



Kotlin Language Documentation

Table of Contents

Getting Started	4
Basic Syntax	4
Idioms	10
Coding Conventions	14
Basics	15
Basic Types	15
Packages	20
Control Flow	21
Returns and Jumps	24
Classes and Objects	26
Classes and Inheritance	26
Properties and Fields	32
Interfaces	35
Visibility Modifiers	37
Extensions	39
Data Classes	42
Generics	43
Generic functions	47
Generic constraints	47
Nested Classes	48
Enum Classes	49
Object Expressions and Declarations	51
Delegation	54
Delegated Properties	55
Functions and Lambdas	59
Functions	59
Higher-Order Functions and Lambdas	65
Inline Functions	69
Other	72

Multi-Declarations	72
Ranges	74
Type Checks and Casts	79
This Expression	81
Equality	82
Operator overloading	83
Null Safety	86
Exceptions	89
Annotations	91
Reflection	94
Dynamic Type	102
Reference	103
Interop	106
Java Interop	106
Tools	116
Documenting Kotlin Code	116
Using Maven	119
Using Ant	123
Using Griffon	126
Using Gradle	127
FAQ	131
FAQ	131
Comparison to Java	133
Comparison to Scala	134

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 visible from outside of a module should have return type explicitly specified:

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

Function returning no meaningful value:

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

`Unit` return type can be omitted:

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

See [Functions](#).

Defining local variables

Assign-once (read-only) local variable:

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

Mutable variable:

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

See also [Properties And Fields](#).

Using string templates

```
fun main(args: Array<String>) {  
    if (args.size() == 0) return  
  
    print("First argument: ${args[0]}")  
}
```

See [String templates](#).

Using conditional expressions

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

Using `if` as an expression:

```
fun max(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 main(args: Array<String>) {  
    if (args.size() < 2) {  
        print("Two integers expected")  
        return  
    }  
  
    val x = parseInt(args[0])  
    val y = parseInt(args[1])  
  
    // 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  
        print(x * y)  
    }  
}
```

or

```
// ...  
if (x == null) {  
    print("Wrong number format in '${args[0]}'")  
    return  
}  
if (y == null) {  
    print("Wrong number format in '${args[1]}'")  
    return  
}  
  
// x and y are automatically cast to non-nullable after null check  
print(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

```

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

```

or

```

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

```

See [for loop](#).

Using a while loop

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

See [while loop](#).

Using when expression

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

See [when expression](#).

Using ranges

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

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

Check if a number is out of range:

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

Iterating over a range:

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

See [Ranges](#).

Using collections

Iterating over a collection:

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

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


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

Using function literals to filter and map collections:

```
names filter { it.startsWith("A") } sortBy { it } map { it.toUpperCase() } forEach {
    print(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 DTO's (POJO's/POCO's)

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

provides a `Customer` class with the following functionality:

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

Declaring a final local variable

```
val a = foo()
```

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) { ... }
for (x in 2..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 Delegates.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")
    }
}
```

Return on try catch block

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

    // Working with result
}
```

Return on if statement

```

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

```

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")
}

```

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

Kotlin does not have fields as a primary concept in the language – it only has properties. Avoid the use of prefixes on properties, such as `_` or `m_` or other kinds of notation. If you need access to a backing field of a property, use the `$` prefix: `$foo` to refer to a field behind property `foo`; never create a private property and call it `_foo`

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  
}
```

Unit

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

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

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	Bitwidth
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`

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 we can use literals almost freely cause the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example

```
val l = 1.toLong() + 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'`, `'\n'`, `'\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

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`, along with a few other useful member functions:

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

    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 = intArray(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.

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)
    """
```

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"
```

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.

Imports

Apart from the default imports declared by the module, 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'
```

Visibility and Package Nesting

If a top-level declaration is marked `private`, it is private to the package it's declared in (see [Visibility Modifiers](#)). Since packages really nest in Kotlin, i.e. package `foo.bar` is considered a member of `foo`, if something is `private` in a package, it is visible to all its subpackages.

Note that members of outer packages are **not** imported by default, i.e. in a file in package `foo.bar` we can't access members of `foo` without importing them.

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
}
```

When `if` has only one branch, or one of its branches results in `Unit`, its type is `Unit`.

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

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.

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 operators

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

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 function literal. 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 function literals passed to [inline-functions](#).) If we need to return from a function literal, 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 function literal. 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 function literal with a [function expression](#). A `return` statement in a function expression will return from the function expression 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)`”.

Named functions automatically define labels:

```
fun outer() {  
    fun inner() {  
        return@outer // the label @outer was defined automatically  
    }  
}
```

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`:

```
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.

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 Members

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() {}
}

```

Wait! How will I hack my libraries now?!

One issue with our approach to overriding (classes and members final by default) is that it would be difficult to subclass something inside the libraries you use to override some method that was not intended for overriding by the library designer, and introduce some nasty hack there.

We think that this is not a disadvantage, for the following reasons:

- Best practices say that you should not allow these hacks anyway
- People successfully use other languages (C++, C#) that have similar approach
- If people really want to hack, there still are ways: you can always write your hack in Java and call it from Kotlin (see [Java Interop](#)), and Aspect frameworks always work for these purposes

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. Thus, when some descendant inherits an abstract member, it does not count as an implementation:

```
abstract class A {
    abstract fun f()
}

interface B {
    open fun f() { print("B") }
}

class C() : A(), B {
    // We are not required to override f()
}
```

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.

