Chapter 1: Getting started with Kotlin

Remarks

Kotlin is a statically-typed object-oriented programming language developed by JetBrains primarily targeting the JVM. Kotlin is developed with the goals of being quick to compile, backwards-compatible, very type safe, and 100% interoperable with Java. Kotlin is also developed with the goal of providing many of the features wanted by Java developers. Kotlin's standard compiler allows it to be compiled both into Java bytecode for the JVM and into JavaScript.

Compiling Kotlin

Kotlin has a standard IDE plugin for Eclipse and IntelliJ. Kotlin can also be compiled using Maven, using Ant, and using Gradle, or through the command line.

It is worth noting in `$ kotlinc Main.kt` will output a java class file, in this case `MainKt.class` (Note the Kt appended to the class name). However if one was to run the class file using `$ java MainKt` java will throw the following exception:

```
Exception in thread "main" java.lang.NoClassDefFoundError: kotlin/jvm/internal/Intrinsics
    at MainKt.main(Main.kt)
Caused by: java.lang.ClassNotFoundException: kotlin.jvm.internal.Intrinsics
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
```

In order to run the resulting class file using Java, one must include the Kotlin runt-time jar file to the current class path.

```
java -cp .:/path/to/kotlin/runtime/jar/kotlin-runtime.jar MainKt
```

Versions

Version	Release Date
1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30
1.0.4	2016-09-22

Version	Release Date
1.0.5	2016-11-08
1.0.6	2016-12-27
1.1.0	2017-03-01
1.1.1	2017-03-14
1.1.2	2017-04-25
1.1.3	2017-06-23

Examples

Hello World

All Kotlin programs start at the `main` function. Here is an example of a simple Kotlin "Hello World" program:

```
package my.program

fun main(args: Array<String>) {
    println("Hello, world!")
}
```

Place the above code into a file named `Main.kt` (this filename is entirely arbitrary)

When targeting the JVM, the function will be compiled as a static method in a class with a name derived from the filename. In the above example, the main class to run would be `my.program.MainKt`.

To change the name of the class that contains top-level functions for a particular file, place the following annotation at the top of the file above the package statement:

```
@file:JvmName("MyApp")
```

In this example, the main class to run would now be `my.program.MyApp`.

See also:

- Package level functions including `@JvmName` annotation.
- Annotation use-site targets

Hello World using an Object Declaration

You can alternatively use an Object Declaration that contains the main function for a Kotlin program.

```
package my.program

object App {
    @JvmStatic fun main(args: Array<String>) {
        println("Hello World")
    }
}
```

The class name that you will run is the name of your object, in this case is `my.program.App`.

The advantage to this method over a top-level function is that the class name to run is more self-evident, and any other functions you add are scoped into the class `App`. You then also have a singleton instance of `App` to store state and do other work.

See also:

Hello World using a Companion Object

Similar to using an Object Declaration, you can define the `main` function of a Kotlin program using a [Companion Object](#) of a class.

```
package my.program

class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            println("Hello World")
        }
    }
}
```

The class name that you will run is the name of your class, in this case is `my.program.App`.

The advantage to this method over a top-level function is that the class name to run is more self-evident, and any other functions you add are scoped into the class `App`. This is similar to the `Object Declaration` example, other than you are in control of instantiating any classes to do further work.

A slight variation that instantiates the class to do the actual "hello":

```
class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            App().run()
        }
    }

    fun run() {
        println("Hello World")
    }
}
```

See also:

- Static Methods including the @JvmStatic annotation

Main methods using varargs

All of these main method styles can also be used with varargs:

```
package my.program

fun main(vararg args: String) {
    println("Hello, world!")
}
```

Compile and Run Kotlin Code in Command Line

As java provide two different commands to compile and run Java code. Same as Kotlin also provide you different commands.

`javac` to compile java files. `java` to run java files.

Same as `kotlinc` to compile kotlin files `kotlin` to run kotlin files.

Reading input from Command Line

The arguments passed from the console can be received in the Kotlin program and it can be used as an input. You can pass N (1 2 3 and so on) numbers of arguments from the command prompt.

A simple example of a command-line argument in Kotlin.

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

    println("Enter Two number")
    var (a, b) = readLine()!!.split(' ') // !! this operator use for
NPE(NullPointerException).

    println("Max number is : ${maxNum(a.toInt(), b.toInt())}")
}



fun maxNum(a: Int, b: Int): Int {

    var max = if (a > b) {
        println("The value of a is $a");
        a
    } else {
        println("The value of b is $b")
        b
    }

    return max;

}
```

Here, Enter two number from the command line to find the maximum number. Output :

```
Enter Two number
71 89 // Enter two number from command line

The value of b is 89
Max number is: 89
```

For !! Operator Please check Null Safety.

Note: Above example compile and run on Intellij.

Read Getting started with Kotlin online: https://riptutorial.com/kotlin/topic/490/getting-started-with-kotlin

Chapter 2: Annotations

Examples

Declaring an annotation

Annotations are means of attaching metadata to code. To declare an annotation, put the annotation modifier in front of a class:

```
annotation class Strippable
```

Annotations can have meta-anotations:

```
    @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
    annotation class Strippable
```

Annotations, like other classes, can have constructors:

```
annotation class Strippable(val importanceValue: Int)
```

But unlike other classes, is limited to the following types:

- types that correspond to Java primitive types (Int, Long etc.);
- strings
- classes (Foo:: class)
- enums
- other annotations
- arrays of the types listed above

Meta-annotations

When declaring an annotation, meta-info can be included using the following meta-annotations:

- `@Target`: specifies the possible kinds of elements which can be annotated with the annotation (classes, functions, properties, expressions etc.)

- `@Retention` specifies whether the annotation is stored in the compiled class files and whether it's visible through reflection at runtime (by default, both are true.)

- `@Repeatable` allows using the same annotation on a single element multiple times.

- `@MustBeDocumented` specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

Example:

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

Read Annotations online: https://riptutorial.com/kotlin/topic/4074/annotations

Chapter 3: Arrays

Examples

Generic Arrays

Generic arrays in Kotlin are represented by `Array<T>`.

To create an empty array, use `emptyArray<T>()` factory function:

```
val empty = emptyArray<String>()
```

To create an array with given size and initial values, use the constructor:

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

Arrays have `get(index: Int): T` and `set(index: Int, value: T)` functions:

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

Arrays of Primitives

These types **do not** inherit from `Array<T>` to avoid boxing, however, they have the same attributes and methods.

Kotlin type	Factory function	JVM type
BooleanArray	booleanArrayOf(true, false)	boolean[]
ByteArray	byteArrayOf(1, 2, 3)	byte[]
CharArray	charArrayOf('a', 'b', 'c')	char[]
DoubleArray	doubleArrayOf(1.2, 5.0)	double[]
FloatArray	floatArrayOf(1.2, 5.0)	float[]
IntArray	intArrayOf(1, 2, 3)	int[]
LongArray	longArrayOf(1, 2, 3)	long[]
ShortArray	shortArrayOf(1, 2, 3)	short[]

Extensions

`average()` is defined for `Byte`, `Int`, `Long`, `Short`, `Double`, `Float` and always returns `Double`:

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // prints 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // prints 2.5
```

`component1()`, `component2()`, **...** `component5()` return an item of the array

`getOrNull(index: Int)` returns null if index is out of bounds, otherwise an item of the array

`first()`, `last()`

`toHashSet()` returns a `HashSet<T>` of all elements

`sortedArray()`, `sortedArrayDescending()` creates and returns a new array with sorted elements of current

`sort()`, `sortDescending` sort the array in-place

`min()`, `max()`

Iterate Array

You can print the array elements using the loop same as the Java enhanced loop, but you need to change keyword from `:` to `in`.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

You can also change data type in for loop.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```

Create an array

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

Create an array using a closure

```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

Create an uninitialized array

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

The returned array will always have a nullable type. Arrays of non-nullable items can't be created uninitialized.

Read Arrays online: https://riptutorial.com/kotlin/topic/5722/arrays

Chapter 4: Basic Lambdas

Syntax

- Explicit parameters:

- { parameterName: ParameterType, otherParameterName: OtherParameterType ->
  anExpression() }

- Inferred parameters:

- val addition: (Int, Int) -> Int = { a, b -> a + b }

- Single parameter `it` shorthand

- val square: (Int) -> Int = { it*it }

- Signature:

- () -> ResultType

- (InputType) -> ResultType

- (InputType1, InputType2) -> ResultType

Remarks

Input type parameters can be left out when they can be left out when they can be inferred from the context. For example say you have a function on a class that takes a function:

```
data class User(val fistName: String, val lastName: String) {
    fun username(userNameGenerator: (String, String) -> String) =
        userNameGenerator(firstName, secondName)
}
```

You can use this function by passing a lambda, and since the parameters are already specified in the function signature there's no need to re-declare them in the lambda expression:

```
val user = User("foo", "bar")
println(user.userName { firstName, secondName ->
    "${firstName.toUppercase}"_"${secondName.toUppercase}"
 }) // prints FOO_BAR
```

This also applies when you are assigning a lambda to a variable:

```
//valid:
val addition: (Int, Int) = { a, b -> a + b }
//valid:
val addition = { a: Int, b: Int -> a + b }
```

```
//error (type inference failure):
val addition = { a, b -> a + b }
```

When the lambda takes one parameter, and the type can be inferred from the context, you can refer to the parameter by `it`.

```
listOf(1,2,3).map { it * 2 } // [2,4,6]
```

Examples

Lambda as parameter to filter function

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

Lambda passed as a variable

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }
val allowedUsers = users.filter(isOfAllowedAge)
```

Lambda for benchmarking a function call

General-purpose stopwatch for timing how long a function takes to run:

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
            body()
        } finally {
            val end = Instant.now()
            return Duration.between(start, end)
        }
    }
}
```

Usage:

```
val time = Benchmark.realtime({
    // some long-running code goes here ...
})
println("Executed the code in $time")
```

Read Basic Lambdas online: https://riptutorial.com/kotlin/topic/5878/basic-lambdas

Chapter 5: Basics of Kotlin

Introduction

This topic covers the basics of Kotlin for beginners.

Remarks

1. Kotlin file has an extension .kt.
2. All classes in Kotlin have a common superclass Any, that is a default super for a class with no supertypes declared(similar to Object in Java).
3. Variables can be declared as val(immutable- assign once) or var(mutables- value can be changed)
4. Semicolon is not needed at end of statement.
5. If a function does not return any useful value, its return type is Unit.It is also optional.
   6.Referential equality is checked by the === operation. a === b evaluates to true if and only if a and b point to the same object.

Examples

Basic examples

1.The Unit return type declaration is optional for functions. The following codes are equivalent.

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
}

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

2.Single-Expression functions:When a function returns a single expression, the curly braces can be omitted and the body is specified after = 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
```

3.String interpolation: Using string values is easy.

```
In java:
    int num=10
```

```
    String s  = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4.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
```

5.In Kotlin,== actually checks for equality of values.By convention, an expression like a == b is translated to

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

Read Basics of Kotlin online: https://riptutorial.com/kotlin/topic/10648/basics-of-kotlin

---

Chapter 6: Class Delegation

Introduction

A Kotlin class may implement an interface by delegating its methods and properties to another object that implements that interface. This provides a way to compose behavior using association rather than inheritance.

Examples

Delegate a method to another class

```
interface Foo {
    fun example()
}

class Bar {
    fun example() {
        println("Hello, world!")
    }
}

class Baz(b : Bar) : Foo by b

Baz(Bar()).example()
```

The example prints `Hello, world!`

Read Class Delegation online: https://riptutorial.com/kotlin/topic/10575/class-delegation

Chapter 7: Class Inheritance

Introduction

Any object-oriented programming language has some form of class inheritance. Let me revise:

Imagine you had to program a bunch of fruit: `Apples`, `Oranges` and `Pears`. They all differ in size, shape and color, that's why we have different classes.

But let's say their differences don't matter for a second and you just want a `Fruit`, no matter which exactly? What return type would `getFruit()` have?

The answer is class `Fruit`. We create a new class and make all fruits inherit from it!

