

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

For more information on generic functions see [Generics](#)

Inline Functions

Inline functions are explained [here](#)

Extension Functions

Extension functions are explained in [their own section](#)

Higher-Order Functions and Lambdas

Higher-Order functions and Lambdas are explained in [their own section](#)

Tail recursive functions

Kotlin supports a style of functional programming known as [tail 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.

Higher-Order Functions and Lambdas

Higher-Order Functions

A higher-order function is a function that takes functions as parameters, or returns a function. A good example of such a function is `lock()` that takes a lock object and a function, acquires the lock, runs the function and releases the lock:

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

Let's examine the code above: `body` has a [function type](#): `() -> T`, so it's supposed to be a function that takes no parameters and returns a value of type `T`. It is invoked inside the `try`-block, while protected by the `lock`, and its result is returned by the `lock()` function.

If we want to call `lock()`, we can pass another function to it as an argument (see [function references](#)):

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

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

Another, often more convenient way is to pass a [lambda expression](#):

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

Lambda expressions are described in more [detail below](#), but for purposes of continuing this section, let's see a brief overview:

- A lambda expression is always surrounded by curly braces,
- Its parameters (if any) are declared before `->` (parameter types may be omitted),
- The body goes after `->` (when present).

In Kotlin, there is a convention that if the last parameter to a function is a function, and you're passing a lambda expression as the corresponding argument, you can specify it outside of parentheses:

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

Another example of a higher-order function would be `map()`:

```
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
}
```

This function can be called as follows:

```
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: implicit name of a single parameter

One other helpful convention is that if a function literal has only one parameter, its declaration may be omitted (along with the `>`), and its name will be `it`:

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

These conventions allow to write [LINQ-style](#) code:

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

Underscore for unused variables (since 1.1)

If the lambda parameter is unused, you can place an underscore instead of its name:

```
map.forEach { _, value -> println("$value!") }
```

Destructuring in Lambdas (since 1.1)

Destructuring in lambdas is described as a part of [destructuring declarations](#).

Inline Functions

Sometimes it is beneficial to enhance performance of higher-order functions using [inline functions](#).

Lambda Expressions and Anonymous Functions

A lambda expression or an anonymous function is a "function literal", i.e. a function that is not declared, but passed immediately as an expression. Consider the following example:

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

Function `max` is a higher-order function, i.e. it takes a function value as the second argument. This second argument is an expression that is itself a function, i.e. a function literal. As a function, it is equivalent to

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

Function Types

For a function to accept another function as a parameter, we have to specify a function type for that parameter. For example the abovementioned function `max` is defined as follows:

```
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  
}
```

The parameter `less` is of type `(T, T) -> Boolean`, i.e. a function that takes two parameters of type `T` and returns a `Boolean`: true if the first one is smaller than the second one.

In the body, line 4, `less` is used as a function: it is called by passing two arguments of type `T`.

A function type is written as above, or may have named parameters, if you want to document the meaning of each parameter.

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

Lambda Expression Syntax

The full syntactic form of lambda expressions, i.e. literals of function types, is as follows:

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

A lambda expression is always surrounded by curly braces, parameter declarations in the full syntactic form go inside parentheses and have optional type annotations, the body goes after an `->` sign. If the inferred return type of the lambda is not `Unit`, the last (or possibly single) expression inside the lambda body is treated as the return value.

If we leave all the optional annotations out, what's left looks like this:

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

It's very common that a lambda expression has only one parameter. If Kotlin can figure the signature out itself, it allows us not to declare the only parameter, and will implicitly declare it for us under the name `it`:

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

We can explicitly return a value from the lambda using the [qualified return](#) syntax. Otherwise, the value of the last expression is implicitly returned. Therefore, the two following snippets are equivalent:

```
ints.filter {  
    val shouldFilter = it > 0  
    shouldFilter  
}  
  
ints.filter {  
    val shouldFilter = it > 0  
    return@filter shouldFilter  
}
```

Note that if a function takes another function as the last parameter, the lambda expression argument can be passed outside the parenthesized argument list. See the grammar for [callSuffix](#).

Anonymous Functions

One thing missing from the lambda expression syntax presented above is the ability to specify the return type of the function. In most cases, this is unnecessary because the return type can be inferred automatically. However, if you do need to specify it explicitly, you can use an alternative syntax: an *anonymous function*.

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

An anonymous function looks very much like a regular function declaration, except that its name is omitted. Its body can be either an expression (as shown above) or a block:

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

The parameters and the return type are specified in the same way as for regular functions, except that the parameter types can be omitted if they can be inferred from context:

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

The return type inference for anonymous functions works just like for normal functions: the return type is inferred automatically for anonymous functions with an expression body and has to be specified explicitly (or is assumed to be `Unit`) for anonymous functions with a block body.

Note that anonymous function parameters are always passed inside the parentheses. The shorthand syntax allowing to leave the function outside the parentheses works only for lambda expressions.

One other difference between lambda expressions and anonymous functions is the behavior of [non-local returns](#). A `return` statement without a label always returns from the function declared with the `fun` keyword. This means that a `return` inside a lambda expression will return from the enclosing function, whereas a `return` inside an anonymous function will return from the anonymous function itself.

Closures

A lambda expression or anonymous function (as well as a [local function](#) and an [object expression](#)) can access its *closure*, i.e. the variables declared in the outer scope. Unlike Java, the variables captured in the closure can be modified:

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

Function Literals with Receiver

Kotlin provides the ability to call a function literal with a specified *receiver object*. Inside the body of the function literal, you can call methods on that receiver object without any additional qualifiers. This is similar to extension functions, which allow you to access members of the receiver object inside the body of the function. One of the most important examples of their usage is [Type-safe Groovy-style builders](#).

The type of such a function literal is a function type with receiver:

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

The function literal can be called as if it were a method on the receiver object:

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

The anonymous function syntax allows you to specify the receiver type of a function literal directly. This can be useful if you need to declare a variable of a function type with receiver, and to use it later.

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

Lambda expressions can be used as function literals with receiver when the receiver type can be inferred from context.

```
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 Functions

Using [higher-order functions](#) imposes certain runtime penalties: each function is an object, and it captures a closure, i.e. those variables that are accessed in the body of the function. Memory allocations (both for function objects and classes) and virtual calls introduce runtime overhead.

But it appears that in many cases this kind of overhead can be eliminated by inlining the lambda expressions. The functions shown below are good examples of this situation. I.e., the `lock()` function could be easily inlined at call-sites. Consider the following case:

```
lock(1) { foo() }
```

Instead of creating a function object for the parameter and generating a call, the compiler could emit the following code

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

Isn't it what we wanted from the very beginning?

To make the compiler do this, we need to mark the `lock()` function with the `inline` modifier:

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

The `inline` modifier affects both the function itself and the lambdas passed to it: all of those will be inlined into the call site.

Inlining may cause the generated code to grow, but if we do it in a reasonable way (do not inline big functions) it will pay off in performance, especially at "megamorphic" call-sites inside loops.

noinline

In case you want only some of the lambdas passed to an inline function to be inlined, you can mark some of your function parameters with the `noinline` modifier:

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

Inlinable lambdas can only be called inside the inline functions or passed as inlinable arguments, but `noinline` ones can be manipulated in any way we like: stored in fields, passed around etc.

Note that if an inline function has no inlinable function parameters and no [reified type parameters](#), the compiler will issue a warning, since inlining such functions is very unlikely to be beneficial (you can suppress the warning if you are sure the inlining is needed).

Non-local returns

In Kotlin, we can only use a normal, unqualified `return` to exit a named function or an anonymous function. This means that to exit a lambda, we have to use a [label](#), and a bare `return` is forbidden inside a lambda, because a lambda can not make the enclosing function return:

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

But if the function the lambda is passed to is inlined, the return can be inlined as well, so it is allowed:

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

Such returns (located in a lambda, but exiting the enclosing function) are called *non-local* returns. We are used to this sort of constructs in loops, which inline functions often enclose:

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

Note that some inline functions may call the lambdas passed to them as parameters not directly from the function body, but from another execution context, such as a local object or a nested function. In such cases, non-local control flow is also not allowed in the lambdas. To indicate that, the lambda parameter needs to be marked with the `crossinline` modifier:

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

break and continue are not yet available in inlined lambdas, but we are planning to support them too

Reified type parameters

Sometimes we need to access a type passed to us as a parameter:

```
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
}
```

Here, we walk up a tree and use reflection to check if a node has a certain type. It's all fine, but the call site is not very pretty:

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

What we actually want is simply pass a type to this function, i.e. call it like this:

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

To enable this, inline functions support *reified type parameters*, so we can write something like this:

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

We qualified the type parameter with the `reified` modifier, now it's accessible inside the function, almost as if it were a normal class. Since the function is inlined, no reflection is needed, normal operators like `is` and `as` are working now. Also, we can call it as mentioned above: `myTree.findParentOfType<MyTreeNodeType>()`.

Though reflection may not be needed in many cases, we can still use it with a reified type parameter:

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

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

Normal functions (not marked as inline) can not have reified parameters. A type that does not have a run-time representation (e.g. a non-reified type parameter or a fictitious type like `Nothing`) can not be used as an argument for a reified type parameter.

For a low-level description, see the [spec document](#).

Inline properties (since 1.1)

The `inline` modifier can be used on accessors of properties that don't have a backing field. You can annotate individual property accessors:

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ...
    inline set(v) { ... }
```

You can also annotate an entire property, which marks both of its accessors as inline:

```
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

At the call site, inline accessors are inlined as regular inline functions.

Coroutines

⚠ Coroutines are *experimental* in Kotlin 1.1. See details [below](#)

Some APIs initiate long-running operations (such as network IO, file IO, CPU- or GPU-intensive work, etc) and require the caller to block until they complete. Coroutines provide a way to avoid blocking a thread and replace it with a cheaper and more controllable operation: *suspension* of a coroutine.

Coroutines simplify asynchronous programming by putting the complications into libraries. The logic of the program can be expressed *sequentially* in a coroutine, and the underlying library will figure out the asynchrony for us. The library can wrap relevant parts of the user code into callbacks, subscribe to relevant events, schedule execution on different threads (or even different machines!), and the code remains as simple as if it was sequentially executed.

Many asynchronous mechanisms available in other languages can be implemented as libraries using Kotlin coroutines. This includes [async/await](#) from C# and ECMAScript, [channels](#) and [select](#) from Go, and [generators/yield](#) from C# and Python. See the description [below](#) for libraries providing such constructs.

Blocking vs Suspending

Basically, coroutines are computations that can be *suspended* without *blocking a thread*. Blocking threads is often expensive, especially under high load, because only a relatively small number of threads is practical to keep around, so blocking one of them leads to some important work being delayed.

Coroutine suspension is almost free, on the other hand. No context switch or any other involvement of the OS is required. And on top of that, suspension can be controlled by a user library to a large extent: as library authors, we can decide what happens upon a suspension and optimize/log/intercept according to our needs.

Another difference is that coroutines can not be suspended at random instructions, but rather only at so called *suspension points*, which are calls to specially marked functions.

Suspending functions

A suspension happens when we call a function marked with the special modifier, `suspend` :

```
suspend fun doSomething(foo: Foo): Bar {  
    ...  
}
```

Such functions are called *suspending functions*, because calls to them may suspend a coroutine (the library can decide to proceed without suspension, if the result for the call in question is already available). Suspending functions can take parameters and return values in the same manner as regular functions, but they can only be called from coroutines and other suspending functions. In fact, to start a coroutine, there must be at least one suspending function, and it is usually anonymous (i.e. it is a suspending lambda). Let's look at an example, a simplified `async()` function (from the [kotlinx.coroutines](#) library):

```
fun <T> async(block: suspend () -> T)
```

Here, `async()` is a regular function (not a suspending function), but the `block` parameter has a function type with the `suspend` modifier: `suspend () -> T`. So, when we pass a lambda to `async()`, it is a *suspending lambda*, and we can call a suspending function from it:

```
async {  
    doSomething(foo)  
    ...  
}
```

To continue the analogy, `await()` can be a suspending function (hence also callable from within an `async {}` block) that suspends a coroutine until some computation is done and returns its result:

```

async {
    ...
    val result = computation.await()
    ...
}

```

More information on how actual `async/await` functions work in `kotlinx.coroutines` can be found [here](#).

Note that suspending functions `await()` and `doSomething()` can not be called from a regular function like `main()`:

```

fun main(args: Array<String>) {
    doSomething() // ERROR: Suspending function called from a non-coroutine context
}

```

Also note that suspending functions can be virtual, and when overriding them, the `suspend` modifier has to be specified:

```

interface Base {
    suspend fun foo()
}

class Derived: Base {
    override suspend fun foo() { ... }
}

```

@RestrictsSuspension annotation

Extension functions (and lambdas) can also be marked `suspend`, just like regular ones. This enables creation of [DSLs](#) and other APIs that users can extend. In some cases the library author needs to prevent the user from adding *new ways* of suspending a coroutine.

To achieve this, the [@RestrictsSuspension](#) annotation may be used. When a receiver class or interface `R` is annotated with it, all suspending extensions are required to delegate to either members of `R` or other extensions to it. Since extensions can't delegate to each other indefinitely (the program would not terminate), this guarantees that all suspensions happen through calling members of `R` that the author of the library can fully control.

This is relevant in the *rare* cases when every suspension is handled in a special way in the library. For example, when implementing generators through the [buildSequence\(\)](#) function described [below](#), we need to make sure that any suspending call in the coroutine ends up calling either `yield()` or `yieldAll()` and not any other function. This is why [SequenceBuilder](#) is annotated with `@RestrictsSuspension`:

```

@RestrictsSuspension
public abstract class SequenceBuilder<in T> {
    ...
}

```

See the sources [on Github](#).

The inner workings of coroutines

We are not trying here to give a complete explanation of how coroutines work under the hood, but a rough sense of what's going on is rather important.

Coroutines are completely implemented through a compilation technique (no support from the VM or OS side is required), and suspension works through code transformation. Basically, every suspending function (optimizations may apply, but we'll not go into this here) is transformed to a state machine where states correspond to suspending calls. Right before a suspension, the next state is stored in a field of a compiler-generated class along with relevant local variables, etc. Upon resumption of that coroutine, local variables are restored and the state machine proceeds from the state right after suspension.

A suspended coroutine can be stored and passed around as an object that keeps its suspended state and locals. The type of such objects is `Continuation`, and the overall code transformation described here corresponds to the classical [Continuation-passing style](#). Consequently, suspending functions take an extra parameter of type `Continuation` under the hood.

More details on how coroutines work may be found in [this design document](#). Similar descriptions of `async/await` in other languages (such as C# or ECMAScript 2016) are relevant here, although the language features they implement may not be as general as Kotlin coroutines.

Experimental status of coroutines

The design of coroutines is [experimental](#), which means that it may be changed in the upcoming releases. When compiling coroutines in Kotlin 1.1, a warning is reported by default: *The feature "coroutines" is experimental*. To remove the warning, you need to specify an [opt-in flag](#).

Due to its experimental status, the coroutine-related API in the Standard Library is put under the `kotlin.coroutines.experimental` package. When the design is finalized and the experimental status lifted, the final API will be moved to `kotlin.coroutines`, and the experimental package will be kept around (probably in a separate artifact) for backward compatibility.

IMPORTANT NOTE: We advise library authors to follow the same convention: add the "experimental" (e.g. `com.example.experimental`) suffix to your packages exposing coroutine-based APIs so that your library remains binary compatible. When the final API is released, follow these steps:

- copy all the APIs to `com.example` (without the experimental suffix),
- keep the experimental package around for backward compatibility.

This will minimize migration issues for your users.

Standard APIs

Coroutines come in three main ingredients:

- language support (i.s. suspending functions, as described above),
- low-level core API in the Kotlin Standard Library,
- high-level APIs that can be used directly in the user code.

Low-level API: `kotlin.coroutines`

Low-level API is relatively small and should never be used other than for creating higher-level libraries. It consists of two main packages:

- [kotlin.coroutines.experimental](#) with main types and primitives such as
 - [createCoroutine\(\)](#)
 - [startCoroutine\(\)](#)
 - [suspendCoroutine\(\)](#)
- [kotlin.coroutines.experimental.intrinsics](#) with even lower-level intrinsics such as [suspendCoroutineOrReturn](#)

More details about the usage of these APIs can be found [here](#).