```
public class Address {  
    public var name: String = ...  
    public var street: String = ...  
    public var city: String = ...  
    public var state: String? = ...  
    public 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 base class member being overridden.

Examples:

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

Note that types are not inferred for properties exposed as parts of the public API, i.e. public and protected, because changing the initializer may cause an unintentional change in the public API then. For example:

```
public val example = 1 // error: a public property must have a type specified  
    explicitly
```

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.

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" // Initializer required, not a nullable type
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any?
    @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 `$` symbol followed by the property name.

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

The `$counter` field can be accessed only from inside the class where the counter property is defined.

The compiler looks at the accessors' bodies, and if they use the backing field (or the accessor implementation is left by default), a backing field is generated, otherwise it is not.

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.

Overriding Properties

See [Overriding Members](#)

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*. For more information, look [here](#).

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.

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 {  
    fun bar() {  
        // body  
    }  
}
```

Properties in Interfaces

Interfaces allow properties as long as these are stateless, that is because interfaces do not allow state.

```
interface MyInterface {  
    val property: Int // abstract  
  
    fun foo() {  
        print(property)  
    }  
}  
  
class Child : MyInterface {  
    override val property: 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()
    }
}

```

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. And if we derive *D* from *A* and *B*, we don't have to override *bar()*, because we have inherited only one implementation of it. But we have inherited two implementations of *foo()*, so the compiler does not know which one to choose, and forces us to override *foo()* and say what we want explicitly.

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` — visible only in the declaring scope and its subscopes (inside the same module);
- `protected` — (applicable only to class/interface members) like `private`, but also visible in subclasses;
- `internal` — (used by default) visible everywhere within the same module (if the owner of declaring scope is visible);
- `public` — visible everywhere (if the owner of declaring scope is visible).

NOTE: Functions *with expression bodies* and all properties declared `public` must always specify return types explicitly. This is required so that we do not accidentally change a type that is a part of a public API by merely altering the implementation.

```
public val foo: Int = 5    // explicit return type required
public fun bar(): Int = 5  // explicit return type required
public fun bar() {}       // block body: return type is Unit and can't be changed
                           accidentally, so not required
```

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, `internal` is used by default, which means that your declarations will be visible everywhere within the same module;
- If you mark a declaration `private`, it will only be visible inside this package and its subpackages, and only within the same module;
- If you mark it `public`, it is visible everywhere;
- `protected` is not available for top-level declarations.

Examples:

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

private fun foo() {} // visible inside this package and subpackaged

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in this package and subpackages

internal val baz = 6    // visible inside the same module, the modifier can be omitted
```

Classes and Interfaces

When 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.

Examples:

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

    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
}

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. Unlike other declarations, 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.

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*. Currently, 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(x: Int, y: Int) {
    val tmp = this[x] // 'this' corresponds to the list
    this[x] = this[y]
    this[y] = 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(x: Int, y: Int) {
    val tmp = this[x] // 'this' corresponds to the list
    this[x] = this[y]
    this[y] = 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 instances of this class.

We would like to emphasize that extension functions are dispatched **statically**, i.e. they are not virtual by receiver type. If there's a member and extension of the same type both applicable to given arguments, a **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".

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.

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 classes that do nothing but hold data. In such classes some functionality is often mechanically derivable from the data they hold. In Kotlin a class can be annotated as `data` :

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

This is called a *data class*. 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.

NOTE that if a constructor parameter does not have a `val` or `var` in front of it, it will not be included in computation of all these functions; nor will be properties declared in the class body or inherited from the superclass.

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 Multi-Declarations

Component functions generated for data classes enable their use in [multi-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.

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, cause the following code would have compiled and cause 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> {
    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> {
    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 length: 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.length == to.length)
    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> = array(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 say that we are dealing with an *out*-projection (the object does not consume any values of unknown types), and that this projection is with the upper bound of the corresponding parameter, i.e. `out Any?` for most cases. Kotlin provides a shorthand syntax for this, that we call a **star-projection**: `Foo<*>` means `Foo<out Bar>` where `Bar` is the upper bound for `Foo`'s type parameter.

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. Usually, we place the type parameters in angle brackets **after** the name of the function:

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

But for [Extension functions](#) it may be necessary to declare type parameters before specifying the receiver type, so Kotlin allows the alternative syntax:

```
fun <T> T.basicToString() : String {  
    return typeinfo.typeinfo(this) + "@" + System.identityHashCode(this)  
}
```

If type parameters are passed explicitly at the call site, they can be only specified **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 sort<T : Comparable<T>>(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(list(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>  
sort(list(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 cloneWhenGreater<T : Comparable<T>>(list: List<T>, threshold: T): List<T>  
    where T : Cloneable {  
    return list when {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.

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.