Syntax

- open {Base Class}
- class {Derived Class} : {Base Class}({Init Arguments})
- override {Function Definition}
- {DC-Object} is {Base Class} == true

Parameters

Parameter	Details
Base Class	Class that is inherited from
Derived Class	Class that inherits from Base Class
Init Arguments	Arguments passed to constructor of Base Class
Function Definition	Function in Derived Class that has different code than the same in the Base Class
DC-Object	"Derived Class-Object" Object that has the type of the Derived Class

Examples

Basics: the 'open' keyword

In Kotlin, classes are **final by default** which means they cannot be inherited from.

To allow inheritance on a class, use the `open` keyword.

```
open class Thing {
    // I can now be extended!
}
```

Note: abstract classes, sealed classes and interfaces will be `open` by default.

Inheriting fields from a class

Defining the base class:

```
open class BaseClass {
    val x = 10
}
```

Defining the derived class:

```
class DerivedClass: BaseClass() {
    fun foo() {
        println("x is equal to " + x)
    }
}
```

Using the subclass:

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // prints: 'x is equal to 10'
}
```

Inheriting methods from a class

Defining the base class:

```
open class Person {
    fun jump() {
        println("Jumping...")
    }
}
```

Defining the derived class:

```
class Ninja: Person() {
    fun sneak() {
        println("Sneaking around...")
    }
}
```

The Ninja has access to all of the methods in Person

```
fun main(args: Array<String>) {
    val ninja = Ninja()
    ninja.jump()  // prints: 'Jumping...'
    ninja.sneak() // prints: 'Sneaking around...'
}
```

Overriding properties and methods

Overriding properties (both read-only and mutable):

```
abstract class Car {
    abstract val name: String;
    open var speed: Int = 0;
}

class BrokenCar(override val name: String) : Car() {
    override var speed: Int
        get() = 0
        set(value) {
            throw UnsupportedOperationException("The car is bloken")
        }
}

fun main(args: Array<String>) {
    val car: Car = BrokenCar("Lada")
    car.speed = 10
}
```

Overriding methods:

```
interface Ship {
    fun sail()
    fun sink()
}

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

    override fun sink() {
        canSail = false
    }
}
```

Read Class Inheritance online: https://riptutorial.com/kotlin/topic/5622/class-inheritance

Chapter 8: Collections

Introduction

Unlike many languages, Kotlin distinguishes between mutable and immutable collections (lists, sets, maps, etc). Precise control over exactly when collections can be edited is useful for eliminating bugs, and for designing good APIs.

Syntax

- listOf, mapOf and setOf returns read-only objects that you cannot add or remove items.
- If you want to add or remove items you have to use arrayListOf, hashMapOf, hashSetOf, linkedMapOf (LinkedHashMap), linkedSetOf (LinkedHashSet), mutableListOf (The Kotlin MultableList collection), mutableMapOf (The Kotlin MultableMap collection), mutableSetOf (The Kotlin MultableSet collection), sortedMapOf or sortedSetOf
- Each collection has methods like first(), last(), get() and lambda functions like filter, map, join, reduce and many others.

Examples

Using list

```
// Create a new read-only List<String>
val list = listOf("Item 1", "Item 2", "Item 3")
println(list) // prints "[Item 1, Item 2, Item 3]"
```

Using map

```
// Create a new read-only Map<Integer, String>
val map = mapOf(Pair(1, "Item 1"), Pair(2, "Item 2"), Pair(3, "Item 3"))
println(map) // prints "{1=Item 1, 2=Item 2, 3=Item 3}"
```

Using set

```
// Create a new read-only Set<String>
val set = setOf(1, 3, 5)
println(set) // prints "[1, 3, 5]"
```

Read Collections online: https://riptutorial.com/kotlin/topic/8846/collections

Chapter 9: Conditional Statements

Remarks

In contrast to Java's `switch`, the `when` statement has no fall-through behavior. This means, that if a branch is matched, the control flow returns after its execution and no `break` statement is required. If you want to combine the bahaviors for multiple arguments, you can write multiple arguments separated by commas:

```
when (x) {
    "foo", "bar" -> println("either foo or bar")
    else -> println("didn't match anything")
}
```

Examples

Standard if-statement

```
val str = "Hello!"
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

The else-branches are optional in normal if-statements.

If-statement as an expression

If-statements can be expressions:

```
val str = if (condition) "Condition met!" else "Condition not met!"
```

Note that the `else`-branch is not optional if the `if`-statement is used as an expression.

This can also been done with a multi-line variant with curly brackets and multiple `else if` statements.

```
val str = if (condition1){
    "Condition1 met!"
    } else if (condition2) {
    "Condition2 met!"
    } else {
    "Conditions not met!"
    }
```

TIP: Kotlin can infer the type of the variable for you but if you want to be sure of the type just annotate it on the variable like: `val str: String =` this will enforce the type and will make it easier to read.

When-statement instead of if-else-if chains

The when-statement is an alternative to an if-statement with multiple else-if-branches:

```
when {
    str.length == 0 -> print("The string is empty!")
    str.length > 5  -> print("The string is short!")
    else            -> print("The string is long!")
}
```

Same code written using an *if-else-if* chain:

```
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

Just like with the if-statement, the else-branch is optional, and you can add as many or as few branches as you like. You can also have multiline-branches:

```
when {
    condition -> {
        doSomething()
        doSomeMore()
    }
    else -> doSomethingElse()
}
```

When-statement argument matching

When given an argument, the `when`-statement matches the argument against the branches in sequence. The matching is done using the `==` operator which performs null checks and compares the operands using the `equals` function. The first matching one will be executed.

```
when (x) {
    "English" -> print("How are you?")
    "German" -> print("Wie geht es dir?")
    else -> print("I don't know that language yet :(")
}
```

The when statement also knows some more advanced matching options:

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
when (x) {
    in names -> print("I know that name!")
```

```
    !in 1..10 -> print("Argument was not in the range from 1 to 10")
    is String -> print(x.length) // Due to smart casting, you can use String-functions here
}
```

When-statement as expression

Like if, when can also be used as an expression:

```
val greeting = when (x) {
    "English" -> "How are you?"
    "German" -> "Wie geht es dir?"
    else -> "I don't know that language yet :("
}
print(greeting)
```

To be used as an expression, the when-statement must be exhaustive, i.e. either have an else branch or cover all possibilities with the branches in another way.

When-statement with enums

when can be used to match enum values:

```
enum class Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

fun doOnDay(day: Day) {
    when(day) {
        Day.Sunday ->     // Do something
        Day.Monday, Day.Tuesday ->     // Do other thing
        Day.Wednesday -> // ...
        Day.Thursday ->  // ...
        Day.Friday ->    // ...
        Day.Saturday ->  // ...
    }
}
```

As you can see in second case line (Monday and Tuedsay) it is also possible to combine two or more enum values.

If your cases are not exhaustive the compile will show an error. You can use else to handle default cases:

```
fun doOnDay(day: Day) {
    when(day) {
        Day.Monday ->    // Work
        Day.Tuesday ->   // Work hard
        Day.Wednesday -> // ...
```

```
        Day.Thursday ->   //
        Day.Friday ->     //
        else ->           // Party on weekend
    }
}
```

Though the same can be done using `if-then-else` construct, `when` takes care of missing `enum` values and makes it more natural.

Check here for more information about kotlin `enum`

Chapter 10: Configuring Kotlin build

Examples

Gradle configuration

`kotlin-gradle-plugin` is used to compile Kotlin code with Gradle. Basically, its version should correspond to the Kotlin version you want to use. E.g. if you want to use Kotlin `1.0.3`, then you need to aplly `kotlin-gradle-plugin` version `1.0.3` too.

It's a good idea to externalize this version in `gradle.properties` or in `ExtraPropertiesExtension`:

```
buildscript {
    ext.kotlin_version = '1.0.3'

    repositories {
      mavenCentral()
    }

    dependencies {
      classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

Then you need to apply this plugin to your project. The way you do this differs when targeting different platforms:

Targeting JVM

```
apply plugin: 'kotlin'
```

Targeting Android

```
apply plugin: 'kotlin-android'
```

Targeting JS

```
apply plugin: 'kotlin2js'
```

These are the default paths:

- kotlin sources: `src/main/kotlin`
- java sources: `src/main/java`
- kotlin tests: `src/test/kotlin`
- java tests: `src/test/java`

- runtime resources: `src/main/resources`
- test resources: `src/test/resources`

You may need to configure `SourceSets` if you're using custom project layout.

Finally, you'll need to add Kotlin standard library dependency to your project:

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

If you want to use Kotlin Reflection you'll also need to add `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`
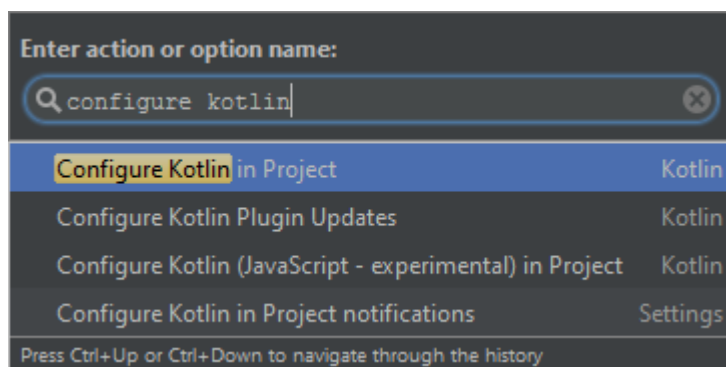
Using Android Studio

Android Studio can configure Kotlin automatically in an Android project.

Install the plugin

To install the Kotlin plugin, go to File > Settings > Editor > Plugins > Install JetBrains Plugin... > Kotlin > Install, then restart Android Studio when prompted.

Configure a project

Create an Android Studio project as normal, then press `Ctrl` + `Shift` + `A`. In the search box, type "Configure Kotlin in Project" and press Enter.

Android Studio will alter your Gradle files to add all the necessary dependencies.

Converting Java

To convert your Java files to Kotlin files, press `Ctrl` + `Shift` + `A` and find "Convert Java File to Kotlin File". This will change the current file's extension to `.kt` and convert the code to Kotlin.

```java
package com.orangeflash81.myapplication;

public class Foo {
    private String name = "Joe Bloggs";

    String getName() { return name; }

    void setName(String value) { name = value; }
}
```

Migrating from Gradle using Groovy script to Kotlin script

Steps:

- clone the gradle-script-kotlin project

- copy/paste from the cloned project to your project:

  - `build.gradle.kts`
  - `gradlew`
  - `gradlew.bat`
  - `settings.gradle`

- update the content of the `build.gradle.kts` based on your needs, you can use as inspiration the scripts in the project just cloned or in one of its samples

- now open Intellij and open your project, in the explorer window, it should be recognized as a Gradle project, if not, expand it first.

- after opening, let Intellij works, open `build.gradle.kts` and check if there are any error. If the

highlighting is not working and/or is everything marked red, then close and reopen Intellij

- open the Gradle window and refresh it

If you are on Windows, you may encounter this bug, download the full Gradle 3.3 distribution and use that instead the one provided. Related.

OSX and Ubuntu work out of the box.

Small bonus, if you want to avoid all the hassle of publicing on Maven and similar, use Jitpack, the lines to add are almost identical compared to Groovy. You can take inspiration from this project of mine.

Read Configuring Kotlin build online: https://riptutorial.com/kotlin/topic/2501/configuring-kotlin-build

Chapter 11: coroutines

Introduction

Examples of Kotlin's experimental(yet) implementation of coroutines

Examples

Simple coroutine which delay's 1 second but not blocks

(from official doc)

```
fun main(args: Array<String>) {
    launch(CommonPool) { // create new coroutine in common thread pool
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main function continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

result

```
Hello,
World!
```

Read coroutines online: https://riptutorial.com/kotlin/topic/10936/coroutines

Chapter 12: Delegated properties

Introduction

Kotlin can delegate the implementation of a property to a handler object. Some standard handlers are included, such as lazy initialization or observable properties. Custom handlers can also be created.

Examples

Lazy initialization

```
val foo : Int by lazy { 1 + 1 }
println(foo)
```

The example prints 2.