Generators API in `kotlin.coroutines`

The only "application-level" functions in `kotlin.coroutines.experimental` are

- [buildSequence\(\)](#)
- [buildIterator\(\)](#)

These are shipped within `kotlin-stdlib` because they are related to sequences. In fact, these functions (and we can limit ourselves to `buildSequence()` alone here) implement *generators*, i.e. provide a way to cheaply build a lazy sequence:

```

val fibonacciSeq = buildSequence {
    var a = 0
    var b = 1

    yield(1)

    while (true) {
        yield(a + b)

        val tmp = a + b
        a = b
        b = tmp
    }
}

```

This generates a lazy, potentially infinite Fibonacci sequence by creating a coroutine that yields consecutive Fibonacci numbers by calling the `yield()` function. When iterating over such a sequence every step of the iterator executes another portion of the coroutine that generates the next number. So, we can take any finite list of numbers out of this sequence, e.g. `fibonacciSeq.take(8).toList()` results in `[1, 1, 2, 3, 5, 8, 13, 21]`. And coroutines are cheap enough to make this practical.

To demonstrate the real laziness of such a sequence, let's print some debug output inside a call to `buildSequence()`:

```

val lazySeq = buildSequence {
    print("START ")
    for (i in 1..5) {
        yield(i)
        print("STEP ")
    }
    print("END")
}

// Print the first three elements of the sequence
lazySeq.take(3).forEach { print("$it ") }

```

Run the code above to see that if we print the first three elements, the numbers are interleaved with the `STEP`s the generating loop. This means that the computation is lazy indeed. To print `1` we only execute until the first `yield(i)`, and print `START` along the way. Then, to print `2` we need to proceed to the next `yield(i)`, and this prints `STEP`. Same for `3`. And the next `STEP` never gets printed (as well as `END`), because we never requested further elements of the sequence.

To yield a collection (or sequence) of values at once, the `yieldAll()` function is available:

```

val lazySeq = buildSequence {
    yield(0)
    yieldAll(1..10)
}

lazySeq.forEach { print("$it ") }

```

The `buildIterator()` works similarly to `buildSequence()`, but returns a lazy iterator.

One can add custom yielding logic to `buildSequence()` by writing suspending extensions to the `SequenceBuilder` class (that bears the `@RestrictsSuspension` annotation described [above](#)):

```

suspend fun SequenceBuilder<Int>.yieldIfOdd(x: Int) {
    if (x % 2 != 0) yield(x)
}

val lazySeq = buildSequence {
    for (i in 1..10) yieldIfOdd(i)
}

```


Other high-level APIs: `kotlinx.coroutines`

Only core APIs related to coroutines are available from the Kotlin Standard Library. This mostly consists of core primitives and interfaces that all coroutine-based libraries are likely to use.

Most application-level APIs based on coroutines are released as a separate library: [kotlinx.coroutines](#). This library covers

- Platform-agnostic asynchronous programming with `kotlinx.coroutines-core`
 - this module includes Go-like channels that support `select` and other convenient primitives
 - a comprehensive guide to this library is available [here](#).
- APIs based on `CompletableFuture` from JDK 8: `kotlinx.coroutines-jdk8`
- Non-blocking IO (NIO) based on APIs from JDK 7 and higher: `kotlinx.coroutines-nio`
- Support for Swing (`kotlinx.coroutines-swing`) and JavaFx (`kotlinx.coroutines-javafx`)
- Support for Rxjava: `kotlinx.coroutines-rx`

These libraries serve as both convenient APIs that make common tasks easy and end-to-end examples of how to build coroutine-based libraries.

Other

Destructuring Declarations

Sometimes it is convenient to *destructure* an object into a number of variables, for example:

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

This syntax is called a *destructuring declaration*. A destructuring declaration creates multiple variables at once. We have declared two new variables: `name` and `age`, and can use them independently:

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

A destructuring declaration is compiled down to the following code:

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

The `component1()` and `component2()` functions are another example of the *principle of conventions* widely used in Kotlin (see operators like `+` and `*`, `for`-loops etc.). Anything can be on the right-hand side of a destructuring declaration, as long as the required number of component functions can be called on it. And, of course, there can be `component3()` and `component4()` and so on.

Note that the `componentN()` functions need to be marked with the `operator` keyword to allow using them in a destructuring declaration.

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

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

Variables `a` and `b` get the values returned by `component1()` and `component2()` called on elements of the collection.

Example: Returning Two Values from a Function

Let's say we need to return two things from a function. For example, a result object and a status of some sort. A compact way of doing this in Kotlin is to declare a [data class](#) and return its instance:

```
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(...)
```

Since data classes automatically declare `componentN()` functions, destructuring declarations work here.

NOTE: we could also use the standard class `Pair` and have `function()` return `Pair<Int, Status>`, but it's often better to have your data named properly.

Example: Destructuring Declarations and Maps

Probably the nicest way to traverse a map is this:

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

To make this work, we should

- present the map as a sequence of values by providing an `iterator()` function,
- present each of the elements as a pair by providing functions `component1()` and `component2()`.

And indeed, the standard library provides such extensions:

```
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()
```

So you can freely use destructuring declarations in `for`-loops with maps (as well as collections of data class instances etc).

Underscore for unused variables (since 1.1)

If you don't need a variable in the destructuring declaration, you can place an underscore instead of its name:

```
val (_, status) = getResult()
```

Destructuring in Lambdas (since 1.1)

You can use the destructuring declarations syntax for lambda parameters. If a lambda has a parameter of the `Pair` type (or `Map.Entry`, or any other type that has the appropriate `componentN` functions), you can introduce several new parameters instead of one by putting them in parentheses:

```
map.mapValues { entry -> "${entry.value}!" }  
map.mapValues { (key, value) -> "$value!" }
```

Note the difference between declaring two parameters and declaring a destructuring pair instead of a parameter:

```
{ a -> ... } // one parameter  
{ a, b -> ... } // two parameters  
{ (a, b) -> ... } // a destructured pair  
{ (a, b), c -> ... } // a destructured pair and another parameter
```

If a component of the destructured parameter is unused, you can replace it with the underscore to avoid inventing its name:

```
map.mapValues { (_, value) -> "$value!" }
```

You can specify the type for the whole destructured parameter or for a specific component separately:

```
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }  
map.mapValues { (_, value: String) -> "$value!" }
```

Collections

Unlike many languages, Kotlin distinguishes between mutable and immutable collections (lists, sets, maps, etc). Precise control over exactly when collections can be edited is useful for eliminating bugs, and for designing good APIs.

It is important to understand up front the difference between a read only *view* of a mutable collection, and an actually immutable collection. Both are easy to create, but the type system doesn't express the difference, so keeping track of that (if it's relevant) is up to you.

The Kotlin `List<out T>` type is an interface that provides read only operations like `size`, `get` and so on. Like in Java, it inherits from `Collection<T>` and that in turn inherits from `Iterable<T>`. Methods that change the list are added by the `MutableList<T>` interface. This pattern holds also for `Set<out T>/MutableSet<T>` and `Map<K, out V>/MutableMap<K, V>`.

We can see basic usage of the list and set types below:

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)           // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)      // prints "[1, 2, 3, 4]"
readOnlyView.clear()       // -> does not compile

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

Kotlin does not have dedicated syntax constructs for creating lists or sets. Use methods from the standard library, such as `listOf()`, `mutableListOf()`, `setOf()`, `mutableSetOf()`. Map creation in NOT performance-critical code can be accomplished with a simple [idiom](#): `mapOf(a to b, c to d)`

Note that the `readOnlyView` variable points to the same list and changes as the underlying list changes. If the only references that exist to a list are of the read only variety, we can consider the collection fully immutable. A simple way to create such a collection is like this:

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

Currently, the `listOf` method is implemented using an array list, but in future more memory-efficient fully immutable collection types could be returned that exploit the fact that they know they can't change.

Note that the read only types are [covariant](#). That means, you can take a `List<Rectangle>` and assign it to `List<Shape>` assuming `Rectangle` inherits from `Shape`. This wouldn't be allowed with the mutable collection types because it would allow for failures at runtime.

Sometimes you want to return to the caller a snapshot of a collection at a particular point in time, one that's guaranteed to not change:

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

The `toList` extension method just duplicates the lists items, thus, the returned list is guaranteed to never change.

There are various useful extension methods on lists and sets that are worth being familiar with:

```

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

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

```

... as well as all the utilities you would expect such as sort, zip, fold, reduce and so on.

Maps follow the same pattern. They can be easily instantiated and accessed like this:

```

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

```

Ranges

Range expressions are formed with `rangeTo` functions that have the operator form `..` which is complemented by `in` and `!in`. Range is defined for any comparable type, but for integral primitive types it has an optimized implementation. Here are some examples of using ranges

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

Integral type ranges (`IntRange`, `LongRange`, `CharRange`) have an extra feature: they can be iterated over. The compiler takes care of converting this analogously to Java's indexed `for`-loop, without extra overhead.

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

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

What if you want to iterate over numbers in reverse order? It's simple. You can use the `downTo()` function defined in the standard library

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

Is it possible to iterate over numbers with arbitrary step, not equal to 1? Sure, the `step()` function will help you

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

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

To create a range which does not include its end element, you can use the `until` function:

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
    println(i)
}
```

How it works

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

`ClosedRange<T>` denotes a closed interval in the mathematical sense, defined for comparable types. It has two endpoints: `start` and `endInclusive`, which are included in the range. The main operation is `contains`, usually used in the form of `in`/`!in` operators.

Integral type progressions (`IntProgression`, `LongProgression`, `CharProgression`) denote an arithmetic progression. Progressions are defined by the `first` element, the `last` element and a non-zero `step`. The first element is `first`, subsequent elements are the previous element plus `step`. 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. Iteration over `Progression` is equivalent to an indexed `for`-loop in Java/JavaScript:

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

For integral types, the `..` operator creates an object which implements both `ClosedRange<T>` and `*Progression`. For example, `IntRange` implements `ClosedRange<Int>` and extends `IntProgression`, thus all operations defined for `IntProgression` are available for `IntRange` as well. The result of the `downTo()` and `step()` functions is always a `*Progression`.

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

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

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

Utility functions

`rangeTo()`

The `rangeTo()` operators on integral types simply call the constructors of `*Range` classes, e.g.:

```
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()`

The `downTo()` extension function is defined for any pair of integral types, here are two examples:

```
fun Long.downTo(other: Int): LongProgression {  
    return LongProgression.fromClosedRange(this, other.toLong(), -1L)  
}  
  
fun Byte.downTo(other: Int): IntProgression {  
    return IntProgression.fromClosedRange(this.toInt(), other, -1)  
}
```

`reversed()`

The `reversed()` extension functions are defined for each `*Progression` classes, and all of them return reversed progressions.

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

`step()`

`step()` extension functions are defined for `*Progression` classes, all of them return progressions with modified `step` values (function parameter). The step value is required to be always positive, therefore this function never changes the direction of iteration.

```

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

```


Type Checks and Casts

is and !is Operators

We can check whether an object conforms to a given type at runtime by using the `is` operator or its negated form `!is`:

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

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

Smart Casts

In many cases, one does not need to use explicit cast operators in Kotlin, because the compiler tracks the `is`-checks for immutable values and inserts (safe) casts automatically when needed:

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

The compiler is smart enough to know a cast to be safe if a negative check leads to a return:

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

or in the right-hand side of `&&` and `||`:

```
// 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
}
```

Such *smart casts* work for [when-expressions](#) and [while-loops](#) as well:

```
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).

"Unsafe" cast operator

Usually, the cast operator throws an exception if the cast is not possible. Thus, we call it *unsafe*. The unsafe cast in Kotlin is done by the infix operator `as` (see [operator precedence](#)):

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

Note that `null` cannot be cast to `String` as this type is not [nullable](#), i.e. if `y` is null, the code above throws an exception. In order to match Java cast semantics we have to have nullable type at cast right hand side, like

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

"Safe" (nullable) cast operator

To avoid an exception being thrown, one can use a *safe* cast operator `as?` that returns `null` on failure:

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

Note that despite the fact that the right-hand side of `as?` is a non-null type `String` the result of the cast is nullable.

This Expression

To denote the current *receiver*, we use `this` expressions:

- In a member of a [class](#), `this` refers to the current object of that class
- In an [extension function](#) or a [function literal with receiver](#), `this` denotes the *receiver* parameter that is passed on the left-hand side of a dot.

If `this` has no qualifiers, it refers to the *innermost enclosing scope*. To refer to `this` in other scopes, *label qualifiers* are used:

Qualified `this`

To access `this` from an outer scope (a [class](#), or [extension function](#), or labeled [function literal with receiver](#)) we write `this@label` where `@label` is a [label](#) 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
      }
    }
  }
}
```

Equality

In Kotlin there are two types of equality:

- Referential equality (two references point to the same object)
- Structural equality (a check for `equals()`)

Referential equality

Referential equality is checked by the `===` operation (and its negated counterpart `!==`). `a === b` evaluates to true if and only if `a` and `b` point to the same object.

Structural equality

Structural equality is checked by the `==` operation (and its negated counterpart `!=`). By convention, an expression like `a == b` is translated to

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

i.e. if `a` is not `null`, it calls the `equals(Any?)` function, otherwise (i.e. `a` is `null`) it checks that `b` is referentially equal to `null`.

Note that there's no point in optimizing your code when comparing to `null` explicitly: `a == null` will be automatically translated to `a === null`.

Operator overloading

Kotlin allows us to provide implementations for a predefined set of operators on our types. These operators have fixed symbolic representation (like `+` or `*`) and fixed [precedence](#). To implement an operator, we provide a [member function](#) or an [extension function](#) with a fixed name, for the corresponding type, i.e. left-hand side type for binary operations and argument type for unary ones. Functions that overload operators need to be marked with the `operator` modifier.

Conventions

Here we describe the conventions that regulate operator overloading for different operators.

Unary operations

Expression	Translated to
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>

This table says that when the compiler processes, for example, an expression `+a`, it performs the following steps:

- Determines the type of `a`, let it be `T`.
- Looks up a function `unaryPlus()` with the `operator` modifier and no parameters for the receiver `T`, i.e. a member function or an extension function.
- If the function is absent or ambiguous, it is a compilation error.
- If the function is present and its return type is `R`, the expression `+a` has type `R`.

Note that these operations, as well as all the others, are optimized for [Basic types](#) and do not introduce overhead of function calls for them.

Expression	Translated to
<code>a++</code>	<code>a.inc()</code> + see below
<code>a--</code>	<code>a.dec()</code> + see below

The `inc()` and `dec()` functions must return a value, which will be assigned to the variable on which the `++` or `--` operation was used. They shouldn't mutate the object on which the `inc` or `dec` was invoked.

The compiler performs the following steps for resolution of an operator in the *postfix* form, e.g. `a++`:

- Determines the type of `a`, let it be `T`.
- Looks up a function `inc()` with the `operator` modifier and no parameters, applicable to the receiver of type `T`.
- Checks that the return type of the function is a subtype of `T`.

The effect of computing the expression is:

- Store the initial value of `a` to a temporary storage `a0`,
- Assign the result of `a.inc()` to `a`,
- Return `a0` as a result of the expression.

For `a--` the steps are completely analogous.

For the *prefix* forms `++a` and `--a` resolution works the same way, and the effect is:

- Assign the result of `a.inc()` to `a`,
- Return the new value of `a` as a result of the expression.

Binary operations

Expression	Translated to
<code>a + b</code>	<code>a.plus(b)</code>

Expression	Translated to
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b), a.mod(b)</code> (deprecated)
<code>a..b</code>	<code>a.rangeTo(b)</code>

For the operations in this table, the compiler just resolves the expression in the *Translated to* column.

Note that the `rem` operator is supported since Kotlin 1.1. Kotlin 1.0 uses the `mod` operator, which is deprecated in Kotlin 1.1.

Expression	Translated to
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

For `in` and `!in` the procedure is the same, but the order of arguments is reversed.

Expression	Translated to
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

Square brackets are translated to calls to `get` and `set` with appropriate numbers of arguments.

Expression	Translated to
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

Parentheses are translated to calls to `invoke` with appropriate number of arguments.

Expression	Translated to
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.modAssign(b)</code>

For the assignment operations, e.g. `a += b`, the compiler performs the following steps:

- If the function from the right column is available
 - If the corresponding binary function (i.e. `plus()` for `plusAssign()`) is available too, report error (ambiguity).
 - Make sure its return type is `Unit`, and report an error otherwise.
 - Generate code for `a.plusAssign(b)`
- Otherwise, try to generate code for `a = a + b` (this includes a type check: the type of `a + b` must be a subtype of `a`).

Note: assignments are *NOT* expressions in Kotlin.

Expression	Translated to
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

Note: `===` and `!==` (identity checks) are not overloadable, so no conventions exist for them

The `==` operation is special: it is translated to a complex expression that screens for `null`'s. `null == null` is always true, and `x == null` for a non-null `x` is always false and won't invoke `x.equals()`.

Expression	Translated to
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

All comparisons are translated into calls to `compareTo`, that is required to return `Int`.