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

```
name(): String  
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 = 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:

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

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*. If there's a name following the `object` keyword, we are not talking about an *expression* anymore. We cannot assign such a thing to a variable, but we can refer to it by its name. 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 in 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 `@platformStatic` annotation. See the [Java interoperability](#) section for more details.

Semantical difference between object expressions and declarations

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

- object declarations are initialized **lazily**, when accessed for the first time
- object expressions are executed (and initialized) **immediately**, where they are used

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() {  
    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 it.

Property delegates don't have to implement any interface, but they have to provide a `get()` function (and `set()` — for `var`'s). For example:

```
class Delegate {  
    fun get(thisRef: Any?, prop: PropertyMetadata): String {  
        return "$thisRef, thank you for delegating '${prop.name}' to me!"  
    }  
  
    fun set(thisRef: Any?, prop: PropertyMetadata, value: String) {  
        println("$value has been assigned to '${prop.name}' in $thisRef.")  
    }  
}
```

When we read from `p` that delegates to an instance of `Delegate`, the `get()` 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 `set()` 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.

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 `get` that takes the following parameters:

- receiver — must be the same or a supertype of the *property owner* (for extension properties — the type being extended),
- metadata — must be of type `PropertyMetadata` 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 `set` that takes the following parameters:

- receiver — same as for `get()`,
- metadata — same as for `get()`,
- new value — must be of the same type as a property or its subtype.

Standard Delegates

The `kotlin.properties.Delegates` object from the standard library provides factory methods for several useful kinds of delegates.

Lazy

`Delegates.lazy()` is a function that takes a lambda and returns a delegate that implements 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.

```
import kotlin.properties.Delegates

val lazy: String by Delegates.lazy {
    println("computed!")
    "Hello"
}

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

If you want **thread safety**, use `blockingLazy()`: it works the same way, but guarantees that the values will be computed only in one thread, and that all threads will see the same value.

Observable

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


```

class User {
    var name: String by Delegates.observable("<no name>") {
        d, 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()`.

Not-Null

Sometimes we have a non-null `var`, but we don’t have an appropriate value to assign to it in the constructor, i.e. it must be assigned later. The problem is that you can’t have an uninitialized non-abstract property in Kotlin:

```

class Foo {
    var bar: Bar // ERROR: must be initialized
}

```

We could initialize it with `null`, but then we’d have to check every time we access it.

`Delegates.notNull()` can solve this problem:

```

class Foo {
    var bar: Bar by Delegates.notNull()
}

```

If this property is read before being written to for the first time, it throws an exception, after the first assignment it works as expected.

Storing Properties in a Map

`Delegates.mapVal()` takes a map instance and returns a delegate that reads property values from this map, using property name as a key. There are many use cases of this kind in applications like parsing JSON or doing other “dynamic” things:

```

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

```

In this example, the constructor takes a map:

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

Delegates take values from this map (by the string keys – names of properties):

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

For `var`'s we can use `mapVar()` (note that it takes a `MutableMap` instead of read-only `Map`).

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

```
// Define extension to Int  
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.

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 = '_')
```

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

There are cases when an explicit return type is required:

- Functions with expression body that are public or protected. These are considered to be part of the public API surface. Not having explicit return types makes it potentially easier to change the type accidentally. This is the same reason why explicit types are required for [properties](#).
- 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)

The last parameter of a function may be marked with `vararg` annotation

```
fun asList<T>(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 annotated as `vararg`. It may be the last parameter or the one before last, if the last parameter has a function type (allowing a lambda to be passed 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 = array(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])
}

```

Local functions can even return from outer functions using [qualified return expressions](#)

```

fun reachable(from: Vertex, to: Vertex): Boolean {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        // here we return from the outer function:
        if (current == to) return@reachable true
        // And here -- from local function:
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(from)
    return false // if dfs() did not return true already
}

```

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 after the function name and before the value parameters

```

fun singletonArray<T>(item: T): Array<T> {
    return Array<T>(1, {item})
}

```

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 `tailRecursive` annotation and meets the required form the compiler optimises out the recursion, leaving behind a fast and efficient loop based version instead.

```
tailRecursive 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 `tailRecursive` annotation 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 lock<T>(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 [function literal](#) (often referred to as *lambda expression*):

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

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

- A function literal 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, then we can omit the parentheses

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

Another example of a higher-order function would be `map()` (of [MapReduce](#)):

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

This function can be called as follows:

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

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} // Infix call + Implicit 'it'
```

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

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

Inline Functions

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

Function Literals and Function Expressions

A function literal or a function expression is an “anonymous function”, 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 max<T>(collection: Collection<out 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, for documentation purposes and to enable calls with [named arguments](#).

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

Function Literal Syntax

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

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

A function literal 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 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 function literal 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'
```

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

Function Expressions

One thing missing from the function literal 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: a *function expression*.

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

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

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

One other difference between function literals and function expressions 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 function literal will return from the enclosing function, whereas a `return` inside a function expression will return from the function expression itself.