Observable properties

```
var foo : Int by Delegates.observable("1") { property, oldValue, newValue ->
    println("${property.name} was changed from $oldValue to $newValue")
}
foo = 2
```

The example prints foo was changed from 1 to 2

Map-backed properties

```
val map = mapOf("foo" to 1)
val foo : String by map
println(foo)
```

The example prints 1

Custom delegation

```
class MyDelegate {
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        return "Delegated value"
    }
}

val foo : String by MyDelegate()
println(foo)
```

The example prints Delegated value

Delegate Can be used as a layer to reduce boilerplate

Consider Kotlin's Null Type system and `WeakReference<T>`.

So let's say we have to save some sort of reference and we wanted to avoid memory leaks, here is where `WeakReference` comes in.

take for example this:

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: WeakReference<MyMemoryExpensiveClass>? = null

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let {
                it.get()?.let(block)
            }
        }
    }

    init {
        reference = WeakReference(this)
    }
}
```

Now this is just with one WeakReference. To Reduce this boilerplate, we can use a custom property delegate to help us like so:

```
class WeakReferenceDelegate<T>(initialValue: T? = null) : ReadWriteProperty<Any, T?> {
    var reference = WeakReference(initialValue)
        private set

    override fun getValue(thisRef: Any, property: KProperty<*>): T? = reference.get()

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        reference = WeakReference(value)
    }
}
```

So Now we can use variables that are wrapped with `WeakReference` just like normal nullable variables !

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: MyMemoryExpensiveClass? by
WeakReferenceDelegate<MyMemoryExpensiveClass>()

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let(block)
        }
    }

    init {
        reference = this
    }
```

```
}
```

Read Delegated properties online: https://riptutorial.com/kotlin/topic/10571/delegated-properties

Chapter 13: DSL Building

Introduction

Focus on the syntax details to design internal DSLs in Kotlin.

Examples

Infix approach to build DSL

If you have:

```
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

you can write the following DSL-like code in your tests:

```
@Test
fun test() {
  100.plusOne() shouldBe 101
}
```

Overriding invoke method to build DSL

If you have:

```
class MyExample(val i: Int) {
  operator fun <R> invoke(block: MyExample.() -> R) = block()
  fun Int.bigger() = this > i
}
```

you can write the following DSL-like code in your production code:

```
fun main2(args: Array<String>) {
    val ex = MyExample(233)
    ex {
        // bigger is defined in the context of `ex`
        // you can only call this method inside this context
        if (777.bigger()) kotlin.io.println("why")
    }
}
```

Using operators with lambdas

If you have:

```
val r = Random(233)
infix inline operator fun Int.rem(block: () -> Unit) {
```

```
    if (r.nextInt(100) < this) block()
}
```

You can write the following DSL-like code:

```
20 % { println("The possibility you see this message is 20%") }
```

Using extensions with lambdas

If you have:

```
operator fun <R> String.invoke(block: () -> R) = {
  try { block.invoke() }
  catch (e: AssertException) { System.err.println("$this\n${e.message}") }
}
```

You can write the following DSL-like code:

```
"it should return 2" {
    parse("1 + 1").buildAST().evaluate() shouldBe 2
}
```

If you feel confused with shouldBe above, see the example Infix approach to build DSL.

Read DSL Building online: https://riptutorial.com/kotlin/topic/10042/dsl-building

Chapter 14: Enum

Remarks

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

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

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

Examples

Initialization

Enum classes as any other classes can have a constructor and be initialized

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

Functions and Properties in enums

Enum classes can also declare members (i.e. properties and functions). A semicolon (`;`) must be placed between the last enum object and the first member declaration.

If a member is `abstract`, the enum objects must implement it.

```
enum class Color {
    RED {
        override val rgb: Int = 0xFF0000
    },
    GREEN {
        override val rgb: Int = 0x00FF00
```

```
    },
    BLUE {
        override val rgb: Int = 0x0000FF
    }

    ;

    abstract val rgb: Int

    fun colorString() = "#%06X".format(0xFFFFFF and rgb)
}
```

Simple enum

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

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

Mutability

Enums can be mutable, this is another way to obtain a singleton behavior:

```
enum class Planet(var population: Int = 0) {
    EARTH(7 * 100000000),
    MARS();

    override fun toString() = "$name[population=$population]"
}

 println(Planet.MARS) // MARS[population=0]
 Planet.MARS.population = 3
 println(Planet.MARS) // MARS[population=3]
```

Read Enum online: https://riptutorial.com/kotlin/topic/2286/enum

Chapter 15: Exceptions

Examples

Catching exception with try-catch-finally

Catching exceptions in Kotlin looks very similar to Java

```
try {
    doSomething()
}
catch(e: MyException) {
    handle(e)
}
finally {
    cleanup()
}
```

You can also catch multiple exceptions

```
try {
    doSomething()
}
catch(e: FileSystemException) {
    handle(e)
}
catch(e: NetworkException) {
    handle(e)
}
catch(e: MemoryException) {
    handle(e)
}
finally {
    cleanup()
}
```

`try` is also an expression and may return value

```
val s: String? = try { getString() } catch (e: Exception) { null }
```

Kotlin doesn't have checked exceptions, so you don't have to catch any exceptions.

```
fun fileToString(file: File) : String {
    //readAllBytes throws IOException, but we can omit catching it
    fileContent = Files.readAllBytes(file)
    return String(fileContent)
}
```

Read Exceptions online: https://riptutorial.com/kotlin/topic/7246/exceptions

Chapter 16: Extension Methods

Syntax

- fun TypeName.extensionName(params, ...) { /* body */ } // Declaration
- fun <T: Any> TypeNameWithGenerics<T>.extensionName(params, ...) { /* body */ } // Declaration with Generics
- myObj.extensionName(args, ...) // invocation

Remarks

Extensions are resolved **statically**. This means that the extension method to be used is determined by the reference-type of the variable you are accessing; it doesn't matter what the variable's type is at runtime, the same extension method will always be called. This is because **declaring an extension method doesn't actually add a member to the receiver type**.

Examples

Top-Level Extensions

Top-level extension methods are not contained within a class.

```
fun IntArray.addTo(dest: IntArray) {
    for (i in 0 .. size – 1) {
        dest[i] += this[i]
    }
}
```

Above an extension method is defined for the type `IntArray`. Note that the object for which the extension method is defined (called the **receiver**) is accessed using the keyword `this`.

This extension can be called like so:

```
val myArray = intArrayOf(1, 2, 3)
intArrayOf(4, 5, 6).addTo(myArray)
```

Potential Pitfall: Extensions are Resolved Statically

The extension method to be called is determined at compile-time based on the reference-type of the variable being accessed. It doesn't matter what the variable's type is at runtime, the same extension method will always be called.

```
open class Super

class Sub : Super()
```

```
fun Super.myExtension() = "Defined for Super"

fun Sub.myExtension() = "Defined for Sub"

fun callMyExtension(myVar: Super) {
    println(myVar.myExtension())
}

callMyExtension(Sub())
```

The above example will print `"Defined for Super"`, because the declared type of the variable `myVar` is `Super`.

Sample extending long to render a human readable string

Given any value of type `Int` or `Long` to render a human readable string:

```
fun Long.humanReadable(): String {
    if (this <= 0) return "0"
    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    val digitGroups = (Math.log10(this.toDouble())/Math.log10(1024.0)).toInt();
    return DecimalFormat("#,##0.#").format(this/Math.pow(1024.0, digitGroups.toDouble())) + "
" + units[digitGroups];
}

fun Int.humanReadable(): String {
    return this.toLong().humanReadable()
}
```

Then easily used as:

```
println(1999549L.humanReadable())
println(someInt.humanReadable())
```

Sample extending Java 7+ Path class

A common use case for extension methods is to improve an existing API. Here are examples of adding `exist`, `notExists` and `deleteRecursively` to the Java 7+ `Path` class:

```
fun Path.exists(): Boolean = Files.exists(this)
fun Path.notExists(): Boolean = !this.exists()
fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()
```

Which can now be invoked in this example:

```
val dir = Paths.get(dirName)
if (dir.exists()) dir.deleteRecursively()
```

Using extension functions to improve readability

In Kotlin you could write code like:

```
val x: Path = Paths.get("dirName").apply {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

But the use of `apply` is not that clear as to your intent. Sometimes it is clearer to create a similar extension function to in effect rename the action and make it more self-evident. This should not be allowed to get out of hand, but for very common actions such as verification:

```
infix inline fun <T> T.verifiedBy(verifyWith: (T) -> Unit): T {
    verifyWith(this)
    return this
}

infix inline fun <T: Any> T.verifiedWith(verifyWith: T.() -> Unit): T {
    this.verifyWith()
    return this
}
```

You could now write the code as:

```
val x: Path = Paths.get("dirName") verifiedWith {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

Which now let's people know what to expect within the lambda parameter.

Note that the type parameter `T` for `verifiedBy` is same as `T: Any?` meaning that even nullable types will be able to use that version of the extension. Although `verifiedWith` requires non-nullable.

Sample extending Java 8 Temporal classes to render an ISO formatted string

With this declaration:

```
fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)
```

You can now simply:

```
val dateAsString = someInstant.toIsoString()
```

Extension functions to Companion Objects (appearance of Static functions)

If you want to extend a class as-if you are a static function, for example for class `Something` add static looking function `fromString`, this can only work if the class has a companion object and that the extension function has been declared upon the companion object:

```
class Something {
    companion object {}
}
```

```
class SomethingElse {
}

fun Something.Companion.fromString(s: String): Something = ...

fun SomethingElse.fromString(s: String): SomethingElse = ...

fun main(args: Array<String>) {
    Something.fromString("") //valid as extension function declared upon the
                             //companion object

    SomethingElse().fromString("") //valid, function invoked on instance not
                                   //statically

    SomethingElse.fromString("") //invalid
}
```

Lazy extension property workaround

Assume you want to create an extension property that is expensive to compute. Thus you would like to cache the computation, by using the lazy property delegate and refer to current instance (`this`), but you cannot do it, as explained in the Kotlin issues KT-9686 and KT-13053. However, there is an official workaround provided here.

In the example, the extension property is `color`. It uses an explicit `colorCache` which can be used with `this` as no `lazy` is necessary:

```
class KColor(val value: Int)

private val colorCache = mutableMapOf<KColor, Color>()

val KColor.color: Color
    get() = colorCache.getOrPut(this) { Color(value, true) }
```

Extensions for easier reference View from code

You can use extensions for reference View, no more boilerplate after you created the views.

Original Idea is by Anko Library

Extensions

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
inline fun <reified T : View> RecyclerView.ViewHolder.find(id: Int): T =
itemView?.findViewById(id) as T

inline fun <reified T : View> View.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Activity.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Fragment.findOptional(id: Int): T? = view?.findViewById(id) as?
T
inline fun <reified T : View> RecyclerView.ViewHolder.findOptional(id: Int): T? =
```

```
itemView?.findViewById(id) as? T
```

Usage

```
val yourButton by lazy { find<Button>(R.id.yourButtonId) }
val yourText by lazy { find<TextView>(R.id.yourTextId) }
val yourEdittextOptional by lazy { findOptional<EditText>(R.id.yourOptionEdittextId) }
```

Read Extension Methods online: https://riptutorial.com/kotlin/topic/613/extension-methods

Chapter 17: Functions

Syntax

- fun ***Name***(***Params***) = ...
- fun ***Name***(***Params***) {...}
- fun ***Name***(***Params***): ***Type*** {...}
- fun <***Type Argument***> ***Name***(***Params***): ***Type*** {...}
- inline fun ***Name***(***Params***): ***Type*** {...}
- { ***ArgName***: ***ArgType*** -> ... }
- { ***ArgName*** -> ... }
- { ***ArgNames*** -> ... }
- { (***ArgName***: ***ArgType***): ***Type*** -> ... }

Parameters

Parameter	Details
Name	Name of the function
Params	Values given to the function with a name and type: `Name:Type`
Type	Return type of the function
Type Argument	Type parameter used in generic programming (not necessarily return type)
ArgName	Name of value given to the function
ArgType	Type specifier for ***ArgName***
ArgNames	List of ArgName separated by commas

Examples

Functions Taking Other Functions

As seen in "Lambda Functions", functions can take other functions as a parameter. The "function type" which you'll need to declare functions which take other functions is as follows:

```
# Takes no parameters and returns anything
() -> Any?

# Takes a string and an integer and returns ReturnType
(arg1: String, arg2: Int) -> ReturnType
```

For example, you could use the vaguest type, `() -> Any?`, to declare a function which executes a lambda function twice:

```
fun twice(x: () -> Any?) {
    x(); x();
}

fun main() {
    twice {
        println("Foo")
    } # => Foo
      # => Foo
}
```

Lambda Functions

Lambda functions are anonymous functions which are usually created during a function call to act as a function parameter. They are declared by surrounding expressions with {braces} - if arguments are needed, these are put before an arrow `->`.

```
{ name: String ->
    "Your name is $name" //This is returned
}
```

The last statement inside a lambda function is automatically the return value.