Infix calls for named functions

We can simulate custom infix operations by using [infix function calls](#).

Null Safety

Nullable types and Non-Null Types

Kotlin's type system is aimed at eliminating the danger of null references from code, also known as the [The Billion Dollar Mistake](#).

One of the most common pitfalls in many programming languages, including Java is that of accessing a member of a null reference, resulting in a null reference exception. In Java this would be the equivalent of a `NullPointerException` or NPE for short.

Kotlin's type system is aimed to eliminate `NullPointerException`'s from our code. The only possible causes of NPE's may be

- An explicit call to `throw NullPointerException()`
- Usage of the `!!` operator that is described below
- External Java code has caused it
- There's some data inconsistency with regard to initialization (an uninitialized `this` available in a constructor is used somewhere)

In Kotlin, the type system distinguishes between references that can hold `null` (nullable references) and those that can not (non-null references). For example, a regular variable of type `String` can not hold `null`:

```
var a: String = "abc"
a = null // compilation error
```

To allow nulls, we can declare a variable as nullable string, written `String?`:

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

Now, if you call a method or access a property on `a`, it's guaranteed not to cause an NPE, so you can safely say

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

But if you want to access the same property on `b`, that would not be safe, and the compiler reports an error:

```
val l = b.length // error: variable 'b' can be null
```

But we still need to access that property, right? There are a few ways of doing that.

Checking for `null` in conditions

First, you can explicitly check if `b` is `null`, and handle the two options separately:

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

The compiler tracks the information about the check you performed, and allows the call to `length` inside the `if`. More complex conditions are supported as well:

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

Note that this only works where `b` is immutable (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), because otherwise it might happen that `b` changes to `null` after the check.

Safe Calls

Your second option is the safe call operator, written `?.`:

```
b?.length
```

This returns `b.length` if `b` is not null, and `null` otherwise. The type of this expression is `Int?`.

Safe calls are useful in chains. For example, if Bob, an Employee, may be assigned to a Department (or not), that in turn may have another Employee as a department head, then to obtain the name of Bob's department head (if any), we write the following:

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

Such a chain returns `null` if any of the properties in it is null.

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

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

Elvis Operator

When we have a nullable reference `r`, we can say "if `r` is not null, use it, otherwise use some non-null value `x`":

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

Along with the complete `if`-expression, this can be expressed with the Elvis operator, written `?:`:

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

If the expression to the left of `?:` is not null, the elvis operator returns it, otherwise it returns the expression to the right. Note that the right-hand side expression is evaluated only if the left-hand side is null.

Note that, since `throw` and `return` are expressions in Kotlin, they can also be used on the right hand side of the elvis operator. This can be very handy, for example, for checking function arguments:

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

The !! Operator

The third option is for NPE-lovers. We can write `b!!`, and this will return a non-null value of `b` (e.g., a `String` in our example) or throw an NPE if `b` is null:

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

Thus, if you want an NPE, you can have it, but you have to ask for it explicitly, and it does not appear out of the blue.

Safe Casts

Regular casts may result into a `ClassCastException` if the object is not of the target type. Another option is to use safe casts that return `null` if the attempt was not successful:

```
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`.

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

Exceptions

Exception Classes

All exception classes in Kotlin are descendants of the class `Throwable`. Every exception has a message, stack trace and an optional cause.

To throw an exception object, use the `throw`-expression

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

To catch an exception, use the `try`-expression

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

There may be zero or more `catch` blocks. `finally` blocks may be omitted. However at least one `catch` or `finally` block should be present.

Try is an expression

`try` is an expression, i.e. it may have a return value.

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

The returned value of a `try`-expression is either the last expression in the `try` block or the last expression in the `catch` block (or blocks). Contents of the `finally` block do not affect the result of the expression.

Checked Exceptions

Kotlin does not have checked exceptions. There are many reasons for this, but we will provide a simple example.

The following is an example interface of the JDK implemented by `StringBuilder` class

```
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  
}
```

And this is no good, see [Effective Java](#), Item 65: *Don't ignore exceptions*.

Bruce Eckel says in [Does Java need Checked Exceptions?](#):

Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

Other citations of this sort:

- [Java's checked exceptions were a mistake](#) (Rod Waldhoff)
- [The Trouble with Checked Exceptions](#) (Anders Hejlsberg)

The Nothing type

`throw` is an expression in Kotlin, so you can use it, for example, as part of an Elvis expression:

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

The type of the `throw` expression is the special type `Nothing`. The type has no values and is used to mark code locations that can never be reached. In your own code, you can use `Nothing` to mark a function that never returns:

```
fun fail(message: String): Nothing {  
    throw IllegalArgumentException(message)  
}
```

When you call this function, the compiler will know that the execution doesn't continue beyond the call:

```
val s = person.name ?: fail("Name required")  
println(s)      // 's' is known to be initialized at this point
```

Java Interoperability

Please see the section on exceptions in the [Java Interoperability section](#) for information about Java interoperability.

Annotations

Annotation Declaration

Annotations are means of attaching metadata to code. To declare an annotation, put the `annotation` modifier in front of a class:

```
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
annotation class Fancy
```

Usage

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

If you need to annotate the primary constructor of a class, you need to add the `constructor` keyword to the constructor declaration, and add the annotations before it:

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

You can also annotate property accessors:

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

Constructors

Annotations may have constructors that take parameters.

```
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.`

Annotation parameters cannot have nullable types, because the JVM does not support storing `null` as a value of an annotation attribute.

If an annotation is used as a parameter of another annotation, its name is not prefixed with the `@` character:

```
annotation class ReplaceWith(val expression: String)

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

Annotations can also be used on lambdas. They will be applied to the `invoke()` method into which the body of the lambda is generated. This is useful for frameworks like [Quasar](#), which uses annotations for concurrency control.

```
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]
    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`

Java Annotations

Java annotations are 100% compatible with 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())
    }
}
```

Since the order of parameters for an annotation written in Java is not defined, you can't use a regular function call syntax for passing the arguments. Instead, you need to use the named argument syntax.

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

Just like in Java, a special case is the `value` parameter; its value can be specified without an explicit name.

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

If the `value` argument in Java has an array type, it becomes a `vararg` parameter in Kotlin:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

For other arguments that have an array type, you need to use `arrayOf` explicitly:

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
// Kotlin
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar")) class C
```

Values of an annotation instance are exposed as properties to Kotlin code.

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```


Reflection

Reflection is a set of language and library features that allows for introspecting the structure of your own program at runtime. Kotlin makes functions and properties first-class citizens in the language, and introspecting them (i.e. learning a name or a type of a property or function at runtime) is closely intertwined with simply using a functional or reactive style.

⚠ On the Java platform, the runtime component required for using the reflection features is distributed as a separate JAR file (`kotlin-reflect.jar`). This is done to reduce the required size of the runtime library for applications that do not use reflection features. If you do use reflection, please make sure that the `.jar` file is added to the classpath of your project.

Class References

The most basic reflection feature is getting the runtime reference to a Kotlin class. To obtain the reference to a statically known Kotlin class, you can use the *class literal* syntax:

```
val c = MyClass::class
```

The reference is a value of type `KClass`.

Note that a Kotlin class reference is not the same as a Java class reference. To obtain a Java class reference, use the `.java` property on a `KClass` instance.

Bound Class References (since 1.1)

You can get the reference to a class of a specific object with the same `::class` syntax by using the object as a receiver:

```
val widget: Widget = ...
assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}" }
```

You obtain the reference to an exact class of an object, for instance `GoodWidget` or `BadWidget`, despite the type of the receiver expression (`Widget`).

Function References

When we have a named function declared like this:

```
fun isOdd(x: Int) = x % 2 != 0
```

We can easily call it directly (`isOdd(5)`), but we can also pass it as a value, e.g. to another function. To do this, we use the `::` operator:

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // prints [1, 3]
```

Here `::isOdd` is a value of function type `(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)
```

If we need to use a member of a class, or an extension function, it needs to be qualified. e.g. `String::toCharArray` gives us an extension function for type `String: String.() -> CharArray`.

Example: Function Composition

Consider the following function:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

It returns a composition of two functions passed to it: `compose(f, g) = f(g(*))`. Now, you can apply it to callable references:

```
fun length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // Prints "[a, abc]"
```

Property References

To access properties as first-class objects in Kotlin, we can also use the `::` operator:

```
var x = 1

fun main(args: Array<String>) {
    println(::x.get()) // prints "1"
    ::x.set(2)
    println(x)         // prints "2"
}
```

The expression `::x` evaluates to a property object of type `KProperty<Int>`, which allows us to read its value using `get()` or retrieve the property name using the `name` property. For more information, please refer to the [docs on the KProperty class](#).

For a mutable property, e.g. `var y = 1`, `::y` returns a value of type `KMutableProperty<Int>`, which has a `set()` method.

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"
}
```

For an extension property:

```

val String.lastChar: Char
    get() = this[length - 1]

fun main(args: Array<String>) {
    println(String::lastChar.get("abc")) // prints "c"
}

```

Interoperability With Java Reflection

On the Java platform, standard library contains extensions for reflection classes that provide a mapping to and from Java reflection objects (see package `kotlin.reflect.jvm`). For example, to find a backing field or a Java method that serves as a getter for a Kotlin property, you can say something like this:

```

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

```

Constructor References

Constructors can be referenced just like methods and properties. They can be used wherever an object of function type is expected that takes the same parameters as the constructor and returns an object of the appropriate type. Constructors are referenced by using the `::` operator and adding the class name. Consider the following function that expects a function parameter with no parameters and return type `Foo`:

```

class Foo

fun function(factory : () -> Foo) {
    val x : Foo = factory()
}

```

Using `::Foo`, the zero-argument constructor of the class `Foo`, we can simply call it like this:

```

function(::Foo)

```

Bound Function and Property References (since 1.1)

You can refer to an instance method of a particular object.

```

val numberRegex = "\\d+".toRegex()
println(numberRegex.matches("29")) // prints "true"

val isNumber = numberRegex::matches
println(isNumber("29")) // prints "true"

```

Instead of calling the method `matches` directly we are storing a reference to it. Such reference is bound to its receiver. It can be called directly (like in the example above) or used whenever an expression of function type is expected:

```

val strings = listOf("abc", "124", "a70")
println(strings.filter(numberRegex::matches)) // prints "[124]"

```

Compare the types of bound and the corresponding unbound references. Bound callable reference has its receiver "attached" to it, so the type of the receiver is no longer a parameter:

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches  
  
val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

Property reference can be bound as well:

```
val prop = "abc"::length  
println(prop.get()) // prints "3"
```

Type-Safe Builders

The concept of [builders](#) is rather popular in the *Groovy* community. Builders allow for defining data in a semi-declarative way. Builders are good for [generating XML](#), [laying out UI components](#), [describing 3D scenes](#) and more...

For many use cases, Kotlin allows to *type-check* builders, which makes them even more attractive than the dynamically-typed implementation made in Groovy itself.

For the rest of the cases, Kotlin supports Dynamic types builders.

A type-safe builder example

Consider the following code:

```
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
            }
        }
    }
```

This is completely legitimate Kotlin code. You can play with this code online (modify it and run in the browser) [here](#).

How it works

Let's walk through the mechanisms of implementing type-safe builders in Kotlin. First of all we need to define the model we want to build, in this case we need to model HTML tags. It is easily done with a bunch of classes. For example, `HTML` is a class that describes the `<html>` tag, i.e. it defines children like `<head>` and `<body>`. (See its declaration [below](#).)

Now, let's recall why we can say something like this in the code:

```
html {
    // ...
}
```

`html` is actually a function call that takes a [lambda expression](#) as an argument. This function is defined as follows:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

This function takes one parameter named `init`, which is itself a function. 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` and `body` are member functions of `HTML`.)

Now, `this` can be omitted, as usual, and we get something that looks very much like a builder already:

```
html {
    head { /* ... */ }
    body { /* ... */ }
}
```

So, what does this call do? Let's look at the body of `html` function as defined above. It creates a new instance of `HTML`, then it initializes it by calling the function that is passed as an argument (in our example this boils down to calling `head` and `body` on the `HTML` instance), and then it returns this instance. This is exactly what a builder should do.

The `head` and `body` functions in the `HTML` class are defined similarly to `html`. The only difference is that they add the built instances to the `children` collection of the enclosing `HTML` instance:

```
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
}
```

Actually these two functions do just the same thing, so we can have a generic version, `initTag`:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

So, now our functions are very simple:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

And we can use them to build `<head>` and `<body>` tags.

One other thing to be discussed here is how we add text to tag bodies. In the example above we say something like

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

So basically, we just put a string inside a tag body, but there is this little `+` in front of it, so it is a function call that invokes a prefix unaryPlus() operation. That operation is actually defined by an extension function unaryPlus() that is a member of the TagWithText abstract class (a parent of Title):

```
fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

So, what the prefix `+` does here is it wraps a string into an instance of TextElement and adds it to the children collection, so that it becomes a proper part of the tag tree.

All this is defined in a package `com.example.html` that is imported at the top of the builder example above. In the last section you can read through the full definition of this package.

Scope control: @DslMarker (since 1.1)

When using DSLs, one might have come across the problem that too many functions can be called in the context. We can call methods of every available implicit receiver inside a lambda and therefore get an inconsistent result, like the tag `head` inside another `head`:

```
html {
    head {
        head {} // should be forbidden
    }
    // ...
}
```

In this example only members of the nearest implicit receiver `this@head` must be available; `head()` is a member of the outer receiver `this@html`, so it must be illegal to call it.

To address this problem, in Kotlin 1.1 a special mechanism to control receiver scope was introduced.

To make the compiler start controlling scopes we only have to annotate the types of all receivers used in the DSL with the same marker annotation. For instance, for HTML Builders we declare an annotation `@HTMLTagMarker`:

```
@DslMarker
annotation class HTMLTagMarker
```

An annotation class is called a DSL marker if it is annotated with the `@DslMarker` annotation.

In our DSL all the tag classes extend the same superclass `Tag`. It's enough to annotate only the superclass with `@HTMLTagMarker` and after that the Kotlin compiler will treat all the inherited classes as annotated:

```
@HTMLTagMarker
abstract class Tag(val name: String) { ... }
```

We don't have to annotate the `HTML` or `Head` classes with `@HTMLTagMarker` because their superclass is already annotated:

```
class HTML() : Tag("html") { ... }
class Head() : Tag("head") { ... }
```

After we've added this annotation, the Kotlin compiler knows which implicit receivers are part of the same DSL and allows to call members of the nearest receivers only:

```
html {
    head {
        head { } // error: a member of outer receiver
    }
    // ...
}
```

Note that it's still possible to call the members of the outer receiver, but to do that you have to specify this receiver explicitly:

```
html {
    head {
        this@html.head { } // possible
    }
    // ...
}
```

Full definition of the `com.example.html` package

This is how the package `com.example.html` is defined (only the elements used in the example above). It builds an HTML tree. It makes heavy use of [extension functions](#) and [lambdas with receiver](#).

Note that the `@DslMarker` annotation is available only since Kotlin 1.1.

```
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")
    }
}

@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
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
    }

```

Type aliases

Type aliases provide alternative names for existing types. If the type name is too long you can introduce a different shorter name and use the new one instead.