Closures

A function literal or expression (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)
```

Extension Function Expressions

In addition to ordinary functions, Kotlin supports extension functions. This kind of functions is so useful that extension function literals and expressions are also supported. One of the most important examples of their usage is [Type-safe Groovy-style builders](#).

An extension function expression differs from an ordinary one in that it has a receiver type specification.

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

Receiver type may be specified explicitly only in function expressions, not in function literals. Function literals can be used as extension function expressions, but only when the receiver type can be inferred from the context.

The type of an extension function expression is a function type with receiver:

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

The function can be called with a dot or in infix form (since it has only one parameter):

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

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 function literals. The functions shown above 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

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

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

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

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

The `inline` annotation 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 `@noinline` annotation:

```
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 a function expression. 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 annotated with the

`InlineOptions.OONLY_LOCAL_RETURN` annotation:

```
inline fun f(inlineOptions(InlineOption.OONLY_LOCAL_RETURN) 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(javaClass<MyTreeNodeType>())
```

What we actually want is simply pass a type to this function, i.e. call is 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: `javaClass()` gives us access to it:

```
inline fun methodsOf<reified T>() = javaClass<T>().getMethods()

fun main(s: Array<String>) {
    println(methodsOf<String>().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](#).

Other

Multi-Declarations

Sometimes it is convenient to *decompose* an object into a number of variables, for example:

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

This syntax is called a *multi-declaration*. A multi-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 multi-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 multi-assignment, 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.

Multi-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, multi-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: Multi-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 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:

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

So you can freely use multi-declarations in `for`-loops with maps (as well as collections of data class instances etc).

Ranges

Range expressions are formed with `rangeTo` functions that have the operator form of `..` which are complemented by `in` and `!in`. Range is defined for any comparable type, but for number primitives it is optimized. Here are examples of using ranges

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

if (x !in 1.0..3.0) println(x)

if (str in "island".. "isle") println(str)
```

Numerical ranges have an extra feature: they can be iterated over. Compiler takes care about converting this in simple analogue of Java's indexed `for`-loop, without extra overhead. Examples

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

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

for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "
```

What if you want to iterate over numbers in reversed order? It's simple. You can use `downTo()` function defined in 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, `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"

for (i in 1.0..2.0 step 0.3) print("$i ") // prints "1.0 1.3 1.6 1.9 "
```

How it works

There are two interfaces in the library: `Range<T>` and `Progression<N>`.

`Range<T>` denotes an interval in the mathematical sense, defined for comparable types. It has two endpoints: `start` and `end`, which are included in the range. The main operation is `contains`, usually used in the form of `in`/`!in` operators.

`Progression<N>` denotes an arithmetic progression, defined for number types. It has `start`, `end` and a non-zero `increment`. `Progression<N>` is a subtype of `Iterable<N>`, so it can be used in `for`-loops and functions like `map`, `filter`, etc. First element is `start`, every next element equals previous plus `increment`. Iteration over `Progression` is equivalent to an indexed `for`-loop in Java/JavaScript:

```
// if increment > 0
for (int i = start; i <= end; i += increment) {
    // ...
}
```

```
// if increment < 0
for (int i = start; i >= end; i += increment) {
    // ...
}
```

For numbers, the `..` operator creates an object which is both `Range` and `Progression`. Result of `downTo()` and `step()` functions is always a `Progression`.

Range Specifications

Use Cases

```
// Checking if value of comparable is in range. Optimized for number primitives.
if (i in 1..10) println(i)

if (x in 1.0..3.0) println(x)

if (str in "island".. "isle") println(str)

// Iterating over arithmetical progression of numbers. Optimized for number primitives
// (as indexed for-loop in Java).
for (i in 1..4) print(i) // prints "1234"

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

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

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

for (i in (1..4).reversed()) print(i) // prints "4321"

for (i in (1..4).reversed() step 2) print(i) // prints "42"

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

for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "

for (x in 1.0..2.0 step 0.3) print("$x ") // prints "1.0 1.3 1.6 1.9 "

for (x in 2.0 downTo 1.0 step 0.3) print("$x ") // prints "2.0 1.7 1.4 1.1 "

for (str in "island".. "isle") println(str) // error: string range cannot be iterated
over
```

Common Interfaces Definition

There are two base interfaces: `Range` and `Progression`.

`Range` interface defines a range, or an interval in a mathematical sense. It has two endpoints, `start` and `end`, and also `contains()` function which checks if the range contains a given number (it also can be used as `in`/`!in` operator, which is neater). `start` and `end` are included in the range. If `start == end`, the range contains exactly one element. If `start > end`, the range is empty.

```
interface Range<T : Comparable<T>> {
    val start: T
    val end: T
    fun contains(element: T): Boolean
}
```

`Progression` defines a kind of arithmetical progression. It has `start` (the first element of progression), `end` (the last element which can be included) and `increment` (difference between each progression element and previous, non-zero). But the main feature of it is that the progression can be iterated over, so it is a subtype of `Iterable`. `end` is not necessary the last element of progression. Also, progression can be empty if `start < end && increment < 0` or `start > end && increment > 0`.

```
interface Progression<N : Number> : Iterable<N> {
    val start: N
    val end: N
    val increment: Number // not N, because for Char we'll want it to be negative
    sometimes
    // fun iterator(): Iterator<N> is defined in Iterable interface
}
```

Iteration over `Progression` is equivalent to an indexed `for`-loop in Java:

```
// if increment > 0
for (int i = start; i <= end; i += increment) {
    // ...
}

// if increment < 0
for (int i = start; i >= end; i += increment) {
    // ...
}
```

Implementation Classes

To avoid unnecessary repetition, let's consider only one number type, `Int`. For other number types implementation is the same. Note that instances can be created using constructors of these classes, while it's more handy to use `rangeTo()` (by this name, or as `..operator`), `downTo()`, `reversed()` and `step()` utility functions, which are introduced later.