The type's are optional, if you put the lambda on a place where the compiler can infer the types.

Multiple arguments:

```
{ argumentOne:String, argumentTwo:String ->
    "$argumentOne - $argumentTwo"
}
```

If the lambda function only needs one argument, then the argument list can be omitted and the single argument be referred to using `it` instead.

```
{ "Your name is $it" }
```

If the only argument to a function is a lambda function, then parentheses can be completely omitted from the function call.

```
# These are identical
listOf(1, 2, 3, 4).map { it + 2 }
listOf(1, 2, 3, 4).map({ it + 2 })
```

Function References

We can reference a function without actually calling it by prefixing the function's name with `::`. This can then be passed to a function which accepts some other function as a parameter.

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

Functions without a receiver will be converted to `(ParamTypeA, ParamTypeB, ...) -> ReturnType`
where `ParamTypeA`, `ParamTypeB` ... are the type of the function parameters and `ReturnType1 is the
type of function return value.

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
```

Functions with a receiver (be it an extension function or a member function) has a different syntax.
You have to add the type name of the receiver before the double colon:

```
class Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function4<Foo, Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo, Foo0, Foo1, Foo2) -> Bar
// takes 4 parameters, with receiver as first and actual parameters following, in their order

// this function can't be called like an extension function, though
val ref = Foo::foo
Foo().ref(Foo0(), Foo1(), Foo2()) // compile error

class Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

However, when a function's receiver is an object, the receiver is omitted from parameter list,
because these is and only is one instance of such type.

```
object Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
// takes 3 parameters, receiver not needed

object Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

Since kotlin 1.1, function reference can also be *bounded* to a variable, which is then called a

bounded function reference.

1.1.0

```kotlin
fun makeList(last: String?): List<String> {
    val list = mutableListOf("a", "b", "c")
    last?.let(list::add)
    return list
}
```

Note this example is given only to show how bounded function reference works. It's bad practice in all other senses.

There is a special case, though. An extension function declared as a member can't be referenced.

```kotlin
class Foo
class Bar {
    fun Foo.foo() {}
    val ref = Foo::foo // compile error
}
```

Basic Functions

Functions are declared using the `fun` keyword, followed by a function name and any parameters. You can also specify the return type of a function, which defaults to `Unit`. The body of the function is enclosed in braces `{}`. If the return type is other than `Unit`, the body must issue a return statement for every terminating branch within the body.

```kotlin
fun sayMyName(name: String): String {
    return "Your name is $name"
}
```

A shorthand version of the same:

```kotlin
fun sayMyName(name: String): String = "Your name is $name"
```

And the type can be omitted since it can be inferred:

```kotlin
fun sayMyName(name: String) = "Your name is $name"
```

Shorthand Functions

If a function contains just one expression, we can omit the brace brackets and use an equals instead, like a variable assignment. The result of the expression is returned automatically.

```kotlin
fun sayMyName(name: String): String = "Your name is $name"
```

Inline Functions

Functions can be declared inline using the `inline` prefix, and in this case they act like macros in C

- rather than being called, they are replaced by the function's body code at compile time. This can lead to performance benefits in some circumstances, mainly where lambdas are used as function parameters.

```
inline fun sayMyName(name: String) = "Your name is $name"
```

One difference from C macros is that inline functions can't access the scope from which they're called:

```
inline fun sayMyName() = "Your name is $name"

fun main() {
    val name = "Foo"
    sayMyName() # => Unresolved reference: name
}
```

Operator functions

Kotlin allows us to provide implementations for a predefined set of operators with 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. Functions that overload operators need to be marked with the `operator` modifier:

```
data class IntListWrapper (val wrapped: List<Int>) {
    operator fun get(position: Int): Int = wrapped[position]
}

val a = IntListWrapper(listOf(1, 2, 3))
a[1] // == 2
```

More operator functions can be found in here

Read Functions online: https://riptutorial.com/kotlin/topic/1280/functions

Chapter 18: Generics

Introduction

A List can hold numbers, words or really anything. That's why we call the List *generic*.

Generics are basically used to define which types a class can hold and which type an object currently holds.

Syntax

- class *ClassName*<***TypeName***>
- class *ClassName*<*>
- *ClassName*<in ***UpperBound***>
- *ClassName*<out ***LowerBound***>
- class *Name*<***TypeName***:***UpperBound***>

Parameters

Parameter	Details
TypeName	Type Name of generic parameter
UpperBound	Covariant Type
LowerBound	Contravariant Type
ClassName	Name of the class

Remarks

Implied Upper Bound is Nullable

In Kotlin Generics, the upper bound of type parameter `T` would be `Any?`. Therefore for this class:

```
class Consumer<T>
```

The type parameter `T` is really `T: Any?`. To make a non-nullable upper bound, explicitly specific `T: Any`. For example:

```
class Consumer<T: Any>
```

Examples

Declaration-site variance

[Declaration-site variance](#) can be thought of as declaration of use-site variance once and for all the use-sites.

```
class Consumer<in T> { fun consume(t: T) { ... } }

fun charSequencesConsumer() : Consumer<CharSequence>() = ...

val stringConsumer : Consumer<String> = charSequenceConsumer() // OK since in-projection
val anyConsumer : Consumer<Any> = charSequenceConsumer() // Error, Any cannot be passed

val outConsumer : Consumer<out CharSequence> = ... // Error, T is `in`-parameter
```

Widespread examples of declaration-site variance are `List<out T>`, which is immutable so that `T` only appears as the return value type, and `Comparator<in T>`, which only receives `T` as argument.

Use-site variance

[Use-site variance](#) is similar to Java wildcards:

Out-projection:

```
val takeList : MutableList<out SomeType> = ... // Java: List<? extends SomeType>

val takenValue : SomeType = takeList[0] // OK, since upper bound is SomeType

takeList.add(takenValue) // Error, lower bound for generic is not specified
```

In-projection:

```
val putList : MutableList<in SomeType> = ... // Java: List<? super SomeType>

val valueToPut : SomeType = ...
putList.add(valueToPut) // OK, since lower bound is SomeType

putList[0] // This expression has type Any, since no upper bound is specified
```

Star-projection

```
val starList : MutableList<*> = ... // Java: List<?>

starList[0] // This expression has type Any, since no upper bound is specified
starList.add(someValue) // Error, lower bound for generic is not specified
```

See also:

- [Variant Generics](#) interoperability when calling Kotlin from Java.

Chapter 19: Idioms

Examples

Creating DTOs (POJOs/POCOs)

Data classes in kotlin are classes created to do nothing but hold data. Such classes are marked as `data`:

```
data class User(var firstname: String, var lastname: String, var age: Int)
```

The code above creates a `User` class with the following automatically generated:

- Getters and Setters for all properties (getters only for `val`s)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (where `N` is the corresponding property in order of declaration)

Just as with a function, default values can also be specified:

```
data class User(var firstname: String = "Joe", var lastname: String = "Bloggs", var age: Int =
20)
```

More details can be found here Data Classes.

Filtering a list

```
val list = listOf(1,2,3,4,5,6)

//filter out even numbers

val even = list.filter { it % 2 == 0 }

println(even) //returns [2,4]
```

Delegate to a class without providing it in the public constructor

Assume you want to delegate to a class but you do not want to provide the delegated-to class in the constructor parameter. Instead, you want to construct it privately, making the constructor caller unaware of it. At first this might seem impossible because class delegation allows to delegate only to constructor parameters. However, there is a way to do it, as given in this answer:

```
class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table
{

    constructor() : this(TreeBasedTable.create()) // or a different type of table if desired
```

```
    }
```

With this, you can just call the constructor of `MyTable` like that: `MyTable()`. The `Table<Int, Int, Int>` to which `MyTable` delegates will be created privately. Constructor caller knows nothing about it.

This example is based on [this SO question](#).

Serializable and serialVersionUid in Kotlin

To create the `serialVersionUID` for a class in Kotlin you have a few options all involving adding a member to the companion object of the class.

The most concise bytecode comes from a `private const val` which will become a private static variable on the containing class, in this case `MySpecialCase`:

```
class MySpecialCase : Serializable {
    companion object {
        private const val serialVersionUID: Long = 123
    }
}
```

You can also use these forms, **each with a side effect of having getter/setter methods** which are not necessary for serialization...

```
class MySpecialCase : Serializable {
    companion object {
        private val serialVersionUID: Long = 123
    }
}
```

This creates the static field but also creates a getter as well `getSerialVersionUID` on the companion object which is unnecessary.

```
class MySpecialCase : Serializable {
    companion object {
        @JvmStatic private val serialVersionUID: Long = 123
    }
}
```

This creates the static field but also creates a static getter as well `getSerialVersionUID` on the containing class `MySpecialCase` which is unnecessary.

But all work as a method of adding the `serialVersionUID` to a `Serializable` class.

Fluent methods in Kotlin

Fluent methods in Kotlin can be the same as Java:

```
fun doSomething() {
   someOtherAction()
```

```
    return this
}
```

But you can also make them more functional by creating an extension function such as:

```
fun <T: Any> T.fluently(func: ()->Unit): T {
    func()
    return this
}
```

Which then allows more obviously fluent functions:

```
fun doSomething() {
    return fluently { someOtherAction() }
}
```

Use let or also to simplify working with nullable objects

`let` in Kotlin creates a local binding from the object it was called upon. Example:

```
val str = "foo"
str.let {
    println(it) // it
}
```

This will print `"foo"` and will return `Unit`.

The difference between `let` and `also` is that you can return any value from a `let` block. `also` in the other hand will always reutrn `Unit`.

Now why this is useful, you ask? Because if you call a method which can return `null` and you want to run some code only when that return value is not `null` you can use `let` or `also` like this:

```
val str: String? = someFun()
str?.let {
    println(it)
}
```

This piece of code will only run the `let` block when `str` is not `null`. Note the `null` safety operator (`?`).

Use apply to initialize objects or to achieve method chaining

The documentation of `apply` says the following:

> calls the specified function block with `this` value as its receiver and returns `this` value.

While the kdoc is not so helpful `apply` is indeed an useful function. In layman's terms `apply` establishes a scope in which `this` is bound to the object you called `apply` on. This enables you to spare some code when you need to call multiple methods on an object which you will then return

---

later. Example:

```
File(dir).apply { mkdirs() }
```

This is the same as writing this:

```
fun makeDir(String path): File {
    val result = new File(path)
    result.mkdirs()
    return result
}
```

Read Idioms online: https://riptutorial.com/kotlin/topic/2273/idioms

Chapter 20: Interfaces

Remarks

See also: Kotlin reference documentation for Interfaces: Interfaces

Examples

Basic Interface

A Kotlin interface contains declarations of abstract methods, and default method implementations although they cannot store state.