It's useful to shorten long generic types. For instance, it's often tempting to shrink collection types:

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

You can provide different aliases for function types:

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

You can have new names for inner and nested classes:

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

Type aliases do not introduce new types. They are equivalent to the corresponding underlying types. When you add `typealias Predicate<T>` and use `Predicate<Int>` in your code, the Kotlin compiler always expand it to `(Int) -> Boolean`. Thus you can pass a variable of your type whenever a general function type is required and vice versa:

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main(args: Array<String>) {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // prints "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // prints "[1]"
}
```

Reference

Grammar

Notation

This section informally explains the grammar notation used below.

Symbols and naming

Terminal symbol names start with an uppercase letter, e.g. **SimpleName**.

Nonterminal symbol names start with lowercase letter, e.g. **kotlinFile**.

Each *production* starts with a colon (:).

Symbol definitions may have many productions and are terminated by a semicolon (;).

Symbol definitions may be prepended with *attributes*, e.g. `start` attribute denotes a start symbol.

EBNF expressions

Operator `|` denotes *alternative*.

Operator `*` denotes *iteration* (zero or more).

Operator `+` denotes *iteration* (one or more).

Operator `?` denotes *option* (zero or one).

alpha { beta } denotes a nonempty *beta*-separated list of *alpha*'s.

Operator ```++``` means that no space or comment allowed between operands.

Semicolons

Kotlin provides "semicolon inference": syntactically, subsentences (e.g., statements, declarations etc) are separated by the pseudo-token [SEMI](#), which stands for "semicolon or newline". In most cases, there's no need for semicolons in Kotlin code.

Syntax

Relevant pages: [Packages](#)

```
start
kotlinFile
: preamble topLevelObject*
;
start
script
: preamble expression*
;
preamble
(used by script, kotlinFile)
: fileAnnotations? packageHeader? import*
;
fileAnnotations
(used by preamble)
: fileAnnotation*
;
fileAnnotation
(used by fileAnnotations)
: "@" "file" ":" ("[" unescapedAnnotation + "]" | unescapedAnnotation)
;
packageHeader
(used by preamble)
: modifiers "package" SimpleName { "." } SEMI?
;
See Packages

import
(used by preamble)
: "import" SimpleName { "." } ( "." ".*" | "as" SimpleName )? SEMI?
```

```

;
See Imports

topLevelObject
(used by kotlinFile)
: class
: object
: function
: property
: typeAlias
;

typeAlias
(used by memberDeclaration, declaration, topLevelObject)
: modifiers "typealias" SimpleName typeParameters? "=" type
;

```

Classes

See [Classes and Inheritance](#)

```

class
(used by memberDeclaration, declaration, topLevelObject)
: modifiers ("class" | "interface") SimpleName
  typeParameters?
  primaryConstructor?
  (":" annotations delegationSpecifier{"", "}")?
  typeConstraints
  (classBody? | enumClassBody)
;

primaryConstructor
(used by class, object)
: (modifiers "constructor")? ("(" functionParameter{"", "}" ")")
;

classBody
(used by objectLiteral, enumEntry, class, companionObject, object)
: ("{" members "}")?
;

members
(used by enumClassBody, classBody)
: memberDeclaration*
;

delegationSpecifier
(used by objectLiteral, class, companionObject, object)
: constructorInvocation
: userType
: explicitDelegation
;

explicitDelegation
(used by delegationSpecifier)
: userType "by" expression
;

typeParameters
(used by typeAlias, class, property, function)
: "<" typeParameter{"", "}" ">"
;

typeParameter
(used by typeParameters)
: modifiers SimpleName (":" userType)?
;

See Generic classes

```

```

typeConstraints
(used by class, property, function)
: ("where" typeConstraint{"", "}")?
;

typeConstraint
(used by typeConstraints)
: annotations SimpleName ":" type
;

See Generic constraints

```

Class members

```

memberDeclaration
(used by members)
: companionObject
: object
: function
: property
: class
: typeAlias
: anonymousInitializer
: secondaryConstructor
;

anonymousInitializer
(used by memberDeclaration)

```

```

: "init" block
;
companionObject
(used by memberDeclaration)
: modifiers "companion" "object" SimpleName? (":" delegationSpecifier{"", "}")? classBody?
;
valueParameters
(used by secondaryConstructor, function)
: ("functionParameter{"", "}"? " ")
;
functionParameter
(used by valueParameters, primaryConstructor)
: modifiers ("val" | "var")? parameter ("=" expression)?
;
block
(used by catchBlock, anonymousInitializer, secondaryConstructor, functionBody, controlStructureBody, try, finallyBlock)
: ("{" statements "}")
;
function
(used by memberDeclaration, declaration, topLevelObject)
: modifiers "fun"
  typeParameters?
  (type "."?
  SimpleName
  typeParameters? valueParameters (":" type)?
  typeConstraints
  functionBody?
  )
;
functionBody
(used by getter, setter, function)
: block
  "==" expression
;
variableDeclarationEntry
(used by for, lambdaParameter, property, multipleVariableDeclarations)
: SimpleName (":" type)?
;
multipleVariableDeclarations
(used by for, lambdaParameter, property)
: ("(" variableDeclarationEntry{"", "}" " ")
;
property
(used by memberDeclaration, declaration, topLevelObject)
: modifiers ("val" | "var")
  typeParameters?
  (type "."?
  (multipleVariableDeclarations | variableDeclarationEntry)
  typeConstraints
  ("by" | "==" expression SEMI)?
  (getter? setter? | setter? getter?) SEMI?
  )
;

```

See [Properties and Fields](#)

```

getter
(used by property)
: modifiers "get"
  modifiers "get" "(" " )" (":" type)? functionBody
;
setter
(used by property)
: modifiers "set"
  modifiers "set" "(" modifiers (SimpleName | parameter) ")" functionBody
;
parameter
(used by functionType, setter, functionParameter)
: SimpleName ":" type
;
object
(used by memberDeclaration, declaration, topLevelObject)
: "object" SimpleName primaryConstructor? (":" delegationSpecifier{"", "}")? classBody?
secondaryConstructor
(used by memberDeclaration)
: modifiers "constructor" valueParameters (":" constructorDelegationCall)? block
;
constructorDelegationCall
(used by secondaryConstructor)
: "this" valueArguments
  "super" valueArguments
;

```

See [Object expressions and Declarations](#)

Enum classes

See [Enum classes](#)

```

enumClassBody
(used by class)
: ("{" enumEntries ("," members)? "}")

```

```

;
enumEntries
(used by enumClassBody)
: (enumEntry{",","} ","? ",";","?")?
;
enumEntry
(used by enumEntries)
: modifiers SimpleName ("(" arguments ")")? classBody?
;

```

Types

See [Types](#)

```

type
(used by namedInfix, simpleUserType, getter, atomicExpression, whenCondition, property, typeArguments, function, typeAlias,
parameter, functionType, variableDeclarationEntry, lambdaParameter, typeConstraint)
: typeModifiers typeReference
;
typeReference
(used by typeReference, nullableType, type)
: ("(" typeReference ")")
: functionType
: userType
: nullableType
: "dynamic"
;
nullableType
(used by typeReference)
: typeReference "?"
;
userType
(used by typeParameter, catchBlock, callableReference, typeReference, delegationSpecifier, constructorInvocation,
explicitDelegation)
: simpleUserType{"."}
;
simpleUserType
(used by userType)
: SimpleName ("<" (optionalProjection type | "*" ){","} ">")?
;
optionalProjection
(used by simpleUserType)
: varianceAnnotation
;
functionType
(used by typeReference)
: (type "." )? ("(" parameter{","} ? ")" "->" type ?
;

```

Control structures

See [Control structures](#)

```

controlStructureBody
(used by whenEntry, for, if, doWhile, while)
: block
: blockLevelExpression
;
if
(used by atomicExpression)
: "if" "(" (expression ")") controlStructureBody SEMI? ("else" controlStructureBody)?
;
try
(used by atomicExpression)
: "try" block catchBlock* finallyBlock?
;
catchBlock
(used by try)
: "catch" "(" (annotations SimpleName ":" userType ")" block
;
finallyBlock
(used by try)
: "finally" block
;
loop
(used by atomicExpression)
: for
: while
: doWhile
;
for
(used by loop)
: "for" "(" (annotations (multipleVariableDeclarations | variableDeclarationEntry) "in" expression ")" controlStructureBody
;
while
(used by loop)
: "while" "(" (expression ")" controlStructureBody

```

```

;
doWhile
(used by loop)
: "do" controlStructureBody "while" "(" expression ")"
;

```

Expressions

Precedence

Precedence	Title	Symbols
Highest	Postfix	++, --, ., ?., ?
	Prefix	-, +, ++, --, !, labelDefinition @
	Type RHS	:, as, as?
	Multiplicative	*, /, %
	Additive	+, -
	Range	..
	Infix function	SimpleName
	Elvis	?:
	Named checks	in, !in, is, !is
	Comparison	<, >, <=, >=
	Equality	==, \!==
	Conjunction	&&
	Disjunction	
Lowest	Assignment	=, +=, -=, *=, /=, %=

Rules

```

expression
(used by for, atomicExpression, longTemplate, whenCondition, functionBody, doWhile, property, script, explicitDelegation, jump,
while, arrayAccess, blockLevelExpression, if, when, valueArguments, functionParameter)
: disjunction (assignmentOperator disjunction)*
;
disjunction
(used by expression)
: conjunction ("||" conjunction)*
;
conjunction
(used by disjunction)
: equalityComparison ("&&" equalityComparison)*
;
equalityComparison
(used by conjunction)
: comparison (equalityOperation comparison)*
;
comparison
(used by equalityComparison)
: namedInfix (comparisonOperation namedInfix)*
;
namedInfix
(used by comparison)
: elvisExpression (inOperation elvisExpression)*
: elvisExpression (isOperation type)?
;
elvisExpression
(used by namedInfix)
: infixFunctionCall ("?:" infixFunctionCall)*
;
infixFunctionCall
(used by elvisExpression)
: rangeExpression (SimpleName rangeExpression)*
;
rangeExpression
(used by infixFunctionCall)
: additiveExpression (".." additiveExpression)*
;
additiveExpression
(used by rangeExpression)
: multiplicativeExpression (additiveOperation multiplicativeExpression)*
;
multiplicativeExpression
(used by additiveExpression)
: typeRHS (multiplicativeOperation typeRHS)*
;

```

```

typeRHS
(used by multiplicativeExpression)
: prefixUnaryExpression (typeOperation prefixUnaryExpression)*
;

prefixUnaryExpression
(used by typeRHS)
: prefixUnaryOperation* postfixUnaryExpression
;

postfixUnaryExpression
(used by prefixUnaryExpression, postfixUnaryOperation)
: atomicExpression postfixUnaryOperation*
: callableReference postfixUnaryOperation*
;

callableReference
(used by postfixUnaryExpression)
: (userType "?"*)? "::" SimpleName typeArguments?
;

atomicExpression
(used by postfixUnaryExpression)
: "(" expression ")"
: literalConstant
: functionLiteral
: "this" labelReference?
: "super" ("<" type ">")? labelReference?
: if
: when
: try
: objectLiteral
: jump
: loop
: SimpleName
;

labelReference
(used by atomicExpression, jump)
: "@" ++ LabelName
;

labelDefinition
(used by prefixUnaryOperation, annotatedLambda)
: LabelName ++ "@"
;

literalConstant
(used by atomicExpression)
: "true" | "false"
: stringTemplate
: NoEscapeString
: IntegerLiteral
: HexadecimalLiteral
: CharacterLiteral
: FloatLiteral
: "null"
;

stringTemplate
(used by literalConstant)
: "\"" stringTemplateElement* "\""
;

stringTemplateElement
(used by stringTemplate)
: RegularStringPart
: ShortTemplateEntryStart (SimpleName | "this")
: EscapeSequence
: longTemplate
;

longTemplate
(used by stringTemplateElement)
: "${" expression "}"
;

declaration
(used by statement)
: function
: property
: class
: typeAlias
: object
;

statement
(used by statements)
: declaration
: blockLevelExpression
;

blockLevelExpression
(used by statement, controlStructureBody)
: annotations ("n")+ expression
;

multiplicativeOperation
(used by multiplicativeExpression)
: "*" : "/" : "%"
;

additiveOperation
(used by additiveExpression)
: "+" : "-"

```



```

;
inOperation
(used by namedInfix)
: "in" : "!in"
;
typeOperation
(used by typeRHS)
: "as" : "as?" : "."
;
isOperation
(used by namedInfix)
: "is" : "!is"
;
comparisonOperation
(used by comparison)
: "<" : ">" : ">=" : "<="
;
equalityOperation
(used by equalityComparison)
: "!=" : "=="
;
assignmentOperator
(used by expression)
: "="
: "+=" : "-=" : "*=" : "/=" : "%="
;
prefixUnaryOperation
(used by prefixUnaryExpression)
: "-" : "+"
: "++" : "--"
: "!"
: annotations
: labelDefinition
;
postfixUnaryOperation
(used by postfixUnaryExpression)
: "++" : "--" : "!!"
: callSuffix
: arrayAccess
: memberAccessOperation postfixUnaryExpression
;
callSuffix
(used by constructorInvocation, postfixUnaryOperation)
: typeArguments? valueArguments annotatedLambda
: typeArguments annotatedLambda
;
annotatedLambda
(used by callSuffix)
: ("@" unescapedAnnotation)* labelDefinition? functionLiteral
;
memberAccessOperation
(used by postfixUnaryOperation)
: "." : "?" : "?"
;
typeArguments
(used by callSuffix, callableReference, unescapedAnnotation)
: "<" type {"", ""} ">"
;
valueArguments
(used by callSuffix, constructorDelegationCall, unescapedAnnotation)
: "(" (SimpleName "=")? "*" expression {"", ""} ")"
;
jump
(used by atomicExpression)
: "throw" expression
: "return" ++ labelReference? expression?
: "continue" ++ labelReference?
: "break" ++ labelReference?
;
functionLiteral
(used by atomicExpression, annotatedLambda)
: "{" statements "}"
: "{" lambdaParameter {"", ""} "->" statements "}"
;
lambdaParameter
(used by functionLiteral)
: variableDeclarationEntry
: multipleVariableDeclarations (":" type)?
;
statements
(used by block, functionLiteral)
: SEMI* statement{SEMI+} SEMI*
;
constructorInvocation
(used by delegationSpecifier)
: userType callSuffix
;
arrayAccess
(used by postfixUnaryOperation)
: "[" expression {"", ""} "]"
;

```

```
objectLiteral
(used by atomicExpression)
: "object" ("." delegationSpecifier{"", "}")? classBody
;
```

When-expression

See [When-expression](#)

```
when
(used by atomicExpression)
: "when" ("(" expression ")")? "{"
  whenEntry*
  "}"
;
whenEntry
(used by when)
: whenCondition {"", "}" "->" controlStructureBody SEMI
: "else" "->" controlStructureBody SEMI
;
whenCondition
(used by whenEntry)
: expression
: ("in" | "in") expression
: ("is" | "is") type
;
```

Modifiers

```
modifiers
(used by typeParameter, getter, packageHeader, class, property, function, typeAlias, secondaryConstructor, enumEntry, setter,
companionObject, primaryConstructor, functionParameter)
: (modifier | annotations)*
;
typeModifiers
(used by type)
: (suspendModifier | annotations)*
;
modifier
(used by modifiers)
: classModifier
: accessModifier
: varianceAnnotation
: memberModifier
: parameterModifier
: typeParameterModifier
: functionModifier
: propertyModifier
;
classModifier
(used by modifier)
: "abstract"
: "final"
: "enum"
: "open"
: "annotation"
: "sealed"
: "data"
;
memberModifier
(used by modifier)
: "override"
: "open"
: "final"
: "abstract"
: "lateinit"
;
accessModifier
(used by modifier)
: "private"
: "protected"
: "public"
: "internal"
;
varianceAnnotation
(used by modifier, optionalProjection)
: "in"
: "out"
;
parameterModifier
(used by modifier)
: "noinline"
: "crossinline"
: "vararg"
;
typeParameterModifier
(used by modifier)
: "reified"
```

```

;
functionModifier
(used by modifier)
: "tailrec"
: "operator"
: "infix"
: "inline"
: "external"
: suspendModifier
;
propertyModifier
(used by modifier)
: "const"
;
suspendModifier
(used by typeModifiers, functionModifier)
: "suspend"
;

```

Annotations

```

annotations
(used by catchBlock, prefixUnaryOperation, blockLevelExpression, for, typeModifiers, class, modifiers, typeConstraint)
: (annotation | annotationList)*
;
annotation
(used by annotations)
: "@" (annotationUseSiteTarget ":")? unescapedAnnotation
;
annotationList
(used by annotations)
: "@" (annotationUseSiteTarget ":")? "[" unescapedAnnotation + "]"
;
annotationUseSiteTarget
(used by annotation, annotationList)
: "field"
: "file"
: "property"
: "get"
: "set"
: "receiver"
: "param"
: "setparam"
: "delegate"
;
unescapedAnnotation
(used by annotation, fileAnnotation, annotatedLambda, annotationList)
: SimpleName {"."} typeArguments? valueArguments?
;

```

Lexical structure

```

helper
Digit
(used by IntegerLiteral, HexDigit)
: ["0".."9"];
IntegerLiteral
(used by literalConstant)
: Digit (Digit | "_" )*
FloatLiteral
(used by literalConstant)
: <Java double literal>;
helper
HexDigit
(used by RegularStringPart, HexadecimalLiteral)
: Digit | ["A".."F", "a".."f"];
HexadecimalLiteral
(used by literalConstant)
: "0x" HexDigit (HexDigit | "_" )*;
CharacterLiteral
(used by literalConstant)
: <character as in Java>;
See Basic types