`IntProgression` class is pretty straightforward and simple:

```
class IntProgression(override val start: Int, override val end: Int, override val
increment: Int): Progression<Int> {
    override fun iterator(): Iterator<Int> = IntProgressionIteratorImpl(start, end,
increment) // implementation of iterator is obvious
}
```

`IntRange` is a bit tricky: it implements `Progression<Int>` along with `Range<Int>`, because it's natural to iterate over a range (default increment value is 1 for both integer and floating-point types):

```

class IntRange(override val start: Int, override val end: Int): Range<Int>,
Progression<Int> {
    override val increment: Int
    get() = 1
    override fun contains(element: Int): Boolean = start <= element && element <= end
    override fun iterator(): Iterator<Int> = IntProgressionIteratorImpl(start, end,
increment)
}

```

`ComparableRange` is also simple (remember that comparisons are translated into invocation of `compareTo()`):

```

class ComparableRange<T : Comparable<T>>(override val start: T, override val end: T):
Range<T> {
    override fun contains(element: T): Boolean = start <= element && element <= end
}

```

Utility functions

`rangeTo()`

Set of `rangeTo()` functions in number types simply call constructors of `*Range` classes, e.g.:

```

class Int {
    //...
    fun rangeTo(other: Byte): IntRange = IntRange(this, other)
    //...
    fun rangeTo(other: Int): IntRange = IntRange(this, other)
    //...
}

```

`downTo()`

`downTo()` extension function is defined for any pair of number types, here are two examples:

```

fun Long.downTo(other: Double): DoubleProgression {
    return DoubleProgression(this, other, -1.0)
}

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

```

`reversed()`

Set of `reversed()` extension functions are defined for each `*Range` and `*Progression` classes, and all of them return reversed progressions.

```

fun IntProgression.reversed(): IntProgression {
    return IntProgression(end, start, -increment)
}

fun IntRange.reversed(): IntProgression {
    return IntProgression(end, start, -1)
}

```

step()

`step()` extension functions are defined for each `*Range` and `*Progression` classes, all of them return progressions with modified `step` value (function parameter). Note that the step value is 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(start, end, if (increment > 0) step else -step)
}

fun IntRange.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression(start, end, step)
}

```

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 Array<Int> -> print(x.sum())
}
```

“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 an [extension function literal](#), `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 [extension function literal](#)) 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 {String.() ->
        val d = this // funLit's receiver
        val d1 = this@lambda // funLit's receiver
      }

      val funLit2 = { (s: String) ->
        // foo()'s receiver, since enclosing function literal
        // 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.

Alternatively, you can check for reference equality using a built-in inline function `identityEquals()` that can be called in the following way

```
a.identityEquals(b)
// or
a identityEquals b // infix call
```

This also returns 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.

Conventions

Here we describe the conventions that regulate operator overloading for different operators.

Unary operations

Expression	Translated to
<code>+a</code>	<code>a.plus()</code>
<code>-a</code>	<code>a.minus()</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 `plus()` with 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

These operations are supposed to change their receiver and (optionally) return a value.

`inc()`/`dec()` shouldn't mutate the receiver object.

By “changing the receiver” we mean *the receiver-variable*, not the receiver object.

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 no parameters, applicable to the receiver of type `T`.
- If the function returns a type `R`, then it must be 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>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.mod(b)</code>
<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.

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.

Symbol	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.

Symbol	Translated to
<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.identityEquals(null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: b.identityEquals(null))</code>

Note: `===` and `!==` (identity checks) are not overloadable, so no conventions exist for them

The `==` operation is special in two ways:

- It is translated to a complex expression that screens for `null`'s, and `null == null` is `true`.
- It looks up a function with a specific *signature*, not just a specific *name*. The function must be declared as

```
fun equals(other: Any?): Boolean
```

Or an extension function with the same parameter list and return type.

Symbol	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 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 references, resulting in null reference exceptions. 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()`
- 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 on `a`, it's guaranteed not to cause an NPE, so you can safely say

```
val l = a.length()
```

But if you want to call the same method 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 call that method, 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.

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.

By the way, `!!` is added for conciseness, and formerly was emulated by an extension function from the standard library, defined as follows:

```
inline fun <T : Any> T?.sure(): T =  
    if (this == null)  
        throw NullPointerException()  
    else  
        this
```

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
```


Exceptions

Exception Classes

All exception classes in Kotlin are descendants of the class `Exception`. 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)

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` keyword in front of a class:

```
annotation class fancy
```

Usage

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

In most cases, the `@` sign is optional. It is only required when annotating expressions or local declarations:

```
fancy class Foo {  
    fancy fun baz(fancy foo: Int): Int {  
        @fancy fun bar() { ... }  
        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 {}
```

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) }
```

Java Annotations

Java annotations are 100% compatible with Kotlin:

```
import org.junit.Test
import org.junit.Assert.*

class Tests {
    Test fun simple() {
        assertEquals(42, getTheAnswer())
    }
}
```

Java annotations can also be made to look like modifiers by renaming them on import:

```
import org.junit.Test as test

class Tests {
    test fun simple() {
        ...
    }
}
```

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
```

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
```

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`. You can use `KClass.properties` and `KClass.extensionProperties` to get the list of [property references](#) for all properties defined in this class and its superclasses.