```
interface MyInterface {
    fun bar()
}
```

This interface can now be implemented by a class as follows:

```
class Child : MyInterface {
   override fun bar() {
       print("bar() was called")
   }
}
```

Interface with default implementations

An interface in Kotlin can have default implementations for functions:

```
interface MyInterface {
    fun withImplementation() {
      print("withImplementation() was called")
    }
}
```

Classes implementing such interfaces will be able to use those functions without reimplementing

```
class MyClass: MyInterface {
    // No need to reimplement here
}
val instance = MyClass()
instance.withImplementation()
```

Properties

Default implementations also work for property getters and setters:

```
interface MyInterface2 {
    val helloWorld
        get() = "Hello World!"
}
```

Interface accessors implementations can't use backing fields

```
interface MyInterface3 {
    // this property won't compile!
    var helloWorld: Int
        get() = field
        set(value) { field = value }
}
```

Multiple implementations

When multiple interfaces implement the same function, or all of them define with one or more implementing, the derived class needs to manually resolve proper call

```
interface A {
    fun notImplemented()
    fun implementedOnlyInA() { print("only A") }
    fun implementedInBoth() { print("both, A") }
    fun implementedInOne() { print("implemented in A") }
}

interface B {
    fun implementedInBoth() { print("both, B") }
    fun implementedInOne() // only defined
}

class MyClass: A, B {
    override fun notImplemented() { print("Normal implementation") }

    // implementedOnlyInA() can by normally used in instances

    // class needs to define how to use interface functions
    override fun implementedInBoth() {
        super<B>.implementedInBoth()
        super<A>.implementedInBoth()
    }

    // even if there's only one implementation, there multiple definitions
    override fun implementedInOne() {
        super<A>.implementedInOne()
        print("implementedInOne class implementation")
    }
}
```

Properties in Interfaces

You can declare properties in interfaces. Since an interface cannot have state you can only declare a property as abstract or by providing default implementation for the accessors.

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

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

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

Conflicts when Implementing Multiple Interfaces with Default Implementations

When implementing more than one interface that have methods of the same name that include default implementations, it is ambiguous to the compiler which implementation should be used. In the case of a conflict, the developer must override the conflicting method and provide a custom implementation. That implementation may chose to delegate to the default implementations or not.

```
interface FirstTrait {
    fun foo() { print("first") }
    fun bar()
}

interface SecondTrait {
    fun foo() { print("second") }
    fun bar() { print("bar") }
}

class ClassWithConflict : FirstTrait, SecondTrait {
    override fun foo() {
        super<FirstTrait>.foo()  // delegate to the default implementation of FirstTrait
        super<SecondTrait>.foo() // delegate to the default implementation of SecondTrait
    }

    // function bar() only has a default implementation in one interface and therefore is ok.
}
```

super keyword

```
interface MyInterface {
    fun funcOne() {
        //optional body
        print("Function with default implementation")
    }
}
```

If the method in the interface has its own default implementation, we can use super keyword to access it.

```
super.funcOne()
```

Chapter 21: Java 8 Stream Equivalents

Introduction

Kotlin provides many extension methods on collections and iterables for applying functional-style operations. A dedicated `Sequence` type allows for lazy composition of several such operations.

Remarks

About laziness

If you want to lazy process a chain, you can convert to a `Sequence` using `asSequence()` before the chain. At the end of the chain of functions, you usually end up with a `Sequence` as well. Then you can use `toList()`, `toSet()`, `toMap()` or some other function to materialize the `Sequence` at the end.

```
// switch to and from lazy
val someList = items.asSequence().filter { ... }.take(10).map { ... }.toList()

// switch to lazy, but sorted() brings us out again at the end
val someList = items.asSequence().filter { ... }.take(10).map { ... }.sorted()
```

Why are there no Types?!?

You will notice the Kotlin examples do not specify the types. This is because Kotlin has full type inference and is completely type safe at compile time. More so than Java because it also has nullable types and can help prevent the dreaded NPE. So this in Kotlin:

```
val someList = people.filter { it.age <= 30 }.map { it.name }
```

is the same as:

```
val someList: List<String> = people.filter { it.age <= 30 }.map { it.name }
```

Because Kotlin knows what `people` is, and that `people.age` is `Int` therefore the filter expression only allows comparison to an `Int`, and that `people.name` is a `String` therefore the `map` step produces a `List<String>` (readonly `List` of `String`).

Now, if `people` were possibly `null`, as-in a `List<People>?` then:

```
val someList = people?.filter { it.age <= 30 }?.map { it.name }
```

Returns a `List<String>?` that would need to be null checked (*or use one of the other Kotlin operators for nullable values, see this Kotlin idiomatic way to deal with nullable values and also Idiomatic way of handling nullable or empty list in Kotlin*)

Reusing Streams

In Kotlin, it depends on the type of collection whether it can be consumed more than once. A `Sequence` generates a new iterator every time, and unless it asserts "use only once" it can reset to the start each time it is acted upon. Therefore while the following fails in Java 8 stream, but works in Kotlin:

```
// Java:
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> s.startsWith("b"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

```
// Kotlin:
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }

stream.forEach(::println) // b1, b2

println("Any B ${stream.any { it.startsWith('b') }}") // Any B true
println("Any C ${stream.any { it.startsWith('c') }}") // Any C false

stream.forEach(::println) // b1, b2
```

And in Java to get the same behavior:

```
// Java:
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Therefore in Kotlin the provider of the data decides if it can reset back and provide a new iterator or not. But if you want to intentionally constrain a `Sequence` to one time iteration, you can use `constrainOnce()` function for `Sequence` as follows:

```
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }
        .constrainOnce()

stream.forEach(::println) // b1, b2
stream.forEach(::println) // Error:java.lang.IllegalStateException: This sequence can be
consumed only once.
```

See also:

- API Reference for extension functions for Iterable
- API reference for extension functions for Array
- API reference for extension functions for List
- API reference for extension functions to Map

Examples

Accumulate names in a List

```
// Java:
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// Kotlin:
val list = people.map { it.name }  // toList() not needed
```

Convert elements to strings and concatenate them, separated by commas

```
// Java:
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));
```

```
// Kotlin:
val joined = things.joinToString() // ", " is used as separator, by default
```

Compute sum of salaries of employee

```
// Java:
int total = employees.stream()
                      .collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val total = employees.sumBy { it.salary }
```

Group employees by department

```
// Java:
Map<Department, List<Employee>> byDept
    = employees.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Kotlin:
val byDept = employees.groupBy { it.department }
```

Compute sum of salaries by department

```
// Java:
Map<Department, Integer> totalByDept
    = employees.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment,
                    Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val totalByDept = employees.groupBy { it.dept }.mapValues { it.value.sumBy { it.salary }}
```

Partition students into passing and failing

```
// Java:
Map<Boolean, List<Student>> passingFailing =
    students.stream()
            .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Kotlin:
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
```

Names of male members

```
// Java:
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

```
// Kotlin:
val namesOfMaleMembers = roster.filter { it.gender == Person.Sex.MALE }.map { it.name }
```

Group names of members in roster by gender

```
// Java:
Map<Person.Sex, List<String>> namesByGender =
    roster.stream().collect(
      Collectors.groupingBy(
          Person::getGender,
          Collectors.mapping(
              Person::getName,
              Collectors.toList())));
```

```
// Kotlin:
val namesByGender = roster.groupBy { it.gender }.mapValues { it.value.map { it.name } }
```

Filter a list to another list

```
// Java:
List<String> filtered = items.stream()
    .filter( item -> item.startsWith("o") )
    .collect(Collectors.toList());
```

```
// Kotlin:
val filtered = items.filter { item.startsWith('o') }
```

Finding shortest string a list

```
// Java:
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

```
// Kotlin:
val shortest = items.minBy { it.length }
```

Different Kinds of Streams #2 - lazily using first item if exists

```
// Java:
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

Different Kinds of Streams #3 - iterate a range of Integers

```
// Java:
IntStream.range(1, 4).forEach(System.out::println);
```

```
// Kotlin:  (inclusive range)
(1..3).forEach(::println)
```

Different Kinds of Streams #4 - iterate an array, map the values, calculate the average

```
// Java:
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println); // 5.0
```

```
// Kotlin:
arrayOf(1,2,3).map { 2 * it + 1}.average().apply(::println)
```

Different Kinds of Streams #5 - lazily iterate a list of strings, map the values, convert to Int, find max

```
// Java:
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

```
// Kotlin:
```

```
sequenceOf("a1", "a2", "a3")
    .map { it.substring(1) }
    .map(String::toInt)
    .max().apply(::println)
```

Different Kinds of Streams #6 - lazily iterate a stream of Ints, map the values, print results

```
// Java:
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:  (inclusive range)
(1..3).map { "a$it" }.forEach(::println)
```

Different Kinds of Streams #7 - lazily iterate Doubles, map to Int, map to String, print each

```
// Java:
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:
sequenceOf(1.0, 2.0, 3.0).map(Double::toInt).map { "a$it" }.forEach(::println)
```

Counting items in a list after filter is applied

```
// Java:
long count = items.stream().filter( item -> item.startsWith("t")).count();
```

```
// Kotlin:
val count = items.filter { it.startsWith('t') }.size
// but better to not filter, but count with a predicate
val count = items.count { it.startsWith('t') }
```

How streams work - filter, upper case, then sort a list

```
// Java:
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
```

```
myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);

// C1
// C2
```

```
// Kotlin:
val list = listOf("a1", "a2", "b1", "c2", "c1")
list.filter { it.startsWith('c') }.map (String::toUpperCase).sorted()
        .forEach (::println)
```

Different Kinds of Streams #1 - eager using first item if it exists

```
// Java:
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
listOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

or, create an extension function on String called ifPresent:

```
// Kotlin:
inline fun String?.ifPresent(thenDo: (String)->Unit) = this?.apply { thenDo(this) }

// now use the new extension function:
listOf("a1", "a2", "a3").firstOrNull().ifPresent(::println)
```

See also: `apply()` function

See also: Extension Functions

See also: `?.` Safe Call operator, and in general nullability:
http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563#34498563

Collect example #5 - find people of legal age, output formatted string

```
// Java:
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

```
// Kotlin:
val phrase = persons
        .filter { it.age >= 18 }
        .map { it.name }
        .joinToString(" and ", "In Germany ", " are of legal age.")

println(phrase)
// In Germany Max and Peter and Pamela are of legal age.
```

And as a side note, in Kotlin we can create simple data classes and instantiate the test data as follows:

```
// Kotlin:
// data class has equals, hashcode, toString, and copy methods automagically
data class Person(val name: String, val age: Int)

val persons = listOf(Person("Tod", 5), Person("Max", 33),
                    Person("Frank", 13), Person("Peter", 80),
                    Person("Pamela", 18))
```

Collect example #6 - group people by age, print age and names together

```
// Java:
Map<Integer, String> map = persons
        .stream()
        .collect(Collectors.toMap(
                p -> p.age,
                p -> p.name,
                (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

Ok, a more interest case here for Kotlin. First the wrong answers to explore variations of creating a `Map` from a collection/sequence:

```
// Kotlin:
val map1 = persons.map { it.age to it.name }.toMap()
println(map1)
// output: {18=Max, 23=Pamela, 12=David}
// Result: duplicates overridden, no exception similar to Java 8

val map2 = persons.toMap({ it.age }, { it.name })
println(map2)
// output: {18=Max, 23=Pamela, 12=David}
// Result: same as above, more verbose, duplicates overridden

val map3 = persons.toMapBy { it.age }
println(map3)
// output: {18=Person(name=Max, age=18), 23=Person(name=Pamela, age=23), 12=Person(name=David,
age=12)}
// Result: duplicates overridden again

val map4 = persons.groupBy { it.age }
println(map4)
// output: {18=[Person(name=Max, age=18)], 23=[Person(name=Peter, age=23), Person(name=Pamela,
```

```
age=23)], 12=[Person(name=David, age=12)]}
// Result: closer, but now have a Map<Int, List<Person>> instead of Map<Int, String>

val map5 = persons.groupBy { it.age }.mapValues { it.value.map { it.name } }
println(map5)
// output: {18=[Max], 23=[Peter, Pamela], 12=[David]}
// Result: closer, but now have a Map<Int, List<String>> instead of Map<Int, String>
```

And now for the correct answer:

```
// Kotlin:
val map6 = persons.groupBy { it.age }.mapValues { it.value.joinToString(";") { it.name } }

println(map6)
// output: {18=Max, 23=Peter;Pamela, 12=David}
// Result: YAY!!
```

We just needed to join the matching values to collapse the lists and provide a transformer to
`joinToString` to move from `Person` instance to the `Person.name`.