NoEscapeString
(used by literalConstant)
: <""""quoted string>;
RegularStringPart
(used by stringTemplateElement)
: <any character other than backslash, quote, $ or newline>
ShortTemplateEntryStart:
: "$"
EscapeSequence:
: UnicodeEscapeSequence | RegularEscapeSequence
UnicodeEscapeSequence:
: "\u" HexDigit{4}
RegularEscapeSequence:
: "\" <any character other than newline>
See String templates

```

SEMI

(used by [whenEntry](#), [if](#), [statements](#), [packageHeader](#), [property](#), [import](#))

: <semicolon or newline>;

SimpleName

(used by [typeParameter](#), [catchBlock](#), [simpleUserType](#), [atomicExpression](#), [LabelName](#), [packageHeader](#), [class](#), [object](#), [infixFunctionCall](#), [function](#), [typeAlias](#), [parameter](#), [callableReference](#), [variableDeclarationEntry](#), [stringTemplateElement](#), [enumEntry](#), [setter](#), [import](#), [companionObject](#), [valueArguments](#), [unescapedAnnotation](#), [typeConstraint](#))

: <java identifier>

: "\"" <java identifier> "\""

;

See [Java interoperability](#)

LabelName

(used by [labelReference](#), [labelDefinition](#))

: "@" [SimpleName](#);

See [Returns and jumps](#)

Compatibility

This page describes compatibility guarantees for different versions and subsystems of Kotlin.

Compatibility glossary

Compatibility means answering the question: for given two versions of Kotlin (for example, 1.2 and 1.1.5), can the code written for one version be used with another version? The list below explains the compatibility modes of different pairs of versions. Note that a version is older if it has a smaller version number (even if it was released later than the version with a larger version number). We use OV for "Older Version", and NV for "Newer Version".

- **C** - Full **C**ompatibility
 - Language
 - no syntax changes (*modulo bugs**)
 - new warnings/hints may be added or removed
 - API (`kotlin-stdlib-*`, `kotlin-reflect-*`)
 - no API changes
 - deprecations with level `WARNING` may be added/removed
 - Binaries (ABI)
 - runtime: binaries can be used interchangeably
 - compilation: binaries can be used interchangeably
- **BCLA** - **B**ackward **C**ompatibility for the **L**anguage and **A**PI
 - Language
 - syntax deprecated in OV may be removed in NV
 - other than that, all code compilable in OV is compilable by in NV (*modulo bugs**)
 - new syntax may be added in NV
 - some restrictions of OV may be lifted in NV
 - new warnings/hints may be added or removed
 - API (`kotlin-stdlib-*`, `kotlin-reflect-*`)
 - new APIs may be added
 - deprecations with level `WARNING` may be added/removed
 - deprecations with level `WARNING` may be elevated to level `ERROR` or `HIDDEN` in NV
- **BCB** - **B**ackward **C**ompatibility for **B**inaries
 - Binaries (ABI)
 - runtime: NV-binaries can be used everywhere where OV binaries worked
 - NV compiler: code compilable against OV binaries is compilable against NV binaries
 - OV compiler may not accept NV binaries (e.g. those that exhibit newer language features or APIs)
- **BC** - Full **B**ackward **C**ompatibility
 - BC = BCLA & BCB
- **EXP** - Experimental feature
 - see [below](#)
- **NO** - No compatibility guarantees
 - we'll do our best to offer smooth migration, but can give no guarantees
 - migration is planned individually for every incompatible subsystem

* No changes *modulo bugs* means that if an important bug is found (e.g. in the compiler diagnostics or elsewhere), a fix for it may introduce a breaking change, but we are always very careful with such changes.

Compatibility guarantees for Kotlin releases

Kotlin for JVM:

- patch version updates (e.g. 1.1.X) are fully compatible
- minor version updates (e.g. 1.X) are backwards compatible

Kotlin	1.0	1.0.X	1.1	1.1.X	...	2.0
1.0	-	C	BC	BC	...	?
1.0.X	C	-	BC	BC	...	?
1.1	BC	BC	-	C	...	?
1.1.X	BC	BC	C	-	...	?
...
2.0	?	?	?	?	...	-

Kotlin for JS: starting with Kotlin 1.1, both patch and minor version updates provide backward compatibility for the language and API (BCLA), but no BCB.

Kotlin	1.0.X	1.1	1.1.X	...	2.0
1.0.X	-	EXP	EXP	...	EXP
1.1	EXP	-	BCLA	...	?
1.1.X	EXP	BCLA	-	...	?
...
2.0	EXP	?	?	...	-

Kotlin Scripts: both patch and minor version updates provide backward compatibility for the language and API (BCLA), but no BCB.

Compatibility across platforms

Kotlin is available for several platforms (JVM/Android, JavaScript and the upcoming native platforms). Every platform has its own peculiarities (e.g. JavaScript has no proper integers), so we have to adapt the language accordingly. Our goal is to provide reasonable code portability without sacrificing too much.

Every platform may feature specific language extensions (such as platform types for JVM and dynamic types for JavaScript) or restrictions (e.g. some overloading-related restrictions on the JVM), but the core language remains the same.

The Standard Library provides a core API available on all platforms, and we strive to make these APIs work in the same way on every platform. Along with these, the Standard Library provides platform-specific extensions (e.g. `java.io` for JVM, or `js()` for JavaScript), plus some APIs that can be called uniformly, but work differently (such as regular expressions for JVM and JavaScript).

Experimental features

Experimental features, such as coroutines in Kotlin 1.1, have exemption from the compatibility modes listed above. Such features require an opt-in to use without compiler warning. Experimental features are at least backwards compatible for patch version updates, but we do not guarantee any compatibility for minor version updates (migration aids will be provided where possible).

Kotlin	1.1	1.1.X	1.2	1.2.X
1.1	-	BC	NO	NO
1.1.X	BC	-	NO	NO
1.2	NO	NO	-	BC
1.2.X	NO	NO	BC	-

EAP builds

We publish Early Access Preview (EAP) builds to special channels where early adopters from the community can try them out and give us feedback. Such builds provide no compatibility guarantees whatsoever (although we do our best to keep them reasonably compatible with releases and with each other). Quality expectations for such builds are also much lower than for releases. Beta builds also fall under this category.

IMPORTANT NOTE: all binaries compiled by EAP builds for 1.X (e.g. 1.1.0-eap-X) are **rejected by release builds of the compiler**. We don't want any code compiled by pre-release versions to be kept around after a stable version is released. This does not concern EAPs of patch versions (e.g. 1.1.3-eap-X), these EAPs produce builds with stable ABI.

Compatibility modes

When a big team is migrating onto a new version, it may appear in a "inconsistent state" at some point, when some developers have already updated, and others haven't. To prevent the former from writing and committing code that others may not be able to compile, we provide the following command line switches (also available in the IDE and [Gradle/Maven](#)):

- `-language-version X.Y` - compatibility mode for Kotlin language version X.Y, reports errors for all language features that came out later
- `-api-version X.Y` - compatibility mode for Kotlin API version X.Y, reports errors for all code using newer APIs from the Kotlin Standard Library (including the code generated by the compiler).

Binary compatibility warnings

If you use the NV Kotlin compiler and have the OV standard library or the OV reflection library in the classpath, it can be a sign that the project is misconfigured. To prevent unexpected problems during compilation or at runtime, we suggest either updating the dependencies to NV, or specifying the API version / language version arguments explicitly. Otherwise the compiler detects that something can go wrong and reports a warning.

For example, if OV = 1.0 and NV = 1.1, you can observe one of the following warnings:

```
Runtime JAR files in the classpath have the version 1.0, which is older than the API version 1.1.
Consider using the runtime of version 1.1, or pass '-api-version 1.0' explicitly to restrict the
available APIs to the runtime of version 1.0.
```

This means that you're using the Kotlin compiler 1.1 against the standard or reflection library of version 1.0. This can be handled in different ways:

- If you intend to use the APIs from the 1.1 Standard Library, or language features that depend on those APIs, you should upgrade the dependency to the version 1.1.
- If you want to keep your code compatible with the 1.0 standard library, you can pass `-api-version 1.0`.
- If you've just upgraded to Kotlin 1.1 but can not use new language features yet (e.g. because some of your teammates may not have upgraded), you can pass `-language-version 1.0`, which will restrict all APIs and language features to 1.0.

```
Runtime JAR files in the classpath should have the same version. These files were found in the classpath:
    kotlin-reflect.jar (version 1.0)
    kotlin-stdlib.jar (version 1.1)
Consider providing an explicit dependency on kotlin-reflect 1.1 to prevent strange errors
Some runtime JAR files in the classpath have an incompatible version. Consider removing them from the
classpath
```

This means that you have a dependency on libraries of different versions, for example the 1.1 standard library and the 1.0 reflection library. To prevent subtle errors at runtime, we recommend you to use the same version of all Kotlin libraries. In this case, consider adding an explicit dependency on the 1.1 reflection library.

```
Some JAR files in the classpath have the Kotlin Runtime library bundled into them.
This may cause difficult to debug problems if there's a different version of the Kotlin Runtime library
in the classpath.
Consider removing these libraries from the classpath
```

This means that there's a library in the classpath which does not depend on the Kotlin standard library as a Gradle/Maven dependency, but is distributed in the same artifact with it (i.e. has it *bundled*). Such a library may cause issues because standard build tools do not consider it an instance of the Kotlin standard library, thus it's not subject to the dependency version resolution mechanisms, and you can end up with several versions of the same library in the classpath. Consider contacting the authors of such a library and suggesting to use the Gradle/Maven dependency instead.

Java Interop

Calling Java code from Kotlin

Kotlin is designed with Java Interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well. In this section we describe some details about calling Java code from Kotlin.

Pretty much all Java code can be used without any issues

```
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 and Setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with `get` and single-argument methods with names starting with `set`) are represented as properties in Kotlin. For example:

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
    }
}
```

Note that, if the Java class only has a setter, it will not be visible as a property in Kotlin, because Kotlin does not support set-only properties at this time.

Methods returning void

If a Java method returns void, it will return `Unit` when called from Kotlin. If, by any chance, someone uses that return value, it will be assigned at the call site by the Kotlin compiler, since the value itself is known in advance (being `Unit`).

Escaping for Java identifiers that are keywords in Kotlin

Some of the Kotlin keywords are valid identifiers in Java: `in`, `object`, `is`, etc. If a Java library uses a Kotlin keyword for a method, you can still call the method escaping it with the backtick (```) character

```
foo.`is`(bar)
```

Null-Safety and Platform Types

Any reference in Java may be `null`, which makes Kotlin's requirements of strict null-safety impractical for objects coming from Java. Types of Java declarations are treated specially in Kotlin and called *platform types*. Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java (see more [below](#)).

Consider the following examples:

```
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)
```

When we call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```
item.substring(1) // allowed, may throw an exception if item == null
```

Platform types are *non-denotable*, meaning that one can not write them down explicitly in the language. When a platform value is assigned to a Kotlin variable, we can rely on type inference (the variable will have an inferred platform type then, as `item` has in the example above), or we can choose the type that we expect (both nullable and non-null types are allowed):

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```

If we choose a non-null type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-null variables from holding nulls. Assertions are also emitted when we pass platform values to Kotlin functions expecting non-null values etc. Overall, the compiler does its best to prevent nulls from propagating far through the program (although sometimes this is impossible to eliminate entirely, because of generics).

Notation for Platform Types

As mentioned above, platform types cannot be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (in error messages, parameter info etc), so we have a mnemonic notation for them:

- `T!` means "`T` or `T?`",
- `(Mutable)Collection<T>!` means "Java collection of `T` may be mutable or not, may be nullable or not",
- `Array<(out) T>!` means "Java array of `T` (or a subtype of `T`), nullable or not"

Nullability annotations

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](#).

Mapped types

Kotlin treats some Java types specially. Such types are not loaded from Java "as is", but are *mapped* to corresponding Kotlin types. The mapping only matters at compile time, the runtime representation remains unchanged. Java's primitive types are mapped to corresponding Kotlin types (keeping [platform types](#) in mind):

Java type	Kotlin type

Java type	Kotlin type
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Some non-primitive built-in classes are also mapped:

Java type	Kotlin type
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.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Java's boxed primitive types are mapped to nullable Kotlin types:

Java type	Kotlin type
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Char	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

Note that a boxed primitive type used as a type parameter is mapped to a platform type: for example, `List<java.lang.Integer>` becomes a `List<Int!>` in Kotlin.

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package `kotlin.collections`):

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Iterable<T>!
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's arrays are mapped as mentioned [below](#):

Java type	Kotlin type
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

Java generics in Kotlin

Kotlin's generics are a little different from Java's (see [Generics](#)). When importing Java types to Kotlin we perform some conversions:

- Java's wildcards are converted into type projections
 - `Foo<? extends Bar>` becomes `Foo<out Bar!>!`
 - `Foo<? super Bar>` becomes `Foo<in Bar!>!`
- Java's raw types are converted into star projections
 - `List` becomes `List<*>!`, i.e. `List<out Any?>!`

Like Java's, Kotlin's generics are not retained at runtime, i.e. objects do not carry information about actual type arguments passed to their constructors, i.e. `ArrayList<Integer>()` is indistinguishable from `ArrayList<Character>()`. This makes it impossible to perform `is`-checks that take generics into account. Kotlin only allows `is`-checks for star-projected generic types:

```
if (a is List<Int>) // Error: cannot check if it is really a List of Ints
// but
if (a is List<*>) // OK: no guarantees about the contents of the list
```

Java Arrays

Arrays in Kotlin are invariant, unlike Java. This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure. Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed (through [platform types](#) of the form `Array<(out) String!>`).

Arrays are used with primitive datatypes on the Java platform to avoid the cost of boxing/unboxing operations. As Kotlin hides those implementation details, a workaround is required to interface with Java code. There are specialized classes for every type of primitive array (`IntArray`, `DoubleArray`, `CharArray`, and so on) to handle this case. They are not related to the `Array` class and are compiled down to Java's primitive arrays for maximum performance.

Suppose there is a Java method that accepts an int array of indices:

```
public class JavaArrayExample {
    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

To pass an array of primitive values you can do the following in Kotlin:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array) // passes int[] to method
```

When compiling to JVM byte codes, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = arrayOf(1, 2, 3, 4)
array[x] = array[x] * 2 // no actual calls to get() and set() generated
for (x in array) { // no iterator created
    print(x)
}
```

Even when we navigate with an index, it does not introduce any overhead

```
for (i in array.indices) { // no iterator created
    array[i] += 2
}
```

Finally, `in`-checks have no overhead either

```
if (i in array.indices) { // same as (i >= 0 && i < array.size)
    print(array[i])
}
```

Java Varargs

Java classes sometimes use a method declaration for the indices with a variable number of arguments (varargs).

```
public class JavaArrayExample {
    public void removeIndices(int... indices) {
        // code here...
    }
}
```

In that case you need to use the spread operator `*` to pass the `IntArray`:

```
val javaObj = JavaArray()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

It's currently not possible to pass `null` to a method that is declared as varargs.

Operators

Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (`invoke()` etc.) Calling Java methods using the infix call syntax is not allowed.

Checked Exceptions

In Kotlin, all exceptions are unchecked, meaning that the compiler does not force you to catch any of them. So, when you call a Java method that declares a checked exception, Kotlin does not force you to do anything:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java would require us to catch IOException here
    }
}
```

Object Methods

When Java types are imported into Kotlin, all the references of the type `java.lang.Object` are turned into `Any`. Since `Any` is not platform-specific, it only declares `toString()`, `hashCode()` and `equals()` as its members, so to make other members of `java.lang.Object` available, Kotlin uses [extension functions](#).

`wait()/notify()`

[Effective Java](#) Item 69 kindly suggests to prefer concurrency utilities to `wait()` and `notify()`. Thus, these methods are not available on references of type `Any`. If you really need to call them, you can cast to `java.lang.Object`:

```
(foo as java.lang.Object).wait()
```

getClass()

To retrieve the Java class of an object, use the `java` extension property on a [class reference](#).

```
val fooClass = foo::class.java
```

The code above uses a [bound class reference](#), which is supported since Kotlin 1.1. You can also use the `javaClass` extension property.