Note that a Kotlin class reference is not the same as a Java class reference. See the [Java interop section](#) for information on obtaining a Java class reference corresponding to a Kotlin class.

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`.

Note that right now the `::` operator cannot be used for overloaded functions. In the future, we plan to provide a syntax for specifying parameter types so that a specific overload of a function could be selected.

If we need to use a member of a class, or an extension function, it needs to be qualified, and the result will be of type “extension function”, e.g. `String::toCharArray` gives us an extension function for type `String: String.() -> CharArray`.

Example: Function Composition

Consider the following function:

```
fun compose<A, B, C>(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.size

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.

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[size - 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"
}
```

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)
```


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 that is taken from [here](#) and slightly adapted:

```
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 a 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 {
  // ...
}
```

This is actually a function call that takes a [function literal](#) as an argument (see [this page](#) for details). Actually, 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. Actually, it is an [extension function](#) that has a receiver of type `HTML` (and returns nothing interesting, i.e. `Unit`). So, when we pass a function literal as an argument to `html`, it is typed as an extension function literal, and there's a `this` reference available:

```
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) {
  val head = Head()
  head.init()
  children.add(head)
  return head
}

fun body(init: Body.() -> Unit) {
  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 initTag<T : Element>(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 `plus()` operation. That operation is actually defined by an extension function `plus()` that is a member of the `TagWithText` abstract class (a parent of `Title`):

```
fun String.plus() {
    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 next section you can read through the full definition of this package.

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 [Extension function literals](#).

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}
```

```

class TextElement(val text: String): Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

abstract class Tag(val name: String): Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun initTag<T: Element>(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.keySet()) {
            builder.append(" $a=\"${attributes[a]}\"")
        }
        return builder.toString()
    }
}

abstract class TagWithText(name: String): Tag(name) {
    fun String.plus() {
        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
}

```

Appendix. Making Java classes nicer

In the code above there's something that looks very nice:

```

class A() : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) { attributes["href"] = value }
}

```

We access the `attributes` map as if it were an *associative array*: just with the `[]` operation. By [convention](#) this compiles to a call to `get(K)` or `set(K, V)`, all right. But we said that `attributes` was a Java `Map`, i.e. it does NOT have a `set(K, V)`. This problem is easily fixable in Kotlin:

```

fun <K, V> Map<K, V>.set(key: K, value: V) = this.put(key, value)

```

So, we simply define an [extension function](#) `set(K, V)` that delegates to vanilla `put` and makes a Kotlin operator available for a Java class.

Dynamic Type

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(*array(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.

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  
}
```

For a more technical description, see the [spec document](#).

Reference

Grammar

We are working on revamping the Grammar definitions and give it some style! Until then, please check the [Grammar from the old site](#)

Interop

Java Interop

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.

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
}
```

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.get(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”

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
byte	kotlin.Byte
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
<code>java.lang.Object</code>	<code>kotlin.Any!</code>
<code>java.lang.Cloneable</code>	<code>kotlin.Cloneable!</code>
<code>java.lang.Comparable</code>	<code>kotlin.Comparable!</code>
<code>java.lang.Enum</code>	<code>kotlin.Enum!</code>
<code>java.lang.Annotation</code>	<code>kotlin.Annotation!</code>
<code>java.lang.Deprecated</code>	<code>kotlin.deprecated!</code>
<code>java.lang.Void</code>	<code>kotlin.Nothing!</code>
<code>java.lang.CharSequence</code>	<code>kotlin.CharSequence!</code>
<code>java.lang.String</code>	<code>kotlin.String!</code>
<code>java.lang.Number</code>	<code>kotlin.Number!</code>
<code>java.lang.Throwable</code>	<code>kotlin.Throwable!</code>

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`):

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
<code>Iterator<T></code>	<code>Iterator<T></code>	<code>MutableIterator<T></code>	<code>(Mutable)Iterator<T>!</code>
<code>Iterable<T></code>	<code>Iterable<T></code>	<code>MutableIterable<T></code>	<code>(Mutable)Iterable<T>!</code>
<code>Collection<T></code>	<code>Collection<T></code>	<code>MutableCollection<T></code>	<code>(Mutable)Collection<T>!</code>
<code>Set<T></code>	<code>Set<T></code>	<code>MutableSet<T></code>	<code>(Mutable)Set<T>!</code>
<code>List<T></code>	<code>List<T></code>	<code>MutableList<T></code>	<code>(Mutable)List<T>!</code>
<code>ListIterator<T></code>	<code>ListIterator<T></code>	<code>MutableListIterator<T></code>	<code>(Mutable)ListIterator<T>!</code>
<code>Map<K, V></code>	<code>Map<K, V></code>	<code>MutableMap<K, V></code>	<code>(Mutable)Map<K, V>!</code>
<code>Map.Entry<K, V></code>	<code>Map.Entry<K, V></code>	<code>MutableMap.MutableEntry<K, V></code>	<code>(Mutable)Map. (Mutable)Entry<K, V>!</code>

Java's arrays are mapped as mentioned [below](#):

Java type	Kotlin type
<code>int[]</code>	<code>kotlin.IntArray!</code>
<code>String[]</code>	<code>kotlin.Array<(out) String>!</code>

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 (though [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 = intArray(0, 1, 2, 3)
javaObj.removeIndices(array) // passes int[] to method
```

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 = intArray(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

It's currently not possible to pass `null` to a method that is declared as varargs.

When compiling to JVM byte codes, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = array(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])
}
```

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 type information from an object, we use the `javaClass` extension property.

```
val fooClass = foo.javaClass
```

Instead of Java's `Foo.class` use `javaClass()`.

```
val fooClass = javaClass<Foo>()
```

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. This class must go first in the supertype list.