Collect example #7a - Map names, join together with delimiter

```
// Java (verbose):
Collector<Person, StringJoiner, String> personNameCollector =
Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
        .stream()
        .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID

// Java (concise)
String names = persons.stream().map(p -> p.name.toUpperCase()).collect(Collectors.joining(" |
"));
```

```
// Kotlin:
val names = persons.map { it.name.toUpperCase() }.joinToString(" | ")
```

Collect example #7b - Collect with SummarizingInt

```
// Java:
IntSummaryStatistics ageSummary =
    persons.stream()
            .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

```
// Kotlin:
```

```
// something to hold the stats...
data class SummaryStatisticsInt(var count: Int = 0,
                                var sum: Int = 0,
                                var min: Int = Int.MAX_VALUE,
                                var max: Int = Int.MIN_VALUE,
                                var avg: Double = 0.0) {
    fun accumulate(newInt: Int): SummaryStatisticsInt {
        count++
        sum += newInt
        min = min.coerceAtMost(newInt)
        max = max.coerceAtLeast(newInt)
        avg = sum.toDouble() / count
        return this
    }
}

// Now manually doing a fold, since Stream.collect is really just a fold
val stats = persons.fold(SummaryStatisticsInt()) { stats, person ->
stats.accumulate(person.age) }

println(stats)
// output: SummaryStatisticsInt(count=4, sum=76, min=12, max=23, avg=19.0)
```

But it is better to create an extension function, 2 actually to match styles in Kotlin stdlib:

```
// Kotlin:
inline fun Collection<Int>.summarizingInt(): SummaryStatisticsInt
        = this.fold(SummaryStatisticsInt()) { stats, num -> stats.accumulate(num) }

inline fun <T: Any> Collection<T>.summarizingInt(transform: (T)->Int): SummaryStatisticsInt =
        this.fold(SummaryStatisticsInt()) { stats, item -> stats.accumulate(transform(item)) }
```

Now you have two ways to use the new `summarizingInt` functions:

```
val stats2 = persons.map { it.age }.summarizingInt()

// or

val stats3 = persons.summarizingInt { it.age }
```

And all of these produce the same results. We can also create this extension to work on `Sequence` and for appropriate primitive types.

Chapter 22: JUnit

Examples

Rules

To add a JUnit rule to a test fixture:

```
@Rule @JvmField val myRule = TemporaryFolder()
```

The `@JvmField` annotation is necessary to expose the backing field with the same visibility (public) as the `myRule` property (see answer). JUnit rules require the annotated rule field to be public.

Read JUnit online: https://riptutorial.com/kotlin/topic/6973/junit

Chapter 23: Kotlin Android Extensions

Introduction

Kotlin has a built-in view injection for Android, allowing to skip manual binding or need for frameworks such as ButterKnife. Some of the advantages are a nicer syntax, better static typing and thus being less error-prone.

Examples

Configuration

Start with a [properly configured gradle project](#).

In your **project-local** (not top-level) `build.gradle` append extensions plugin declaration below your Kotlin plugin, on top-level indentation level.

```
buildscript {
    ...
}

apply plugin: "com.android.application"
...
apply plugin: "kotlin-android"
apply plugin: "kotlin-android-extensions"
...
```

Using Views

Assuming we have an activity with an example layout called `activity_main.xml`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="My button"/>
</LinearLayout>
```

We can use Kotlin extensions to call the button without any additional binding like so:

```kotlin
import kotlinx.android.synthetic.main.activity_main.my_button

class MainActivity: Activity() {
    override fun onCreate(savedInstanceBundle: Bundle?) {
        super.onCreate(savedInstanceBundle)
```

```
        setContentView(R.layout.activity_main)
        // my_button is already casted to a proper type of "Button"
        // instead of being a "View"
        my_button.setText("Kotlin rocks!")
    }
}
```

You can also import all ids appearing in layout with a `*` notation

```
// my_button can be used the same way as before
import kotlinx.android.synthetic.main.activity_main.*
```

Synthetic views can't be used outside of Activities/Fragments/Views with that layout inflated:

```
import kotlinx.android.synthetic.main.activity_main.my_button

class NotAView {
    init {
        // This sample won't compile!
        my_button.setText("Kotlin rocks!")
    }
}
```

Product flavors

Android extensions also work with multiple Android Product Flavors. For example if we have flavors in `build.gradle` like so:

```
android {
    productFlavors {
        paid {
            ...
        }
        free {
            ...
        }
    }
}
```

And for example, only the free flavor has a buy button:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/buy_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Buy full version"/>
</LinearLayout>
```

We can bind to the flavor specifically:

```
import kotlinx.android.synthetic.free.main_activity.buy_button
```

Painfull listener for getting notice, when the view is completely drawn now is so simple and awesome with Kotlin's extension

```
mView.afterMeasured {
    // inside this block the view is completely drawn
    // you can get view's height/width, it.height / it.width
}
```

Under the hood

```
inline fun View.afterMeasured(crossinline f: View.() -> Unit) {
viewTreeObserver.addOnGlobalLayoutListener(object : ViewTreeObserver.OnGlobalLayoutListener {
    override fun onGlobalLayout() {
        if (measuredHeight > 0 && measuredWidth > 0) {
            viewTreeObserver.removeOnGlobalLayoutListener(this)
            f()
        }
    }
})
}
```

Read Kotlin Android Extensions online: https://riptutorial.com/kotlin/topic/9474/kotlin-android-extensions

Chapter 24: Kotlin Caveats

Examples

Calling a toString() on a nullable type

A thing to look out for when using the `toString` method in Kotlin is the handling of null in combination with the `String?`.

For example you want to get text from an `EditText` in Android.

You would have a piece of code like:

```
// Incorrect:
val text = view.textField?.text.toString() ?: ""
```

You would expect that if the field did not exists the value would be empty string but in this case it is `"null"`.

```
// Correct:
val text = view.textField?.text?.toString() ?: ""
```

Read Kotlin Caveats online: https://riptutorial.com/kotlin/topic/6608/kotlin-caveats

---

Chapter 25: Kotlin for Java Developers

Introduction

Most people coming to Kotlin do have a programming background in Java.

This topic collects examples comparing Java to Kotlin, highlighting the most important differences and those gems Kotlin offers over Java.

Examples

Declaring Variables

In Kotlin, variable declarations look a bit different than Java's:

```
val i : Int = 42
```

- They start with either `val` or `var`, making the declaration `final` ("**val**ue") or **var**iable.

- The type is noted after the name, separated by a `:`

- Thanks to Kotlin's *type inference* the explicit type declaration can be obmitted if there is an assignment with a type the compiler is able to unambigously detect

Java	Kotlin
`int i = 42;`	`var i = 42` (*or* `var i : Int = 42`)
`final int i = 42;`	`val i = 42`

Quick Facts

- Kotlin does not need `;` to end statements
- Kotlin is **null-safe**
- Kotlin is **100% Java interoperable**
- Kotlin has **no primitives** (but optimizes their object counterparts for the JVM, if possible)
- Kotlin classes have **properties, not fields**
- Kotlin offers **data classes** with auto-generated `equals`/`hashCode` methods and field accessors
- Kotlin only has runtime Exceptions, **no checked Exceptions**
- Kotlin has **no `new` keyword**. Creating objects is done just by calling the constructor like any other method.
- Kotlin supports (limited) **operator overloading**. For example, accessing a value of a map can be written like: `val a = someMap["key"]`
- Kotlin can not only be compiled to byte code for the JVM, but also into **Java Script**, enabling you to write both backend and frontend code in Kotlin

- Kotlin is **fully compatible with Java 6**, which is especially interesting in regards for support of (not so) old Android devices
- Kotlin is an **officially supported** language **for Android development**
- Kotlin's collections have built-in disctinction between **mutable and immutable collections**.
- Kotlin supports **Coroutines** (experimental)

Equality & Identity

Kotlin uses `==` for equality (that is, calls `equals` internally) and `===` for referential identity.

Java	Kotlin
`a.equals(b);`	`a == b`
`a == b;`	`a === b`
`a != b;`	`a !== b`

See: https://kotlinlang.org/docs/reference/equality.html

IF, TRY and others are expressions, not statements

In Kotlin, `if`, `try` and others are expressions (so they do return a value) rather than (void) statements.

So, for example, Kotlin does not have Java's ternary *Elvis Operator*, but you can write something like this:

```
val i = if (someBoolean) 33 else 42
```

Even more unfamiliar, but equally expressive, is the `try` *expression*:

```
val i = try {
    Integer.parseInt(someString)
}
catch (ex : Exception)
{
    42
}
```

Chapter 26: logging in kotlin

Remarks

Related question: Idiomatic way of logging in Kotlin

Examples

kotlin.logging

```
class FooWithLogging {
  companion object: KLogging()

  fun bar() {
    logger.info { "hello $name" }
  }

  fun logException(e: Exception) {
    logger.error(e) { "Error occured" }
  }
}
```

Using kotlin.logging framework

Read logging in kotlin online: https://riptutorial.com/kotlin/topic/3258/logging-in-kotlin

Chapter 27: Loops in Kotlin

Remarks

In Kotlin, loops are compiled down to optimized loops wherever possible. For example, if you iterate over a number range, the bytecode will be compiled down to a corresponding loop based on plain int values to avoid the overhead of object creation.

Examples

Repeat an action x times

```
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}
```

Looping over iterables

You can loop over any iterable by using the standard for-loop:

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

Lots of things in Kotlin are iterable, like number ranges:

```
for(i in 0..9) {
    print(i)
}
```

If you need an index while iterating:

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

There is also a functional approach to iterating included in the standard library, without apparent language constructs, using the forEach function:

```
iterable.forEach {
    print(it.toString())
}
```

`it` in this example implicitly holds the current element, see Lambda Functions

---

While Loops

While and do-while loops work like they do in other languages:

```
while(condition) {
    doSomething()
}

do {
    doSomething()
} while (condition)
```

In the do-while loop, the condition block has access to values and variables declared in the loop body.

Break and continue

Break and continue keywords work like they do in other languages.

```
while(true) {
    if(condition1) {
        continue // Will immediately start the next iteration, without executing the rest of
the loop body
    }
    if(condition2) {
        break // Will exit the loop completely
    }
}
```

If you have nested loops, you can label the loop statements and qualify the break and continue statements to specify which loop you want to continue or break:

```
outer@ for(i in 0..10) {
    inner@ for(j in 0..10) {
        break       // Will break the inner loop
        break@inner // Will break the inner loop
        break@outer // Will break the outer loop
    }
}
```

This approach won't work for the functional `forEach` construct, though.

Iterating over a Map in kotlin

```
//iterates over a map, getting the key and value at once

var map = hashMapOf(1 to "foo", 2 to "bar", 3 to "baz")

for ((key, value) in map) {
    println("Map[$key] = $value")
}
```

Recursion

Looping via recursion is also possible in Kotlin as in most programming languages.

```
fun factorial(n: Long): Long = if (n == 0) 1 else n * factorial(n - 1)

println(factorial(10)) // 3628800
```

In the example above, the `factorial` function will be called repeatedly by itself until the given condition is met.

Functional constructs for iteration

The Kotlin Standard Library also provides numerous useful functions to iteratively work upon collections.

For example, the `map` function can be used to transform a list of items.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.map { "Number $it" }
```

One of the many advantages of this style is it allows to chain operations in a similar fashion. Only a minor modification would be required if say, the list above were needed to be filtered for even numbers. The `filter` function can be used.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.filter { it % 2 == 0 }.map { "Number $it" }
```

Read Loops in Kotlin online: https://riptutorial.com/kotlin/topic/2727/loops-in-kotlin

Chapter 28: Null Safety

Examples

Nullable and Non-Nullable types

Normal types, like `String`, are not nullable. To make them able to hold null values, you have to explicitly denote that by putting a `?` behind them: `String?`

```
var string        : String = "Hello World!"
var nullableString: String? = null

string = nullableString    // Compiler error: Can't assign nullable to non-nullable type.
nullableString = string    // This will work however!
```

Safe call operator

To access functions and properties of nullable types, you have to use special operators.

The first one, `?.`, gives you the property or function you're trying to access, or it gives you null if the object is null:

```
val string: String? = "Hello World!"
print(string.length)   // Compile error: Can't directly access property of nullable type.
print(string?.length)  // Will print the string's length, or "null" if the string is null.
```

Idiom: calling multiple methods on the same, null-checked object

An elegant way to call multiple methods of a null-checked object is using Kotlin's `apply` like this:

```
obj?.apply {
    foo()
    bar()
}
```

This will call `foo` and `bar` on `obj` (which is `this` in the `apply` block) only if `obj` is non-null, skipping the entire block otherwise.

To bring a nullable variable into scope as a non-nullable reference without making it the implicit receiver of function and property calls, you can use `let` instead of `apply`:

```
nullable?.let { notnull ->
    notnull.foo()
    notnull.bar()
}
```

`notnull` could be named anything, or even left out and used through the implicit lambda parameter `it`.

Smart casts

If the compiler can infer that an object can't be null at a certain point, you don't have to use the special operators anymore:

```
var string: String? = "Hello!"
print(string.length)  // Compile error
if(string != null) {
    // The compiler now knows that string can't be null
    print(string.length)  // It works now!
}
```

> **Note:** The compiler won't allow you to smart cast mutable variables that could potentially be modified between the null-check and the intended usage.
>
> If a variable is accessible from outside the scope of the current block (because they are members of a non-local object, for example), you need to create a new, local reference which you can then smart cast and use.

Eliminate nulls from an Iterable and array

Sometimes we need to change type from `Collection<T?>` to `Collections<T>`. In that case, `filterNotNull` is our solution.

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Null Coalescing / Elvis Operator

Sometimes it is desirable to evaluate a nullable expression in an if-else fashion. The elvis operator, `?:`, can be used in Kotlin for such a situation.

For instance:

```
val value: String = data?.first() ?: "Nothing here."
```

The expression above returns `"Nothing here"` if `data?.first()` or `data` itself yield a `null` value else the result of `data?.first()`.

It is also possible to throw exceptions using the same syntax to abort code execution.