```
val fooClass = foo.javaClass
```

clone()

To override `clone()`, your class needs to extend `kotlin.Cloneable`:

```
class Example : Cloneable {  
    override fun clone(): Any { ... }  
}
```

Do not forget about [Effective Java](#), Item 11: *Override clone judiciously*.

finalize()

To override `finalize()`, all you need to do is simply declare it, without using the `override` keyword:

```
class C {  
    protected fun finalize() {  
        // finalization logic  
    }  
}
```

According to Java's rules, `finalize()` must not be `private`.

Inheritance from Java classes

At most one Java class (and as many Java interfaces as you like) can be a supertype for a class in Kotlin.

Accessing static members

Static members of Java classes form "companion objects" for these classes. We cannot pass such a "companion object" around as a value, but can access the members explicitly, for example

```
if (Character.isLetter(a)) {  
    // ...  
}
```

Java Reflection

Java reflection works on Kotlin classes and vice versa. As mentioned above, you can use `instance::class.java`, `ClassName::class.java` or `instance.javaClass` to enter Java reflection through `java.lang.Class`.

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 Conversions

Just like Java 8, Kotlin supports SAM conversions. This means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

You can use this for creating instances of SAM interfaces:

```
val runnable = Runnable { println("This runs in a runnable") }
```

...and in method calls:

```
val executor = ThreadPoolExecutor()  
// Java signature: void execute(Runnable command)  
executor.execute { println("This runs in a thread pool") }
```

If the Java class has multiple methods taking functional interfaces, you can choose the one you need to call by using an adapter function that converts a lambda to a specific SAM type. Those adapter functions are also generated by the compiler when needed.

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

Note that SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Also note that this feature works only for Java interop; since Kotlin has proper function types, automatic conversion of functions into implementations of Kotlin interfaces is unnecessary and therefore unsupported.

Using JNI with Kotlin

To declare a function that is implemented in native (C or C++) code, you need to mark it with the `external` modifier:

```
external fun foo(x: Int): Double
```

The rest of the procedure works in exactly the same way as in Java.

Calling Kotlin from Java

Kotlin code can be called from Java easily.

Properties

A Kotlin property is compiled to the following Java elements:

- A getter method, with the name calculated by prepending the `get` suffix;
- A setter method, with the name calculated by prepending the `set` suffix (only for `var` properties);
- A private field, with the same name as the property name (only for properties with backing fields).

For example, `var firstName: String` gets compiled to the following Java declarations:

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

If the name of the property starts with `is`, a different name mapping rule is used: the name of the getter will be the same as the property name, and the name of the setter will be obtained by replacing `is` with `set`. For example, for a property `isOpen`, the getter will be called `isOpen()` and the setter will be called `setOpen()`. This rule applies for properties of any type, not just `Boolean`.

Package-Level Functions

All the functions and properties declared in a file `example.kt` inside a package `org.foo.bar`, including extension functions, are compiled into static methods of a Java class named `org.foo.bar.ExampleKt`.

```
// example.kt
package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

The name of the generated Java class can be changed using the `@JvmName` annotation:

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() {
}
```



```
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

Having multiple files which have the same generated Java class name (the same package and the same name or the same `@JvmName` annotation) is normally an error. However, the compiler has the ability to generate a single Java facade class which has the specified name and contains all the declarations from all the files which have that name. To enable the generation of such a facade, use the `@JvmMultifileClass` annotation in all of the files.

```
// 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();
```

Instance Fields

If you need to expose a Kotlin property as a field in Java, you need to annotate it with the `@JvmField` annotation. The field will have the same visibility as the underlying property. You can annotate a property with `@JvmField` if it has a backing field, is not private, does not have `open`, `override` or `const` modifiers, and is not a delegated property.

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

[Late-Initialized](#) properties are also exposed as fields. The visibility of the field will be the same as the visibility of `lateinit` property setter.

Static Fields

Kotlin properties declared in a named object or a companion object will have static backing fields either in that named object or in the class containing the companion object.

Usually these fields are private but they can be exposed in one of the following ways:

- `@JvmField` annotation;

- `lateinit` modifier;
- `const` modifier.

Annotating such a property with `@JvmField` makes it a static field with the same visibility as the property itself.

```
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
```

A [late-initialized](#) property in an object or a companion object has a static backing field with the same visibility as the property setter.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

Properties annotated with `const` (in classes as well as at the top level) are turned into static fields in 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;
```

Static Methods

As mentioned above, Kotlin represents package-level functions as static methods. Kotlin can also generate static methods for functions defined in named objects or companion objects if you annotate those functions as `@JvmStatic`. If you use this annotation, the compiler will generate both a static method in the enclosing class of the object and an instance method in the object itself. For example:

```
class C {
    companion object {
        @JvmStatic fun foo() {}
        fun bar() {}
    }
}
```

Now, `foo()` is static in Java, while `bar()` is not:

```
C.foo(); // works fine
C.bar(); // error: not a static method
C.Companion.foo(); // instance method remains
C.Companion.bar(); // the only way it works
```

Same for named objects:

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

In Java:

```
Obj.foo(); // works fine
Obj.bar(); // error
Obj.INSTANCE.bar(); // works, a call through the singleton instance
Obj.INSTANCE.foo(); // works too
```

`@JvmStatic` annotation can also be applied on a property of an object or a companion object making its getter and setter methods be static members in that object or the class containing the companion object.

Visibility

The Kotlin visibilities are mapped to Java in the following way:

- `private` members are compiled to `private` members;
- `private` top-level declarations are compiled to package-local declarations;
- `protected` remains `protected` (note that Java allows accessing protected members from other classes in the same package and Kotlin doesn't, so Java classes will have broader access to the code);
- `internal` declarations become `public` in Java. Members of `internal` classes go through name mangling, to make it harder to accidentally use them from Java and to allow overloading for members with the same signature that don't see each other according to Kotlin rules;
- `public` remains `public`.

KClass

Sometimes you need to call a Kotlin method with a parameter of type `KClass`. There is no automatic conversion from `Class` to `KClass`, so you have to do it manually by invoking the equivalent of the `Class<T>.kotlin` extension property:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

Handling signature clashes with @JvmName

Sometimes we have a named function in Kotlin, for which we need a different JVM name the byte code. The most prominent example happens due to *type erasure*:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

These two functions can not be defined side-by-side, because their JVM signatures are the same:

`filterValid(Ljava/util/List;)Ljava/util/List;`. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with `@JvmName` and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

From Kotlin they will be accessible by the same name `filterValid`, but from Java it will be `filterValid` and `filterValidInt`.

The same trick applies when we need to have a property `x` alongside with a function `getX()`:

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

Overloads Generation

Normally, if you write a Kotlin method with default parameter values, it will be visible in Java only as a full signature, with all parameters present. If you wish to expose multiple overloads to Java callers, you can use the `@JvmOverloads` annotation.

```
@JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
    ...
}
```

For every parameter with a default value, this will generate one additional overload, which has this parameter and all parameters to the right of it in the parameter list removed. In this example, the following methods will be generated:

```
// Java
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

The annotation also works for constructors, static methods etc. It can't be used on abstract methods, including methods defined in interfaces.

Note that, as described in [Secondary Constructors](#), if a class has default values for all constructor parameters, a public no-argument constructor will be generated for it. This works even if the `@JvmOverloads` annotation is not specified.

Checked Exceptions

As we mentioned above, Kotlin does not have checked exceptions. So, normally, the Java signatures of Kotlin functions do not declare exceptions thrown. Thus if we have a function in Kotlin like this:

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

And we want to call it from Java and catch the exception:

```
// Java
try {
    demo.Example.foo();
}
catch (IOException e) { // error: foo() does not declare IOException in the throws list
    // ...
}
```

we get an error message from the Java compiler, because `foo()` does not declare `IOException`. To work around this problem, use the `@Throws` annotation in Kotlin:

```
@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

Null-safety

When calling Kotlin functions from Java, nobody prevents us from passing `null` as a non-null parameter. That's why Kotlin generates runtime checks for all public functions that expect non-nulls. This way we get a `NullPointerException` in the Java code immediately.

Variant generics

When Kotlin classes make use of [declaration-site variance](#), there are two options of how their usages are seen from the Java code. Let's say we have the following class and two functions that use it:

```
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
```

A naive way of translating these functions into Java would be this:

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

The problem is that in Kotlin we can say `unboxBase(boxDerived("s"))`, 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) { ... }
```

Here we make use of Java's *wildcards types* (`? extends Base`) to emulate declaration-site variance through use-site variance, because it is all Java has.

To make Kotlin APIs work in Java we generate `Box<Super>` as `Box<? extends Super>` for covariantly defined `Box` (or `Foo<? super Bar>` for contravariantly defined `Foo`) when it appears *as a parameter*. When it's a return value, we don't generate wildcards, because otherwise Java clients will have to deal with them (and it's against the common Java coding style). Therefore, the functions from our example are actually translated as follows:

```
// return type - no wildcards
Box<Derived> boxDerived(Derived value) { ... }

// parameter - wildcards
Base unboxBase(Box<? extends Base> box) { ... }
```

NOTE: when the argument type is final, there's usually no point in generating the wildcard, so `Box<String>` is always `Box<String>`, no matter what position it takes.

If we need wildcards where they are not generated by default, we can use the `@JvmWildcard` annotation:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// is translated to
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

On the other hand, if we don't need wildcards where they are generated, we can use `@JvmSuppressWildcards`:

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// is translated to
// Base unboxBase(Box<Base> box) { ... }
```

NOTE: `@JvmSuppressWildcards` can be used not only on individual type arguments, but on entire declarations, such as functions or classes, causing all wildcards inside them to be suppressed.

Translation of type `Nothing`

The type `Nothing` is special, because it has no natural counterpart in Java. Indeed, every Java reference type, including `java.lang.Void`, accepts `null` as a value, and `Nothing` doesn't accept even that. So, this type cannot be accurately represented in the Java world. This is why Kotlin generates a raw type where an argument of type `Nothing` is used:

```
fun emptyList(): List<Nothing> = listOf()
// is translated to
// List emptyList() { ... }
```

JavaScript

Kotlin JavaScript Overview

Kotlin provides the ability to target JavaScript. It does so by transpiling Kotlin to JavaScript. The current implementation targets ECMAScript 5.1 but there are plans to eventually target ECMAScript 2015 also.

When you choose the JavaScript target, any Kotlin code that is part of the project as well as the standard library that ships with Kotlin is transpiled to JavaScript. However, this excludes the JDK and any JVM or Java framework or library used. Any file that is not Kotlin will be ignored during compilation.

The Kotlin compiler tries to comply with the following goals:

- Provide output that is optimal in size
- Provide output that is readable JavaScript
- Provide interoperability with existing module systems
- Provide the same functionality in the standard library whether targeting JavaScript or the JVM (to the largest possible degree).

How can it be used

You may want to compile Kotlin to JavaScript in the following scenarios:

- Creating Kotlin code that targets client-side JavaScript
 - **Interacting with DOM elements.** Kotlin provides a series of statically typed interfaces to interact with the Document Object Model, allowing creation and update of DOM elements.
 - **Interacting with graphics such as WebGL.** You can use Kotlin to create graphical elements on a web page using WebGL.
- Creating Kotlin code that targets server-side JavaScript
 - **Working with server-side technology.** You can use Kotlin to interact with server-side JavaScript such as node.js

Kotlin can be used together with existing third-party libraries and frameworks, such as JQuery or ReactJS. To access third-party frameworks with a strongly-typed API, you can convert TypeScript definitions from the [Definitely Typed](#) type definitions repository to Kotlin using the [ts2kt](#) tool. Alternatively, you can use the [dynamic type](#) to access any framework without strong typing.

Kotlin is also compatible with CommonJS, AMD and UMD, [making interaction with different](#) module systems straightforward.

Getting Started with Kotlin to JavaScript

To find out how to start using Kotlin for JavaScript, please refer to the [tutorials](#).

Dynamic Type

⚠ The dynamic type is not supported in code targeting the JVM

Being a statically typed language, Kotlin still has to interoperate with untyped or loosely typed environments, such as the JavaScript ecosystem. To facilitate these use cases, the `dynamic` type is available in the language:

```
val dyn: dynamic = ...
```

The `dynamic` type basically turns off Kotlin's type checker:

- a value of this type can be assigned to any variable or passed anywhere as a parameter,
- any value can be assigned to a variable of type `dynamic` or passed to a function that takes `dynamic` as a parameter,
- `null`-checks are disabled for such values.

The most peculiar feature of `dynamic` is that we are allowed to call **any** property or function with any parameters on a `dynamic` variable:

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere
dyn.whatever(*arrayOf(1, 2, 3))
```

On the JavaScript platform this code will be compiled "as is": `dyn.whatever(1)` in Kotlin becomes `dyn.whatever(1)` in the generated JavaScript code.

When calling functions written in Kotlin on values of `dynamic` type, keep in mind the name mangling performed by the Kotlin to JavaScript compiler. You may need to use the [@JsName annotation](#) to assign well-defined names to the functions that you need to call.

A dynamic call always returns `dynamic` as a result, so we can chain such calls freely:

```
dyn.foo().bar.baz()
```

When we pass a lambda to a dynamic call, all of its parameters by default have the type `dynamic`:

```
dyn.foo {
    x -> x.bar() // x is dynamic
}
```

Expressions using values of `dynamic` type are translated to JavaScript "as is", and do not use the Kotlin operator conventions. The following operators are supported:

- binary: `+`, `-`, `*`, `/`, `%`, `>`, `<`, `>=`, `<=`, `==`, `!=`, `===`, `!==`, `&&`, `||`
- unary
 - prefix: `-`, `+`, `!`
 - prefix and postfix: `++`, `--`
- assignments: `+=`, `-=`, `*=`, `/=`, `%=`
- indexed access:
 - read: `d[a]`, more than one argument is an error
 - write: `d[a1] = a2`, more than one argument in `[]` is an error

`in`, `!in` and `..` operations with values of type `dynamic` are forbidden.

For a more technical description, see the [spec document](#).

Calling JavaScript from Kotlin

Kotlin was designed for easy interoperability with the Java platform. It sees Java classes as Kotlin classes, and Java sees Kotlin classes as Java classes. However, JavaScript is a dynamically-typed language, which means it does not check types in compile-time. You can freely talk to JavaScript from Kotlin via [dynamic](#) types, but if you want the full power of the Kotlin type system, you can create Kotlin headers for JavaScript libraries.

Inline JavaScript

You can inline some JavaScript code into your Kotlin code using the `js("...")` function. For example:

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

The parameter of `js` is required to be a string constant. So, the following code is incorrect:

```
fun jsTypeOf(o: Any): String {
    return js(getTypeOf() + " o") // error reported here
}
fun getTypeOf() = "typeof"
```

external modifier

To tell Kotlin that a certain declaration is written in pure JavaScript, you should mark it with `external` modifier. When the compiler sees such a declaration, it assumes that the implementation for the corresponding class, function or property is provided by the developer, and therefore does not try to generate any JavaScript code from the declaration. This means that you should omit bodies of `external` declarations. For example:

```
external fun alert(message: Any?): Unit

external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}

external val window: Window
```

Note that `external` modifier is inherited by nested declarations, i.e. in `Node` class we do not put `external` before member functions and properties.

The `external` modifier is only allowed on package-level declarations. You can't declare an `external` member of a non-`external` class.

Declaring (static) members of a class

In JavaScript you can define members either on a prototype or a class itself. I.e.:

```
function MyClass() {
}
MyClass.sharedMember = function() { /* implementation */ };
MyClass.prototype.ownMember = function() { /* implementation */ };
```

There's no such syntax in Kotlin. However, in Kotlin we have `companion` objects. Kotlin treats companion objects of `external` class in a special way: instead of expecting an object, it assumes members of companion objects to be members of the class itself. To describe `MyClass` from the example above, you can write:

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

Declaring optional parameters

An external function can have optional parameters. How the JavaScript implementation actually computes default values for these parameters, is unknown to Kotlin, thus it's impossible to use the usual syntax to declare such parameters in Kotlin. You should use the following syntax:

```
external fun myFunWithOptionalArgs(x: Int,
    y: String = definedExternally,
    z: Long = definedExternally)
```

This means you can call `myFunWithOptionalArgs` with one required argument and two optional arguments (their default values are calculated by some JavaScript code).

Extending JavaScript classes

You can easily extend JavaScript classes as they were Kotlin classes. Just define an `external` class and extend it by non-`external` class. For example:

```
external open class HTMLElement : Element() {
    /* members */
}

class CustomElement : HTMLElement() {
    fun foo() {
        alert("bar")
    }
}
```

There are some limitations:

1. When a function of external base class is overloaded by signature, you can't override it in a derived class.
2. You can't override a function with default arguments.

Note that you can't extend a non-external class by external classes.

external interfaces

JavaScript does not have the concept of interfaces. When a function expects its parameter to support `foo` and `bar` methods, you just pass objects that actually have these methods. You can use interfaces to express this for statically-typed Kotlin, for example:

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

Another use case for external interfaces is to describe settings objects. For example:

```

external interface JQueryAjaxSettings {
    var async: Boolean

    var cache: Boolean

    var complete: (JQueryXHR, String) -> Unit

    // etc
}

fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")

external class JQuery {
    companion object {
        fun get(settings: JQueryAjaxSettings): JQueryXHR
    }
}

fun sendQuery() {
    JQuery.get(JQueryAjaxSettings()).apply {
        complete = { (xhr, data) ->
            window.alert("Request complete")
        }
    })
}

```

External interfaces have some restrictions:

1. They can't be used on the right hand side of `is` checks.
2. `as` cast to external interface always succeeds (and produces a warning in compile-time).
3. They can't be passed as reified type arguments.
4. They can't be used in class literal expression (i.e. `I::class`).