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.javaClass` or `javaClass<ClassName>()` to enter Java reflection through `java.lang.Class`. You can then “convert” to Kotlin reflection by calling `.kotlin`:

```
val kClass = x.javaClass.kotlin
```

In much the same way you can convert from Kotlin reflection to Java: `ClassName::class.java` is the same as `javaClass<ClassName>()`. Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, getting a containing `KPackage` instance for a Java class, and getting a `KProperty` for a Java field.

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") }
```

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.

Calling Kotlin code from Java

Kotlin code can be called from Java easily.

Package-Level Functions

All the functions and properties declared inside a package `org.foo.bar` are put into a Java class named `org.foo.bar.BarPackage`.

```
package demo  
  
class Foo  
  
fun bar() {  
}
```

```
// Java  
new demo.Foo();  
demo.DemoPackage.bar();
```

For the root package (the one that's called a "default package" in Java), a class named `_DefaultPackage` is created.

Static Methods and Fields

As mentioned above, Kotlin generates static methods for package-level functions. On top of that, it also generates static methods for functions defined in named objects or companion objects of classes and annotated as `@platformStatic`. For example:

```
class C {  
    companion object {  
        platformStatic 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
```

Same for named objects:

```
object Obj {
    platformStatic 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
```

Also, public properties defined in objects and companion objects are turned into static fields in Java:

```
object Obj {
    val CONST = 1
}
```

In Java:

```
int c = Obj.CONST;
```

Handling signature clashes with @platformName

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 `@platformName` and specify a different name as an argument:

```
fun List<String>.filterValid(): List<String>
@platformName("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
    @platformName("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:

```

package demo

fun foo() {
    throw IOException()
}

```

And we want to call it from Java and catch the exception:

```

// Java
try {
    demo.DemoPackage.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.

Properties

Property getters are turned into *get*-methods, and setters – into *set*-methods.

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.

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. 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.

`@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.

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 *kotlin.version*. The possible values are:

- X.Y-SNAPSHOT: Correspond to snapshot version for X.Y releases, updated with every successful build on the CI server. These versions are not really stable and are only recommended for testing new compiler features. Currently all builds are published as 0.1-SNAPSHOT. To use a snapshot, you need to [configure a snapshot repository in the pom file](#).
- X.Y.Z: Correspond to release or milestone versions X.Y.Z, updated manually. These are stable builds. Release versions are published to Maven Central Repository. No extra configuration is needed in your pom file.

The correspondence between milestones and versions is displayed below:

Milestone	Version
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

Configuring Snapshot Repositories

To use a snapshot version of Kotlin, include the following repository definitions to the pom

```

<repositories>
  <repository>
    <id>sonatype.oss.snapshots</id>
    <name>Sonatype OSS Snapshot Repository</name>
    <url>http://oss.sonatype.org/content/repositories/snapshots</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>sonatype.oss.snapshots</id>
    <name>Sonatype OSS Snapshot Repository</name>
    <url>http://oss.sonatype.org/content/repositories/snapshots</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>

```

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>

```

Compiling Kotlin only source code

To compile source code, specify the source directories in the tag:

```

<sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
<testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>

```

The Kotlin Maven Plugin needs to be referenced to compile the sources:


```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <executions>
    <execution>
      <id>compile</id>
      <phase>compile</phase>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <phase>test-compile</phase>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>

```

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.

It could be done by moving Kotlin compilation to previous phase, process-sources (feel free to suggest a better solution if you have one):

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>0.1-SNAPSHOT</version>

  <executions>
    <execution>
      <id>compile</id>
      <phase>process-sources</phase>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <phase>process-test-sources</phase>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>

```

Using External Annotations

Kotlin uses external annotation to have precise information about types in Java libraries. To specify these annotations, use annotationPaths in :

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>0.1-SNAPSHOT</version>

  <configuration>
    <annotationPaths>
      <annotationPath>path to annotations root</annotationPath>
    </annotationPaths>
  </configuration>

  ...
```

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>

```

Targeting JavaScript with single source folder

```

<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
  classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <kotlin2js src="root1" output="out.js"/>
  </target>
</project>

```

Targeting JavaScript with Prefix, PostFix and sourcemap options

```

<project name="Ant Task Test" default="build">
  <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml"
  classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <kotlin2js src="root1" output="out.js" outputPrefix="prefix"
  outputPostfix="postfix" sourcemap="true"/>
  </target>
</project>

```

Targeting JavaScript with single source folder and metaInfo option

The `metaInfo` option is useful, if you want to distribute the result of translation as a Kotlin/JavaScript library. If `metaInfo` was set to `true`, then during compilation additional JS file with binary metadata will be created. This file should be distributed together with the result of translation.