```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```

> Reminder: NullPointerExceptions can be thrown using the assertion operator (e.g. `data!!.second()!!`)

Assertion

`!!` suffixes ignore nullability and returns a non-null version of that type. `KotlinNullPointerException` will be thrown if the object is a `null`.

```
val message: String? = null
println(message!!) //KotlinNullPointerException thrown, app crashes
```

Elvis Operator (?:)

In Kotlin, we can declare variable which can hold `null reference`. Suppose we have a nullable reference `a`, we can say "if `a` is not null, use it, otherwise use some non-null value `x`"

```
var a: String? = "Nullable String Value"
```

Now, `a` can be null. So when we need to access value of `a`, then we need to perform safety check, whether it contains value or not. We can perform this safety check by conventional `if...else` statement.

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

But here comes advance operator `Elvis`(Operator Elvis : `?:`). Above `if...else` can be expressed with the Elvis operator as below:

```
val b = a?.length ?: -1
```

If the expression to the left of `?:` (here : `a?.length`) is not null, the elvis operator returns it, otherwise it returns the expression to the right (here: `-1`). Right-hand side expression is evaluated only if the left-hand side is null.

Chapter 29: Ranges

Introduction

Range expressions are formed with rangeTo functions that have the operator form .. which is complemented by in and !in. Range is defined for any comparable type, but for integral primitive types it has an optimized implementation

Examples

Integral Type Ranges

Integral type ranges (IntRange , LongRange , CharRange) have an extra feature: they can be iterated over. The compiler takes care of converting this analogously to Java's indexed for-loop, without extra overhead

```
for (i in 1..4) print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing
```

downTo() function

if you want to iterate over numbers in reverse order? It's simple. You can use the downTo() function defined in the standard library

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

step() function

Is it possible to iterate over numbers with arbitrary step, not equal to 1? Sure, the step() function will help you

```
for (i in 1..4 step 2) print(i) // prints "13"
for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

until function

To create a range which does not include its end element, you can use the until function:

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
println(i)
}
```

Read Ranges online: https://riptutorial.com/kotlin/topic/10121/ranges

---

Chapter 30: RecyclerView in Kotlin

Introduction

I just want to share my little bit knowledge and code of RecyclerView using Kotlin.

Examples

Main class and Adapter

I am assuming that you have aware about the some syntax of **Kotlin** and how to use, just add
RecyclerView in **activity_main.xml** file and set with adapter class.

```
class MainActivity : AppCompatActivity(){

    lateinit var mRecyclerView : RecyclerView
    val mAdapter : RecyclerAdapter = RecyclerAdapter()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toolbar = findViewById(R.id.toolbar) as Toolbar
        setSupportActionBar(toolbar)

        mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

        mRecyclerView.setHasFixedSize(true)
        mRecyclerView.layoutManager = LinearLayoutManager(this)
        mAdapter.RecyclerAdapter(getList(), this)
        mRecyclerView.adapter = mAdapter
    }

    private fun getList(): ArrayList<String> {
        var list : ArrayList<String> = ArrayList()
        for  (i in 1..10) { // equivalent of 1 <= i && i <= 10
            println(i)
            list.add("$i")
        }
        return list
    }
}
```

this one is your recycler view **adapter** class and create **main_item.xml** file what you want

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    var mItems: ArrayList<String>  = ArrayList()
    lateinit var mClick : OnClick

    fun RecyclerAdapter(item : ArrayList<String>, mClick : OnClick){
        this.mItems = item
        this.mClick = mClick;
    }
```

```
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = mItems[position]
        holder.bind(item, mClick, position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.main_item, parent, false))
    }

    override fun getItemCount(): Int {
        return mItems.size
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val card = view.findViewById(R.id.card) as TextView
        fun bind(str: String, mClick: OnClick, position: Int){
            card.text = str
            card.setOnClickListener { view ->
                mClick.onClickListner(position)
            }
        }
    }
}
```

Read RecyclerView in Kotlin online: https://riptutorial.com/kotlin/topic/10143/recyclerview-in-kotlin

Chapter 31: Reflection

Introduction

Reflection is a language's ability to inspect code at runtime instead of compile time.

Remarks

Reflection is a mechanism to introspect language constructs (classes and functions) at the runtime.

When targeting JVM platform, runtime reflection features are distributed in separate JAR: `kotlin-reflect.jar`. This is done to reduce runtime size, cut unused features and make it possible to target another (like JS) platforms.

Examples

Referencing a class

To obtain a reference to a `KClass` object representing some class use double colons:

```
val c1 = String::class
val c2 = MyClass::class
```

Referencing a function

Functions are first-class citizens in Kotlin. You can obtain a reference on it using double colons and then pass it to another function:

```
fun isPositive(x: Int) = x > 0

val numbers = listOf(-2, -1, 0, 1, 2)
println(numbers.filter(::isPositive)) // [1, 2]
```

Inter-operating with Java reflection

To obtain a Java's `Class` object from Kotlin's `KClass` use the `.java` extension property:

```
val stringKClass: KClass<String> = String::class
val c1: Class<String> = stringKClass.java

val c2: Class<MyClass> = MyClass::class.java
```

The latter example will be optimized by the compiler to not allocate an intermediate `KClass` instance.

Getting values of all properties of a class

Given `Example` class extending `BaseExample` class with some properties:

```
open class BaseExample(val baseField: String)

class Example(val field1: String, val field2: Int, baseField: String):
    BaseExample(baseField) {

    val field3: String
        get() = "Property without backing field"

    val field4 by lazy { "Delegated value" }

    private val privateField: String = "Private value"
}
```

One can get hold of all properties of a class:

```
val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Running this code will cause an exception to be thrown. Property `private val privateField` is declared private and calling `member.get(example)` on it will not succeed. One way to handle this it to filter out private properties. To do that we have to check the visibility modifier of a property's Java getter. In case of `private val` the getter does not exist so we can assume private access.

The helper function and it's usage might look like this:

```
fun isFieldAccessible(property: KProperty1<*, *>): Boolean {
    return property.javaGetter?.modifiers?.let { !Modifier.isPrivate(it) } ?: false
}

val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.filter { isFieldAccessible(it) }.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Another approach is to make private properties accessible using reflection:

```
example::class.memberProperties.forEach { member ->
    member.isAccessible = true
    println("${member.name} -> ${member.get(example)}")
}
```

Setting values of all properties of a class

As an example we want to set all string properties of a sample class

```
class TestClass {
    val readOnlyProperty: String
        get() = "Read only!"

    var readWriteString = "asd"
    var readWriteInt = 23

    var readWriteBackedStringProperty: String = ""
        get() = field + '5'
        set(value) { field = value + '5' }

    var readWriteBackedIntProperty: Int = 0
        get() = field + 1
        set(value) { field = value - 1 }

    var delegatedProperty: Int by TestDelegate()

    private var privateProperty = "This should be private"

    private class TestDelegate {
        private var backingField = 3

        operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int {
            return backingField
        }

        operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: Int) {
            backingField += value
        }
    }
}
```

Getting mutable properties builds on getting all properties, filtering mutable properties by type. We also need to check visibility, as reading private properties results in run time exception.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            System.out.println("${prop.name} -> ${prop.get(instance)}")
        }
```

To set all `String` properties to `"Our Value"` we can additionally filter by the return type. Since Kotlin is based on Java VM, Type Erasure is in effect, and thus Properties returning generic types such as `List<String>` will be the same as `List<Any>`. Sadly reflection is not a golden bullet and there is no sensible way to avoid this, so you need to watch out in your use-cases.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        // We only want strings
        .filter{ it.returnType.isSubtypeOf(String::class.starProjectedType) }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            // Instead of printing the property we set it to some value
            prop.setter.call(instance, "Our Value")
```

```
        }
```

Read Reflection online: https://riptutorial.com/kotlin/topic/2402/reflection

Chapter 32: Regex

Examples

Idioms for Regex Matching in When Expression

Using immutable locals:

Uses less horizontal space but more vertical space than the "anonymous temporaries" template. Preferable over the "anonymous temporaries" template if the `when` expression is in a loop--in that case, regex definitions should be placed outside the loop.

```
import kotlin.text.regex

var string = /* some string */

val regex1 = Regex( /* pattern */ )
val regex2 = Regex( /* pattern */ )
/* etc */

when {
    regex1.matches(string) -> /* do stuff */
    regex2.matches(string) -> /* do stuff */
    /* etc */
}
```

Using anonymous temporaries:

Uses less vertical space but more horizontal space than the "immutable locals" template. Should not be used if then `when` expression is in a loop.

```
import kotlin.text.regex

var string = /* some string */

when {
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    /* etc */
}
```

Using the visitor pattern:

Has the benefit of closely emulating the "argument-ful" `when` syntax. This is beneficial because it more clearly indicates the argument of the `when` expression, and also precludes certain programmer mistakes that could arise from having to repeat the `when` argument in every `whenEntry`.

Either the "immutable locals" or the "anonymous temporaries" template may be used with this implementation the visitor pattern.

```
import kotlin.text.regex

var string = /* some string */

when (RegexWhenArgument(string)) {
    Regex( /* pattern */ ) -> /* do stuff */
    Regex( /* pattern */ ) -> /* do stuff */
    /* etc */
}
```

And the minimal definition of the wrapper class for the `when` expression argument:

```
class RegexWhenArgument (val whenArgument: CharSequence) {
    operator fun equals(whenEntry: Regex) = whenEntry.matches(whenArgument)
    override operator fun equals(whenEntry: Any?) = (whenArgument == whenEntry)
}
```

Introduction to regular expressions in Kotlin

This post shows how to use most of the functions in the `Regex` class, work with null safely related to the `Regex` functions, and how raw strings makes it easier to write and read regex patterns.

The RegEx class

To work with regular expressions in Kotlin, you need to use the `Regex(pattern: String)` class and invoke functions like `find(..)` or `replace(..)` on that regex object.

An example on how to use the `Regex` class that returns true if the `input` string contains c or d:

```
val regex = Regex(pattern = "c|d")
val matched = regex.containsMatchIn(input = "abc")    // matched: true
```

The essential thing to understand with all the `Regex` functions is that the result is based on matching the regex `pattern` and the `input` string. Some of the functions requires a full match, while the rest requires only a partial match. The `containsMatchIn(..)` function used in the example requires a partial match and is explained later in this post.

Null safety with regular expressions

Both `find(..)` and `matchEntire(..)` will return a `MatchResult?` object. The `?` character after `MatchResult` is necessary for Kotlin to handle null safely.

An example that demonstrates how Kotlin handles null safely from a `Regex` function, when the `find(..)` function returns null:

```
val matchResult =
    Regex("c|d").find("efg")          // matchResult: null
val a = matchResult?.value           // a: null
val b = matchResult?.value.orEmpty() // b: ""
a?.toUpperCase()                     // Still needs question mark. => null
b.toUpperCase()                      // Accesses the function directly. => ""
```

With the `orEmpty()` function, `b` can't be null and the `?` character is unnecessary when you call functions on `b`.

If you don't care about this safe handling of null values, Kotlin allows you to work with null values like in Java with the `!!` characters:

```
a!!.toUpperCase()                    // => KotlinNullPointerException
```

Raw strings in regex patterns

Kotlin provides an improvement over Java with a raw string that makes it possible to write pure regex patterns without double backslashes, that are necessary with a Java string. A raw string is represented with a triple quote:

```
"""\d{3}-\d{3}-\d{4}""" // raw Kotlin string
"\\d{3}-\\d{3}-\\d{4}"  // standard Java string
```

find(input: CharSequence, startIndex: Int): MatchResult?

The `input` string will be matched against the `pattern` in the `Regex` object. It returns a `Matchresult?` object with the first matched text after the `startIndex`, or `null` if the pattern didn't match the `input` string. The result string is retrieved from the `MatchResult?` object's `value` property. The `startIndex` parameter is optional with the default value 0.

To extract the first valid phone number from a string with contact details:

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

With no valid phone number in the `input` string, the variable `phoneNumber` will be `null`.

findAll(input: CharSequence, startIndex: Int): Sequence

Returns all the matches from the `input` string that matches the regex `pattern`.

To print out all numbers separated with space, from a text with letters and digits:

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

The `matchedResults` variable is a sequence with `MatchResult` objects. With an `input` string without digits, the `findAll(..)` function will return an empty sequence.

matchEntire(input: CharSequence): MatchResult?

If all the characters in the `input` string matches the regex `pattern`, a string equal to the `input` will be returned. Else, `null` will be returned.

Returns the input string if the whole input string is a number:

```
val a = Regex("""\d+""").matchEntire("100")?.value          // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value  // b: null
```

matches(input: CharSequence): Boolean

Returns true if the whole input string matches the regex pattern. False otherwise.

Tests if two strings contains only digits:

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")              // => true
regex.matches(input = "50 dollars")      // => false
```

containsMatchIn(input: CharSequence): Boolean

Returns true if part of the input string matches the regex pattern. False otherwise.

Test if two strings contains at least one digit:

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
Regex("""\d+""").containsMatchIn("Fifty dollars")    // => false
```

split(input: CharSequence, limit: Int): List

Returns a new list without all the regex matches.

To return lists without digits:

```
val a = Regex("""\d+""").split("ab12cd34ef")     // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

There is one element in the list for each split. The first `input` string has three numbers. That results in a list with three elements.

replace(input: CharSequence, replacement: String): String

Replaces all matches of the regex `pattern` in the `input` string with the replacement string.

To replace all digits in a string with an x:

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // result: abxcdxef
```

Read Regex online: https://riptutorial.com/kotlin/topic/8364/regex

Chapter 33: Singleton objects

Introduction

An *object* is a special kind of class, which can be declared using `object` keyword. Objects are similar to Singletons (a design pattern) in java. It also functions as the static part of java. Beginners who are switching from java to kotlin can vastly use this feature, in place of static, or singletons.

Examples

Use as repalcement of static methods/fields of java

```
object CommonUtils {

    var anyname: String ="Hello"

    fun dispMsg(message: String) {
        println(message)
    }
}
```

From any other class, just invoke the variable and functions in this way:

```
CommonUtils.anyname
CommonUtils.dispMsg("like static call")
```

Use as a singleton

Kotlin objects are actually just singletons. Its primary advantage is that you don't have to use `SomeSingleton.INSTANCE` to get the instance of the singleton.

In java your singleton looks like this:

```
public enum SharedRegistry {
    INSTANCE;
    public void register(String key, Object thing) {}
}

public static void main(String[] args) {
    SharedRegistry.INSTANCE.register("a", "apple");
    SharedRegistry.INSTANCE.register("b", "boy");
    SharedRegistry.INSTANCE.register("c", "cat");
    SharedRegistry.INSTANCE.register("d", "dog");
}
```

In kotlin, the equivalent code is

```
object SharedRegistry {
    fun register(key: String, thing: Object) {}
}

fun main(Array<String> args) {
    SharedRegistry.register("a", "apple")
    SharedRegistry.register("b", "boy")
    SharedRegistry.register("c", "cat")
    SharedRegistry.register("d", "dog")
}
```

It's obvoiusly less verbose to use.

Read Singleton objects online: https://riptutorial.com/kotlin/topic/10152/singleton-objects

Chapter 34: Strings

Examples

Elements of String

Elements of String are characters that can be accessed by the indexing operation `string[index]`.

```
val str = "Hello, World!"
println(str[1]) // Prints e
```

String elements can be iterated with a for-loop.

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

String Literals

Kotlin has two types of string literals:

- Escaped string
- Raw string

Escaped string handles special characters by escaping them. Escaping is done with a backslash. The following escape sequences are supported: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` and `\$`. To encode any other character, use the Unicode escape sequence syntax: `\uFF00`.

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

Raw string delimited by a triple quote `"""`, contains no escaping and can contain newlines and any other characters

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

Leading whitespace can be removed with trimMargin() function.

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

Default margin prefix is pipe character `|`, this can be set as a parameter to trimMargin; e.g. `trimMargin(">")`.

String Templates

Both escaped strings and raw strings can contain template expressions. Template expression is a piece of code which is evaluated and its result is concatenated into string. It starts with a dollar sign `$` and consists of either a variable 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"
```

To include a literal dollar sign in a string, escape it using a backslash:

```
val str = "\$foo" // evaluates to "$foo"
```

The exception is raw strings, which do not support escaping. In raw strings you can use the following syntax to represent a dollar sign.

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

String Equality

In Kotlin strings are compared with `==` operator which chect for their structural equality.

```
val str1 = "Hello, World!"
val str2 = "Hello," + " World!"
println(str1 == str2) // Prints true
```

Referential equality is checked with `===` operator.

```
val str1 = """
    |Hello, World!
    """.trimMargin()

val str2 = """
    #Hello, World!
    """.trimMargin("#")

val str3 = str1

println(str1 == str2) // Prints true
println(str1 === str2) // Prints false
println(str1 === str3) // Prints true
```

Read Strings online: https://riptutorial.com/kotlin/topic/8285/strings

Chapter 35: Type aliases

Introduction

With type aliases, we can give a alias to other type. It's ideal for giving a name to function types like `(String) -> Boolean` or generic type like `Pair<Person, Person>`.

Type aliases support generics. An alias can replace a type with generics and an alias can be generics.

Syntax

- **typealias** *alias-name* **=** *existing-type*

Remarks

Type aliases is a feature of the compiler. Nothing is added in the generated code for the JVM. All aliases will be replaced by the real type.

Examples

Function type

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

Generic type

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

Read Type aliases online: https://riptutorial.com/kotlin/topic/9453/type-aliases

Chapter 36: Type-Safe Builders

Remarks

A *type-safe builder* is a concept, rather than a language feature, so it is not strictly formalized.

A typical structure of a type-safe builder

A single builder function usually consists of 3 steps:

1. Create an object.
2. Execute lambda to initialize the object.
3. Add the object to structure or return it.

Type-safe builders in Kotlin libraries

The concept of type-safe builders is widely used in some Kotlin libraries and frameworks, eg.:

- Anko
- Wasabi
- Ktor
- Spec

Examples

Type-safe tree structure builder

Builders can be defined as a set of extension functions taking lambda expressions with receivers as arguments. In this example, a menu of a `JFrame` is being built:

```kotlin
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
    menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
    menu.init()
    add(menu)
}

fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
    menuItem.init()
    add(menuItem)
```

```
    }
```

These functions can then be used to build a tree structure of objects in an easy way:

```
class MyFrame : JFrame() {
    init {
        menuBar {
            menu("Menu1") {
                menuItem("Item1") {
                    // Initialize MenuItem with some Action
                }
                menuItem("Item2") {}
            }
            menu("Menu2") {
                menuItem("Item3") {}
                menuItem("Item4") {}
            }
        }
    }
}
```

Read Type-Safe Builders online: https://riptutorial.com/kotlin/topic/6010/type-safe-builders

Chapter 37: Vararg Parameters in Functions

Syntax

- **Vararg Keyword**: `vararg` is used in a method declaration to indicate that a variable number of parameters will be accepted.
- **Spread Operator**: An asterisk (`*`) before an array that is used in function calls to "unfold" the contents into individual parameters.

Examples

Basics: Using the vararg keyword

Define the function using the `vararg` keyword.

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

Now you can pass as many parameters (of the correct type) into the function as you want.

```
printNumbers(0, 1)              // Prints "0" "1"
printNumbers(10, 20, 30, 500)   // Prints "10" "20" "30" "500"
```

Notes: Vararg parameters *must* be the last parameter in the parameter list.

Spread Operator: Passing arrays into vararg functions

Arrays can be passed into vararg functions using the **Spread Operator**, `*`.

Assuming the following function exists...

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

You can **pass an array** into the function like so...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(*numbers)

// This is the same as passing in (1, 2, 3)
```

The spread operator can also be used **in the middle** of the parameters...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(10, 20, *numbers, 30, 40)

// This is the same as passing in (10, 20, 1, 2, 3, 30, 40)
```

Read Vararg Parameters in Functions online: https://riptutorial.com/kotlin/topic/5835/vararg-parameters-in-functions

Chapter 38: Visibility Modifiers

Introduction

In Kotlin, there are 4 types of visibility modifiers are available.

Public: This can be accessed from anywhere.

Private: This can only be accessed from the module code.

Protected: This can only be accessed from the class defining it and any derived classes.

Internal: This can only be accessed from the scope of the class defining it.

Syntax

* `<visibility modifier> val/var <variable name> = <value>`

Examples

Code Sample

Public: `public val name = "Avijit"`

Private: `private val name = "Avijit"`

Protected: `protected val name = "Avijit"`

Internal: `internal val name = "Avijit"`

Read Visibility Modifiers online: https://riptutorial.com/kotlin/topic/10019/visibility-modifiers

---

Credits

S. No	Chapters	Contributors
1	Getting started with Kotlin	babedev, Community, cyberscientist, ganesshkumar, Ihor Kucherenko, Jayson Minard, mnoronha, neworld, Parker Hoyes, Ruckus T-Boom, Sach, Sean Reilly, Sheigutn, Simón Oroño, UnKnown, Urko Pineda
2	Annotations	Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
3	Arrays	egor.zhdan, Sam, UnKnown
4	Basic Lambdas	memoizr, Rich Kuzsma
5	Basics of Kotlin	Shinoo Goyal
6	Class Delegation	Sam
7	Class Inheritance	byxor, KeksArmee, piotrek1543, Slav
8	Collections	Ascension
9	Conditional Statements	Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin, Spidfire
10	Configuring Kotlin build	Aaron Christiansen, elect, madhead
11	coroutines	Jemo Mgebrishvili
12	Delegated properties	Sam, Seaskyways
13	DSL Building	Dmitriy L, ice1000
14	Enum	David Soroko, Kirill Rakhman, SerCe
15	Exceptions	Brad Larson, jereksel, Sapan Zaveri
16	Extension Methods	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
17	Functions	Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire
18	Generics	hotkey, Jayson Minard, KeksArmee
19	Idioms	Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson

		Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
20	Interfaces	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
21	Java 8 Stream Equivalents	Brad, Gerson, Jayson Minard, Piero Divasto, Sam
22	JUnit	jenglert
23	Kotlin Android Extensions	Jemo Mgebrishvili, Ritave
24	Kotlin Caveats	Grigory Konushev, Spidfire
25	Kotlin for Java Developers	Thorsten Schleinzer
26	logging in kotlin	Konrad Jamrozik, olivierlemasle, oshai
27	Loops in Kotlin	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
28	Null Safety	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
29	Ranges	Nihal Saxena
30	RecyclerView in Kotlin	Mohit Suthar
31	Reflection	atok, Kirill Rakhman, madhead, Ritave, Sup
32	Regex	Espen, Travis
33	Singleton objects	Divya, glee8e
34	Strings	Januson, Sam
35	Type aliases	Kevin Robatel
36	Type-Safe Builders	Slav
37	Vararg Parameters in Functions	byxor, piotrek1543, Sam
38	Visibility Modifiers	Avijit Karmakar