Calling Kotlin from JavaScript

Kotlin compiler generates normal JavaScript classes, functions and properties you can freely use from JavaScript code. Nevertheless, there are some subtle things you should remember.

Isolating declarations in a separate JavaScript object

To prevent spoiling the global object, Kotlin creates an object that contains all Kotlin declarations from the current module. So if you name your module as `myModule`, all declarations are available to JavaScript via `myModule` object. For example:

```
fun foo() = "Hello"
```

Can be called from JavaScript like this:

```
alert(myModule.foo());
```

This is not applicable when you compile your Kotlin module to JavaScript module (see [JavaScript Modules](#) for more information on this). In this case there won't be a wrapper object, instead, declarations will be exposed as a JavaScript module of a corresponding kind. For example, in case of CommonJS you should write:

```
alert(require('myModule').foo());
```

Package structure

Kotlin exposes its package structure to JavaScript, so unless you define your declarations in the root package, you have to use fully-qualified names in JavaScript. For example:

```
package my.qualified.packagename

fun foo() = "Hello"
```

Can be called from JavaScript like this:

```
alert(myModule.my.qualified.packagename.foo());
```

@JsName annotation

In some cases (for example, to support overloads), Kotlin compiler mangles names of generated functions and attributes in JavaScript code. To control the generated names, you can use the `@JsName` annotation:

```
// Module 'kjs'
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

Now you can use this class from JavaScript in the following way:

```
var person = new kjs.Person("Dmitry"); // refers to module 'kjs'
person.hello(); // prints "Hello Dmitry!"
person.helloWithGreeting("Servus"); // prints "Servus Dmitry!"
```

If we didn't specify the `@JsName` annotation, the name of the corresponding function would contain a suffix calculated from the function signature, for example `hello_61zpoes$`.

Note that Kotlin compiler does not apply such mangling to `external` declarations, so you don't have to use `@JsName` on them. Another case worth noticing is inheriting non-external classes from external classes. In this case any overridden functions won't be mangled as well.

The parameter of `@JsName` is required to be a constant string literal which is a valid identifier. The compiler will report an error on any attempt to pass non-identifier string to `@JsName`. The following example produces a compile-time error:

```
@JsName("new C()") // error here
external fun newC()
```

Representing Kotlin types in JavaScript

- Kotlin numeric types, except for `kotlin.Long` are mapped to JavaScript Number.
- `kotlin.Char` is mapped to JavaScript Number representing character code.
- Kotlin can't distinguish between numeric types at run time (except for `kotlin.Long`), i.e. the following code works:

```
fun f() {
    val x: Int = 23
    val y: Any = x
    println(y as Float)
}
```

- Kotlin preserves overflow semantics for `kotlin.Int`, `kotlin.Byte`, `kotlin.Short`, `kotlin.Char` and `kotlin.Long`.
- There's no 64 bit integer number in JavaScript, so `kotlin.Long` is not mapped to any JavaScript object, it's emulated by a Kotlin class.
- `kotlin.String` is mapped to JavaScript String.
- `kotlin.Any` is mapped to JavaScript Object (i.e. `new Object()`, `{}`, etc).
- `kotlin.Array` is mapped to JavaScript Array.
- Kotlin collections (i.e. `List`, `Set`, `Map`, etc) are not mapped to any specific JavaScript type.
- `kotlin.Throwable` is mapped to JavaScript Error.
- Kotlin preserves lazy object initialization in JavaScript.
- Kotlin does not implement lazy initialization of top-level properties in JavaScript.

JavaScript Modules

Kotlin allows you to compile your Kotlin projects to JavaScript modules for popular module systems. Here is the list of available options:

1. Plain. Don't compile for any module system. As usual, you can access a module by its name in the global scope. This option is used by default.
2. [Asynchronous Module Definition \(AMD\)](#), which is in particular used by require.js library.
3. [CommonJS](#) convention, widely used by node.js/npm (`require` function and `module.exports` object)
4. Unified Module Definitions (UMD), which is compatible with both *AMD* and *CommonJS*, and works as "plain" when neither *AMD* nor *CommonJS* is available at runtime.

Choosing the Target Module System

Choosing the target module system depends on your build environment:

From IntelliJ IDEA

Setup per module: Open File -> Project Structure..., find your module in Modules and select "Kotlin" facet under it. Choose appropriate module system in "Module kind" field.

Setup for the whole project: Open File -> Settings, select "Build, Execution, Deployment" -> "Compiler" -> "Kotlin compiler". Choose appropriate module system in "Module kind" field.

From Maven

To select module system when compiling via Maven, you should set `moduleKind` configuration property, i.e. your `pom.xml` should look like this:

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <goals>
        <goal>js</goal>
      </goals>
    </execution>
  </executions>
  <!-- Insert these lines -->
  <configuration>
    <moduleKind>commonjs</moduleKind>
  </configuration>
  <!-- end of inserted text -->
</plugin>
```

Available values are: `plain`, `amd`, `commonjs`, `umd`.

From Gradle

To select module system when compiling via Gradle, you should set `moduleKind` property, i.e.

```
compileKotlin2Js.kotlinOptions.moduleKind = "commonjs"
```

Available values are similar to Maven.

@JsModule annotation

To tell Kotlin that an `external` class, package, function or property is a JavaScript module, you can use `@JsModule` annotation. Consider you have the following CommonJS module called "hello":

```
module.exports.sayHello = function(name) { alert("Hello, " + name); }
```

You should declare it like this in Kotlin:

```
@JsModule("hello")
external fun sayHello(name: String)
```

Applying @JsModule to packages

Some JavaScript libraries export packages (namespaces) instead of functions and classes. In terms of JavaScript it's an object that has members that *are* classes, functions and properties. Importing these packages as Kotlin objects often looks unnatural. The compiler allows to map imported JavaScript packages to Kotlin packages, using the following notation:

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

external class C
```

where the corresponding JavaScript module is declared like this:

```
module.exports = {
    foo: { /* some code here */ },
    C: { /* some code here */ }
}
```

Important: files marked with `@file:JsModule` annotation can't declare non-external members. The example below produces compile-time error:

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // error here
```

Importing deeper package hierarchies

In the previous example the JavaScript module exports a single package. However, some JavaScript libraries export multiple packages from within a module. This case is also supported by Kotlin, though you have to declare a new `.kt` file for each package you import.

For example, let's make our example a bit more complicated:

```
module.exports = {
    mylib: {
        pkg1: {
            foo: function() { /* some code here */ },
            bar: function() { /* some code here */ }
        },
        pkg2: {
            baz: function() { /* some code here */ }
        }
    }
}
```

To import this module in Kotlin, you have to write two Kotlin source files:

```

@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg1")
package extlib.pkg1

external fun foo()

external fun bar()

```

and

```

@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg2")
package extlib.pkg2

external fun baz()

```

@JsNonModule annotation

When a declaration has `@JsModule`, you can't use it from Kotlin code when you don't compile it to a JavaScript module. Usually, developers distribute their libraries both as JavaScript modules and downloadable `.js` files that you can copy to project's static resources and include via `<script>` element. To tell Kotlin that it's ok to use a `@JsModule` declaration from non-module environment, you should put `@JsNonModule` declaration. For example, given JavaScript code:

```

function topLevelSayHello(name) { alert("Hello, " + name); }
if (module && module.exports) {
    module.exports = topLevelSayHello;
}

```

can be described like this:

```

@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)

```

Notes

Kotlin is distributed with `kotlin.js` standard library as a single file, which is itself compiled as an UMD module, so you can use it with any module system described above. Also it is available on NPM as [kotlin package](#)

JavaScript Reflection

At this time, JavaScript does not support the full Kotlin reflection API. The only supported part of the API is the `::class` syntax which allows you to refer to the class of an instance, or the class corresponding to the given type. The value of a `::class` expression is a stripped-down [KClass](#) implementation that only supports the [simpleName](#) and [isInstance](#) members.

In addition to that, you can use [KClass.js](#) to access the [JsClass](#) instance corresponding to the class. The `JsClass` instance itself is a reference to the constructor function. This can be used to interoperate with JS functions that expect a reference to a constructor.

Examples:

```
class A
class B
class C

inline fun <reified T> foo() {
    println(T::class.simpleName)
}

val a = A()
println(a::class.simpleName) // Obtains class for an instance; prints "A"
println(B::class.simpleName) // Obtains class for a type; prints "B"
println(B::class.js.name)    // prints "B"
foo<C>()                      // prints "C"
```

Tools

Documenting Kotlin Code

The language used to document Kotlin code (the equivalent of Java's JavaDoc) is called **KDoc**. In its essence, KDoc combines JavaDoc's syntax for block tags (extended to support Kotlin's specific constructs) and Markdown for inline markup.

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 Syntax

Just like with JavaDoc, KDoc comments start with `/**` and end with `*/`. Every line of the comment may begin with an asterisk, which is not considered part of the contents of the comment.

By convention, the first paragraph of the documentation text (the block of text until the first blank line) is the summary description of the element, and the following text is the detailed description.

Every block tag begins on a new line and starts with the `@` character.

Here's an example of a class documented using 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 { ... }
}
```

Block Tags

KDoc currently supports the following block tags:

`@param <name>`

Documents a value parameter of a function or a type parameter of a class, property or function. To better separate the parameter name from the description, if you prefer, you can enclose the name of the parameter in brackets. The following two syntaxes are therefore equivalent:

`@param name description.`
`@param[name] description.`

`@return`

Documents the return value of a function.

`@constructor`

Documents the primary constructor of a class.

`@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>`

Documents an exception which can be thrown by a method. Since Kotlin does not have checked exceptions, there is also no expectation that all possible exceptions are documented, but you can still use this tag when it provides useful information for users of the class.

`@sample <identifier>`

Embeds the body of the function with the specified qualified name into the documentation for the current element, in order to show an example of how the element could be used.

`@see <identifier>`

Adds a link to the specified class or method to the **See Also** block of the documentation.

`@author`


Specifies the author of the element being documented.

`@since`

Specifies the version of the software in which the element being documented was introduced.

`@suppress`

Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.

 KDoc does not support the `@deprecated` tag. Instead, please use the `@Deprecated` annotation.

Inline Markup

For inline markup, KDoc uses the regular [Markdown](#) syntax, extended to support a shorthand syntax for linking to other elements in the code.

Linking to Elements

To link to another element (class, method, property or parameter), simply put its name in square brackets:

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.

You can also use qualified names in the links. Note that, unlike JavaDoc, qualified names always use the dot character to separate the components, even before a method name:

Use `[kotlin.reflect.KClass.properties]` to enumerate the properties of the class.

Names in links are resolved using the same rules as if the name was used inside the element being documented. In particular, this means that if you have imported a name into the current file, you don't need to fully qualify it when you use it in a KDoc comment.

Note that KDoc does not have any syntax for resolving overloaded members in links. Since the Kotlin documentation generation tool puts the documentation for all overloads of a function on the same page, identifying a specific overloaded function is not required for the link to work.

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.

Using 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.

Plugin and Versions

The *kotlin-gradle-plugin* compiles Kotlin sources and modules.

The version of Kotlin to use is usually defined as the *kotlin_version* property:

```
buildscript {
    ext.kotlin_version = '<version to use>'

    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

This is not required when using Kotlin Gradle plugin 1.1.1 and above with the [Gradle plugins DSL](#).

Targeting the JVM

To target the JVM, the Kotlin plugin needs to be applied:

```
apply plugin: "kotlin"
```

Or, starting with Kotlin 1.1.1, the plugin can be applied using the [Gradle plugins DSL](#):

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "<version to use>"
}
```

The `version` should be literal in this block, and it cannot be applied from another build script.

Kotlin sources can be mixed with Java sources in the same folder, or in different folders. The default convention is using different folders:

```
project
- src
  - main (root)
    - kotlin
    - java
```

The corresponding *sourceSets* property should be updated if not using the default convention

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

Targeting JavaScript

When targeting JavaScript, a different plugin should be applied:

```
apply plugin: "kotlin2js"
```

This plugin only works for Kotlin files so it is recommended to keep Kotlin and Java files separate (if it's the case that the same project contains Java files). As with targeting the JVM, if not using the default convention, we need to specify the source folder using *sourceSets*

```
sourceSets {  
    main.kotlin.srcDirs += 'src/main/myKotlin'  
}
```

If you want to create a re-usable library, use `kotlinOptions.metaInfo` to generate additional JS file with binary descriptors. This file should be distributed together with the result of translation.

```
compileKotlin2Js {  
    kotlinOptions.metaInfo = true  
}
```

Targeting Android

Android's Gradle model is a little different from ordinary Gradle, so if we want to build an Android project written in Kotlin, we need *kotlin-android* plugin instead of *kotlin*:

```
buildscript {  
    ...  
}  
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'
```

Android Studio

If using Android Studio, the following needs to be added under `android`:

```
android {  
    ...  
  
    sourceSets {  
        main.java.srcDirs += 'src/main/kotlin'  
    }  
}
```

This lets Android Studio know that the `kotlin` directory is a source root, so when the project model is loaded into the IDE it will be properly recognized. Alternatively, you can put Kotlin classes in the Java source directory, typically located in `src/main/java`.

Configuring Dependencies

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, apply the `kotlin-kapt` plugin:

```

apply plugin: 'kotlin-kapt'

```

Or, starting with Kotlin 1.1.1, you can apply it using the plugins DSL:

```

plugins {
    id "org.jetbrains.kotlin.kapt" version "<version to use>"
}

```

Then 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`. Note that `kaptAndroidTest` and `kaptTest` extends `kapt`, so you can just provide the `kapt` dependency and it will be available both for production sources and tests.

Some annotation processors (such as `AutoFactory`) rely on precise types in declaration signatures. By default, Kapt replaces every unknown type (including types for the generated classes) to `NonExistentClass`, but you can change this behavior. Add the additional flag to the `build.gradle` file to enable error type inferring in stubs:

```

kapt {
    correctErrorTypes = true
}

```

Note that this option is experimental and it is disabled by default.

Incremental compilation

Kotlin supports optional incremental compilation in Gradle. Incremental compilation tracks changes of source files between builds so only files affected by these changes would be compiled.

Starting with Kotlin 1.1.1, incremental compilation is enabled by default.

There are several ways to override the default setting:

1. add `kotlin.incremental=true` or `kotlin.incremental=false` line either to a `gradle.properties` or a `local.properties` file;
2. add `-Pkotlin.incremental=true` or `-Pkotlin.incremental=false` to gradle command line parameters. Note that in this case the parameter should be added to each subsequent build, and any build with disabled incremental compilation invalidates incremental caches.

When incremental compilation is enabled, you should see the following warning message in your build log:

Using kotlin incremental compilation

Note, that the first build won't be incremental.

Coroutines support

[Coroutines](#) support is an experimental feature in Kotlin 1.1, so the Kotlin compiler reports a warning when you use coroutines in your project. To turn off the warning, add the following block to your `build.gradle` file:

```
kotlin {
    experimental {
        coroutines 'enable'
    }
}
```

Compiler Options

To specify additional compilation options, use the `kotlinOptions` property of a Kotlin compilation task.

When targeting the JVM, the tasks are called `compileKotlin` for production code and `compileTestKotlin` for test code. The tasks for custom source sets are called accordingly to the `compile<Name>Kotlin` pattern.

When targeting JavaScript, the tasks are called `compileKotlin2Js` and `compileTestKotlin2Js` respectively, and `compile<Name>Kotlin2Js` for custom source sets.

Examples:

```
compileKotlin {
    kotlinOptions.suppressWarnings = true
}

compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```

A complete list of options for the Gradle tasks follows:

Attributes common for JVM and JS

Name	Description	Possible values	Default value
apiVersion	Allow to use declarations only from the specified version of bundled libraries	"1.0", "1.1"	"1.1"

Name	Description	Possible values	Default value
languageVersion	Provide source compatibility with specified language version	"1.0", "1.1"	"1.1"
suppressWarnings	Generate no warnings		false
verbose	Enable verbose logging output		false
freeCompilerArgs	A list of additional compiler arguments		[]

Attributes specific for JVM

Name	Description	Possible values	Default value
javaParameters	Generate metadata for Java 1.8 reflection on method parameters		false
jdkHome	Path to JDK home directory to include into classpath, if differs from default JAVA_HOME		
jvmTarget	Target version of the generated JVM bytecode (1.6 or 1.8), default is 1.6	"1.6", "1.8"	"1.6"
noJdk	Don't include Java runtime into classpath		false
noReflect	Don't include Kotlin reflection implementation into classpath		true
noStdlib	Don't include Kotlin runtime into classpath		true

Attributes specific for JS

Name	Description	Possible values	Default value
main	Whether a main function should be called	"call", "noCall"	"call"
metaInfo	Generate .meta.js and .kjsm files with metadata. Use to create a library		true
moduleKind	Kind of a module generated by compiler	"plain", "amd", "commonjs", "umd"	"plain"
noStdlib	Don't use bundled Kotlin stdlib		true
outputFile	Output file path		
sourceMap	Generate source map		false
target	Generate JS files for specific ECMA version	"v5"	"v5"

OSGi

For OSGi support see the [Kotlin OSGi page](#).