```

<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
  classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <!-- out.meta.js will be created, which contains binary descriptors -->
    <kotlin2js src="root1" output="out.js" metaInfo="true"/>
  </target>
</project>

```

References

Complete list of elements and attributes are listed below

kotlinc Attributes

Name	Description	Required	Default Value
src	Kotlin source file or directory to compile	Yes	
output	Destination directory or .jar file name	Yes	
classpath	Compilation class path	No	
classpathref	Compilation class path reference	No	
stdlib	Path to "kotlin-runtime.jar"	No	""
includeRuntimeJar	If output is a .jar file, whether Kotlin runtime library is included in the jar	No	true

withKotlin attributes

Name	Description	Required	Default Value
externalAnnotations	Path to external annotations	No	

kotlin2js Attributes

Name	Description	Required
src	Kotlin source file or directory to compile	Yes
output	Destination file	Yes
library	Library files (kt, dir, jar)	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

Using Griffon

Griffon support is [provided externally](#)

Using Gradle

Plugin and Versions

The *kotlin-gradle-plugin* compiles Kotlin sources and modules.

Define the version of Kotlin we want to use via *kotlin.version*. The possible values are:

- X.Y-SNAPSHOT: Correspond to snapshot version for X.Y releases, updated with every successful build on the CI server. These versions are not really stable and are only recommended for testing new compiler features. Currently all builds are published as 0.1-SNAPSHOT. To use a snapshot, we need to [configure a snapshot repository in the build.gradle file](#).
- X.Y.Z: Correspond to release or milestone versions X.Y.Z, updated manually. These are stable builds. Release versions are published to Maven Central Repository. No extra configuration is needed in the build.gradle file.

The correspondence between milestones and versions is displayed below:

Milestone	Version
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

Targeting the JVM

To target the JVM, the Kotlin plugin needs to be applied

```
apply plugin: "kotlin"
```

As of M11, Kotlin sources can be mixed with Java sources in the same folder, or in different folders. The default convention is using different folders:

```
project
- 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.

Configuring Dependencies

We need to add dependencies on kotlin-gradle-plugin and the Kotlin standard library:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:<version>'
    }
}

apply plugin: "kotlin" // or apply plugin: "kotlin2js" if targeting JavaScript

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.jetbrains.kotlin:kotlin-stdlib:<version>'
}
```

Using Snapshot versions

If we want to use a snapshot version (nightly build), we need to add the snapshot repository and change the version to 0.1-SNAPSHOT:

```
buildscript {
    repositories {
        mavenCentral()
        maven {
            url 'http://oss.sonatype.org/content/repositories/snapshots'
        }
    }
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:0.1-SNAPSHOT'
    }
}

apply plugin: "kotlin" // or apply plugin: "kotlin2js" if targeting JavaScript

repositories {
    mavenCentral()
    maven {
        url 'http://oss.sonatype.org/content/repositories/snapshots'
    }
}

dependencies {
    compile 'org.jetbrains.kotlin:kotlin-stdlib:0.1-SNAPSHOT'
}
```

Using External Annotations

External annotations for JDK and Android SDK will be configured automatically. If we want to add more annotations for some libraries, we need to add the following line to the Gradle script:

```
kotlinOptions.annotations = file('<path to annotations>')
```

Examples

The [Kotlin Repository](#) contains examples:

- [Kotlin](#)
- [Mixed Java and Kotlin](#)
- [Android](#)
- [JavaScript](#)

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 main design goals behind this project are

- To create a Java-compatible language,
- That compiles at least as fast as Java,
- Make it safer than Java, i.e. statically check for common pitfalls such as null pointer dereference,
- Make it more concise than Java by supporting variable type inference, higher-order functions (closures), extension functions, mixins and first-class delegation, etc;
- And, keeping the useful level of expressiveness (see above), make it way simpler than the most mature competitor – Scala.

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

Is it Java Compatible?

Yes. The compiler emits Java byte-code. Kotlin can call Java, and Java can call Kotlin. See [Java interoperability](#).

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-Orientated language. However it has support for higher-order functions as well as function literals and top-level functions. In addition, there are a good number of common functional language constructs in the standard Kotlin 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 also does not have wildcard types. Inline functions support reified type parameters.

Are semicolons required?

No. They are optional.

Are curly braces required?

Yes.

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 effect tooling?

No. It 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: Operator overloading, Custom Control Structures via inline functions, Infix function calls, Extension Functions, Annotations and language quotations.

What ECMAScript level does the JavaScript support?

Currently at 5.

Does the JavaScript back-end support module systems?

Yes. There are plans to provide 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

- [Function literals](#) + [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](#)

Comparison to Scala

The two main design goals for Kotlin are:

- Make compilation at least as fast as Java
- Keep useful level of expressiveness while maintaining the language simple as possible

Taking this into account, if you are happy with Scala, you probably 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](#)
- Built-in XML
 - See [Type-safe Groovy-style builders](#)

Things that may be added to Kotlin later:

- Structural types
- Value types
- Yield operator
- Actors
- Parallel collections

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