Examples

The [Kotlin Repository](#) contains examples:

- [Kotlin](#)
- [Mixed Java and Kotlin](#)
- [Android](#)
- [JavaScript](#)

Using Maven

Plugin and Versions

The *kotlin-maven-plugin* compiles Kotlin sources and modules. Currently only Maven v3 is supported.

Define the version of Kotlin you want to use via a *kotlin.version* property:

```
<properties>
  <kotlin.version>1.1.1</kotlin.version>
</properties>
```

Dependencies

Kotlin has an extensive standard library that can be used in your applications. Configure the following dependency in the pom file

```
<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-stdlib</artifactId>
    <version>${kotlin.version}</version>
  </dependency>
</dependencies>
```

If your project uses [Kotlin reflection](#) or testing facilities, you need to add the corresponding dependencies as well. The artifact IDs are `kotlin-reflect` for the reflection library, and `kotlin-test` and `kotlin-test-junit` for the testing libraries.

Compiling Kotlin only source code

To compile source code, specify the source directories in the tag:

```
<build>
  <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
  <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

The Kotlin Maven Plugin needs to be referenced to compile the sources:

```
<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>
```

Compiling Kotlin and Java sources

To compile mixed code applications Kotlin compiler should be invoked before Java compiler. In maven terms that means kotlin-maven-plugin should be run before maven-compiler-plugin using the following method, making sure that the kotlin plugin is above the maven-compiler-plugin in your pom.xml file.

```
<build>
  <plugins>
    <plugin>
      <artifactId>kotlin-maven-plugin</artifactId>
      <groupId>org.jetbrains.kotlin</groupId>
      <version>${kotlin.version}</version>
      <executions>
        <execution>
          <id>compile</id>
          <goals> <goal>compile</goal> </goals>
          <configuration>
            <sourceDirs>
              <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
              <sourceDir>${project.basedir}/src/main/java</sourceDir>
            </sourceDirs>
          </configuration>
        </execution>
        <execution>
          <id>test-compile</id>
          <goals> <goal>test-compile</goal> </goals>
          <configuration>
            <sourceDirs>
              <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>
              <sourceDir>${project.basedir}/src/test/java</sourceDir>
            </sourceDirs>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <executions>
        <!-- Replacing default-compile as it is treated specially by maven -->
        <execution>
          <id>default-compile</id>
          <phase>none</phase>
        </execution>
        <!-- Replacing default-testCompile as it is treated specially by maven -->
        <execution>
          <id>default-testCompile</id>
          <phase>none</phase>
        </execution>
        <execution>
          <id>java-compile</id>
          <phase>compile</phase>
          <goals> <goal>compile</goal> </goals>
        </execution>
        <execution>
          <id>java-test-compile</id>
          <phase>test-compile</phase>
          <goals> <goal>testCompile</goal> </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Jar file

To create a small Jar file containing just the code from your module, include the following under `build->plugins` in your Maven pom.xml file, where `main.class` is defined as a property and points to the main Kotlin or Java class.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
        <mainClass>${main.class}</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

Self-contained Jar file

To create a self-contained Jar file containing the code from your module along with dependencies, include the following under `build->plugins` in your Maven pom.xml file, where `main.class` is defined as a property and points to the main Kotlin or Java class.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals> <goal>single</goal> </goals>
      <configuration>
        <archive>
          <manifest>
            <mainClass>${main.class}</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

This self-contained jar file can be passed directly to a JRE to run your application:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Targeting JavaScript

In order to compile JavaScript code, you need to use the `js` and `test-js` goals for the `compile` execution:

```

<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <phase>compile</phase>
      <goals>
        <goal>js</goal>
      </goals>
    </execution>
    <execution>
      <id>test-compile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>test-js</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

You also need to change the standard library dependency:

```

<groupId>org.jetbrains.kotlin</groupId>
<artifactId>kotlin-stdlib-js</artifactId>
<version>${kotlin.version}</version>

```

For unit testing support, you also need to add a dependency on the `kotlin-test-js` artifact.

See the [Getting Started with Kotlin and JavaScript with Maven](#) tutorial for more information.

Specifying compiler options

Additional options for the compiler can be specified as tags under the `<configuration>` element of the Maven plugin node:

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>
  <executions>...</executions>
  <configuration>
    <nowarn>true</nowarn> <!-- Disable warnings -->
  </configuration>
</plugin>

```

Many of the options can also be configured through properties:

```

<project ...>
  <properties>
    <kotlin.compiler.languageVersion>1.0</kotlin.compiler.languageVersion>
  </properties>
</project>

```

The following attributes are supported:

Attributes common for JVM and JS

Name	Property name	Description	Possible values	Default value
nowarn		Generate no warnings	true, false	false
languageVersion	kotlin.compiler.languageVersion	Provide source compatibility with specified	"1.0",	"1.1"

Name	Property name	Description	Possible values	Default value
apiVersion	kotlin.compiler.apiVersion	language version Allow to use declarations only from the specified version of bundled libraries	"1.0", "1.1"	1.1
sourceDirs		The directories containing the source files to compile		The project source roots
compilerPlugins		Enabled compiler plugins		[]
pluginOptions		Options for compiler plugins		[]
args		Additional compiler arguments		[]

Attributes specific for JVM

Name	Property name	Description	Possible values	Default value
jvmTarget	kotlin.compiler.jvmTarget	Target version of the generated JVM bytecode	"1.6", "1.8"	"1.6"
jdkHome	kotlin.compiler.jdkHome	Path to JDK home directory to include into classpath, if differs from default JAVA_HOME		

Attributes specific for JS

Name	Property name	Description	Possible values	Default value
outputFile		Output file path		
metaInfo		Generate .meta.js and .kjsm files with metadata. Use to create a library	true, false	true
sourceMap		Generate source map	true, false	false
moduleKind		Kind of a module generated by compiler	"plain", "amd", "commonjs", "umd"	"plain"

Generating documentation

The standard JavaDoc generation plugin (`maven-javadoc-plugin`) does not support Kotlin code. To generate documentation for Kotlin projects, use [Dokka](#); please refer to the [Dokka README](#) for configuration instructions. Dokka supports mixed-language projects and can generate output in multiple formats, including standard JavaDoc.

OSGi

For OSGi support see the [Kotlin OSGi page](#).

Examples

An example Maven project can be [downloaded directly from the GitHub repository](#)

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 metadata -->
    <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
libraries	Paths to Kotlin libraries	No
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

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 couldn'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.

Compiler Plugins

All-open compiler plugin

Kotlin has classes and their members `final` by default, which makes it inconvenient to use frameworks and libraries such as Spring AOP that require classes to be `open`. The `all-open` compiler plugin adapts Kotlin to the requirements of those frameworks and makes classes annotated with a specific annotation and their members open without the explicit `open` keyword. For instance, when you use Spring, you don't need all the classes to be open, but only classes annotated with specific annotations like `@Configuration` or `@Service`. The `all-open` plugin allows to specify these annotations.

We provide all-open plugin support both for Gradle and Maven, as well as the IDE integration. For Spring you can use the `kotlin-spring` compiler plugin ([see below](#)).

How to use all-open plugin

Add the plugin in `build.gradle`:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"
```

Or, if you use the Gradle plugins DSL, add it to the `plugins` block:

```
plugins {
    id "org.jetbrains.kotlin.plugin.allopen" version "<version to use>"
}
```

Then specify the annotations that will make the class open:

```
allOpen {
    annotation("com.my.Annotation")
}
```

If the class (or any of its superclasses) is annotated with `com.my.Annotation`, the class itself and all its members will become open.

It also works with meta-annotations:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // will be all-open
```

`MyFrameworkAnnotation` is also the annotation that makes the class open, because it's annotated with `com.my.Annotation`.

Here's how to use all-open with Maven:

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <configuration>
    <compilerPlugins>
      <!-- Or "spring" for the Spring support -->
      <plugin>all-open</plugin>
    </compilerPlugins>

    <pluginOptions>
      <!-- Each annotation is placed on its own line -->
      <option>all-open:annotation=com.my.Annotation</option>
      <option>all-open:annotation=com.their.AnotherAnnotation</option>
    </pluginOptions>
  </configuration>

  <dependencies>
    <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-allopen</artifactId>
      <version>${kotlin.version}</version>
    </dependency>
  </dependencies>
</plugin>

```

Kotlin-spring compiler plugin

You don't need to specify Spring annotations by hand, you can use the `kotlin-spring` plugin, which automatically configures the all-open plugin according to the requirements of Spring.

```

buildscript {
  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
  }
}

apply plugin: "kotlin-spring"

```

Or using the Gradle plugins DSL:

```

plugins {
  id "org.jetbrains.kotlin.plugin.spring" version "<version to use>"
}

```

The Maven example is similar to the one above.

The plugin specifies the following annotations: `@Component`, `@Async`, `@Transactional`, `@Cacheable`. Thanks to meta-annotations support classes annotated with `@Configuration`, `@Controller`, `@RestController`, `@Service` or `@Repository` are automatically opened since these annotations are meta-annotated with `@Component`.

Of course, you can use both `kotlin-allopen` and `kotlin-spring` in the same project. Note that if you use start.spring.io the `kotlin-spring` plugin will be enabled by default.

No-arg compiler plugin

The no-arg compiler plugin generates an additional zero-argument constructor for classes with a specific annotation. The generated constructor is synthetic so it can't be directly called from Java or Kotlin, but it can be called using reflection. This allows the Java Persistence API (JPA) to instantiate the `data` class although it doesn't have the no-arg constructor from Kotlin or Java point of view (see the description of `kotlin-jpa` plugin [below](#)).

How to use no-arg plugin

The usage is pretty similar to all-open. You add the plugin and specify the list of annotations that must lead to generating a no-arg constructor for the annotated classes.

How to use no-arg in Gradle:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-noarg"
```

Or using the Gradle plugins DSL:

```
plugins {
    id "org.jetbrains.kotlin.plugin.noarg" version "<version to use>"
}
```

Then specify the annotation types:

```
noArg {
    annotation("com.my.Annotation")
}
```

How to use no-arg in Maven:

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <configuration>
    <compilerPlugins>
      <!-- Or "jpa" for JPA support -->
      <plugin>no-arg</plugin>
    </compilerPlugins>

    <pluginOptions>
      <option>no-arg:annotation=com.my.Annotation</option>
    </pluginOptions>
  </configuration>

  <dependencies>
    <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-noarg</artifactId>
      <version>${kotlin.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

Kotlin-jpa compiler plugin

The plugin specifies [@Entity](#) and [@Embeddable](#) annotations as markers that no-arg constructor should be generated for a class. That's how you add the plugin in Gradle:

```
buildscript {  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"  
    }  
}  
  
apply plugin: "kotlin-jpa"
```

Or using the Gradle plugins DSL:

```
plugins {  
    id "org.jetbrains.kotlin.plugin.jpa" version "<version to use>"  
}
```

The Maven example is similar to the one above.

FAQ

FAQ

Common Questions

What is Kotlin?

Kotlin is a statically typed language that targets the JVM and JavaScript. It is a general-purpose language intended for industry use.

It is developed by a team at JetBrains although it is an OSS language and has external contributors.

Why a new language?

At JetBrains, we've been developing for the Java platform for a long time, and we know how good it is. On the other hand, we know that the Java programming language has certain limitations and problems that are either impossible or very hard to fix due to backward-compatibility issues. We know that Java is going to stand long, but we believe that the community can benefit from a new statically typed JVM-targeted language free of the legacy trouble and having the features so desperately wanted by the developers.

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 language 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 is an OSS language and is licensed under the Apache 2 OSS License. The IntelliJ Plug-in is also OSS.

It is hosted on GitHub and we happily accept contributors

Where can I get an HD Kotlin logo?

Logos can be downloaded [here](#). Please follow simple rules in the `readme.txt` inside the archive.

Is it Java Compatible?

Yes. The compiler emits Java byte-code. Kotlin can call Java, and Java can call Kotlin. See [Java interoperability](#).

Which minimum Java version is required for running Kotlin code?

Kotlin generates bytecode which is compatible with Java 6 or newer. This ensures that Kotlin can be used in environments such as Android, where Java 6 is the latest supported version.

Is there tooling support?

Yes. There is an IntelliJ IDEA plugin that is available as an OSS project under the Apache 2 License. You can use Kotlin both in the [free OSS Community Edition and Ultimate Edition](#) of IntelliJ IDEA.

Is there Eclipse support?

Yes. Please refer to the [tutorial](#) for installation instructions.

Is there a standalone compiler?

Yes. You can download the standalone compiler and other builds tools from the [release page on GitHub](#)

Is Kotlin a Functional Language?

Kotlin is an Object-Oriented language. However it has support for higher-order functions as well as lambda expressions and top-level functions. In addition, there are a good number of common functional language constructs in the Kotlin Standard Library (such as map, flatMap, reduce, etc.). Also, there's no clear definition on what a Functional Language is so we couldn't say Kotlin is one.

Does Kotlin support generics?

Kotlin supports generics. It also supports declaration-site variance and usage-site variance. Kotlin does not have wildcard types. Inline functions support reified type parameters.

Are semicolons required?

No. They are optional.

Why have type declarations on the right?

We believe it makes the code more readable. Besides, it enables some nice syntactic features. For instance, it is easy to leave type annotations out. Scala has also proven pretty well this is not a problem.

Will right-handed type declarations affect tooling?

No, they won't. We can still implement suggestions for variable names, etc.

Is Kotlin extensible?

We are planning on making it extensible in a few ways: from inline functions to annotations and type loaders.

Can I embed my DSL into the language?

Yes. Kotlin provides a few features that help along: Operator overloading, Custom Control Structures via inline functions, Infix function calls, Extension Functions, Annotations.

What ECMAScript level does Kotlin for JavaScript support?

Currently at 5.

Does the JavaScript back-end support module systems?

Yes. There are plans to provide at least CommonJS and AMD support.

Comparison to Java

Some Java issues addressed in Kotlin

Kotlin fixes a series of issues that Java suffers from

- Null references are [controlled by the type system](#).
- [No raw types](#)
- Arrays in Kotlin are [invariant](#)
- Kotlin has proper [function types](#), as opposed to Java's SAM-conversions
- [Use-site variance](#) without wildcards
- Kotlin does not have checked [exceptions](#)

What Java has that Kotlin does not

- [Checked exceptions](#)
- [Primitive types](#) that are not classes
- [Static members](#)
- [Non-private fields](#)
- [Wildcard-types](#)

What Kotlin has that Java does not

- [Lambda expressions](#) + [Inline functions](#) = performant custom control structures
- [Extension functions](#)
- [Null-safety](#)
- [Smart casts](#)
- [String templates](#)
- [Properties](#)
- [Primary constructors](#)
- [First-class delegation](#)
- [Type inference for variable and property types](#)
- [Singletons](#)
- [Declaration-site variance & Type projections](#)
- [Range expressions](#)
- [Operator overloading](#)
- [Companion objects](#)
- [Data classes](#)
- [Separate interfaces for read-only and mutable collections](#)

Comparison to 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. Taking this into account, if you are happy with Scala, you most likely do not need Kotlin.

What Scala has that Kotlin does not

- Implicit conversions, parameters, etc
 - In Scala, sometimes it's very hard to tell what's happening in your code without using a debugger, because too many implicits get into the picture
 - To enrich your types with functions in Kotlin use [Extension functions](#).
- Overridable type members
- Path-dependent types
- Macros
- Existential types
 - [Type projections](#) are a very special case
- Complicated logic for initialization of traits
 - See [Classes and Inheritance](#)
- Custom symbolic operations
 - See [Operator overloading](#)
- Structural types
- Value types
 - We plan to support [Project Valhalla](#) once it is released as part of the JDK
- Yield operator and actors
 - See [Coroutines](#)
- Parallel collections
 - Kotlin supports Java 8 streams, which provide similar functionality

What Kotlin has that Scala does not

- [Zero-overhead null-safety](#)
 - Scala has Option, which is a syntactic and run-time wrapper
- [Smart casts](#)
- [Kotlin's Inline functions facilitate Nonlocal jumps](#)
- [First-class delegation](#). Also implemented via 3rd party plugin: Autoproxy

