MEM LNC

Kotlin

The
Ultimate Beginner's Guide to Learn kotlin
Programming Step by Step

• • •

2ND
EDITION

MOAML MOHMMED

kotlin

The Ultimate Beginner's Guide to Learn kotlin Programming Step by Step

2020

Moaml mohmmed

2nd edition

Mem lnc

Introduction

Kotlin is the new lovechild of the JVM developers' world.

Google promoted Kotlin as a first class language on its Java-based Android platform back in May. Since then, the whole development world has been wondering: what is this language? Kotlin has been around for a few years and has been running on production systems, after the languages 1.0 release in February 2016, for a year or so. The language has received a lot of praise and loving words from the developer community. It is a breath of fresh air, a good upgrade to systems running older versions of Java, and still somehow an old dog in a familiar playing field.

What is Kotlin? What does it bring that the JVM doesn't already have?

Kotlin vs. Java

There are a few approaches we can take when introducing Kotlin. We can discuss it through Java, the language Kotlin needs to be based on due to its JVM runtime, or we can do it through Scala, the language Kotlin is heavily influenced by. There is no doubt that Kotlin is better than Java. It is much safer and more concise. It provides you with a bunch of additions to your standard Java language and enhances a few bits and pieces that Java developers have grown to dislike. Additions include things like null safety, extension functions, data classes, objects, first class functions as well as extensive and expressive lambdas. Kotlin also enhances Java’s type inference and type system and takes massive leaps forward with collections.

Kotlin vs. Scala

Perhaps, it’s better to compare Kotlin against Scala. This comparison might scare some of you quite a bit because Scala has the reputation of being simultaneously intriguing and frightening. It heavily introduces functional programming paradigm to you while still mixing it into familiar object orientation (hence in an awfully lot of cases creating a mishmash of advanced techniques from both paradigms), brings in some new build tools, and gives your internal flow state a frustrating break every now and then due to long compile times.

I come bearing both good news and bad news. Let’s start with the bad news:

Bad news is that Kotlin is similar to Scala, it follows the same path as Scala does

The good news: luckily, it’s only slightly similar to Scala in every aspect.

Kotlin & Functional Programming Paradigm

The functional programming paradigm is big part of Kotlin as well. Luckily, it doesn't go into the higher-kinded types, monadic do-continuations, or advanced type theory concepts that make you seek out Bartosz Milewski and his brilliant book on Category Theory. Kotlin introduces easy-to-use collection manipulation functions and functional pipelines for you. You will get your maps, filters, and folds, which in most cases are enough to get to the functional programming path.

Java devs that have been lucky enough to jump into Java 8 (hugs and kisses to you Android and/or enterprise developers) will be familiar with the these basics and will feel right at home when they jump into Kotlin. They will also find conciseness and safety of better type system, which will spark their first crush towards the language. It is just so pretty and seamless to pipe these functions together and build a clean pipeline. And when you come back to it after a few weeks, you’ll still feel like you can somewhat understand it. Smiles all around.

Java and Kotlin

(You can skip reading this section if you are a beginner)

The question that may come to Java developers' minds is "What should I learn now?" There are a range of languages that deserve consideration, such as Clojure, Rust or Haskell. But what if you want to learn something that helps you pay bills and is so easy to use? Kotlin may be your best choice, and we will try in this article to explain why.

Java and Kotlin and difference between them

A programming language developed by JetBrains, who were behind the IDEA idea as well as other things.

A simple and flexible alternative to Java

Matches well with the existing Java code

Translate to Java bytecode

Works on JVM

It also translates to JavaScript

If you have read its documents, you will notice a number of important things:

Allows you to do a lot of things with a few codes

Many of the problems in Java are solved

Help you continue to use the usual ecosystem of Java

Allows you to program the front and back interface in the same language

100% compatible with Java

They perform well compared to alternatives (Clojure, Scala)

Add only a thin layer of complexity to Java

Sounds good, does not it? Here are some examples to compare with Java.

Elements of values versus data of items

What you see here is an old Java object (POJO) with the usual patterns:

```
public class HexagonValueObject {



    private final int x;
    private final int y;
    private final int z;



    public HexagonValueObject (int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
```

```
public int getX () {
    return x;
}


public int getY () {
    return y;
}


public int getZ () {
    return z;
}


@Override
public boolean equals (Object o) {
    if (this == o) return true;
    if (o = null | getClass ()! = o.getClass ()) return false;
    HexagonValueObject hexagon = (HexagonValueObject) o;
    return getX () == hexagon.getX () &&
            getY () == hexagon.getY () &&
            getZ () == hexagon.getZ ();
}
```

```
    @Override
    public int hashCode () {
        return Objects.hash (getX (), getY (), getZ ());
    }



    @Override
    public String toString () {
        return "HexagonValueObject {" +
                "x =" + x +
                ", y =" + y +
                ", z =" + z +
                '}';
   }
```

Creating value elements is a cumbersome process even using libraries like Lombok) Lombok requires an attached installation in your IDE environment to work, which may not be possible in all development environments. This problem can be overcome with tools such as Delombok, Solution to the problem), at least IDEA (or Eclipse) gives you some help in generating a lot of those functions, but adding a field and forgetting the adjustment of the equals function will result in bad surprises. Let's now look at the corresponding code in Kotlin:

```
data class HexagonDataClass (val x: Int, val y: Int, val z: Int)
```

just awesome! We've shortened a lot of writing compared to the Java version. Data items in Kotlin give you equals + hashCode and

toString Plus

Outcomes and setters. You can also copy them, creating a new object with some rewriting fields.

Stratification of text strings String interpolation

Dealing with text strings in Java is cumbersome. But can be simplified using String.format however will remain ugly.

```
public class JavaUser {
    private final String name;
    private final int age;



    public String toHumanReadableFormat () {
        return "JavaUser {" +
                "name = '" + name +' \ " +
                ", age =" + age + "
                '}';
    }



    public String toHumanReadableFormatWithStringFormat () {
        return String.format ("JavaUser {name = '% s', age =% s}", name, age);
    }
}
```

Kotlin found a solution to this, adding the concept of string filler, which extended the use of variables in literal strings. It was also possible to call jobs from them!

```
class KotlinUser (val name: String, val age: Int) {



    fun toHumanReadableFormat () = "JavaUser {name = '$ name', age = $
age}"



    fun toHumanReadableFormatWithMethodCall () =
        "JavaUser {name = '$ {name.capitalize ()}', age = $ age}"
}
```

Extension functions

Writing servers in Java decorators can be difficult, as they are not ideal.

If you want to write a library, which can be used with all the categories that List offers, you can not simply use it in your server because it will need you to provide many other functions, so you will have to extend the AbstractList.

```
public class ListPresenterDecorator <T> extends AbstractList <T> {

    private List <T> list;

    public ListPresenterDecorator (List <T> list) {
        this.list = list;
    }
    public String present () {
        return list.stream ()
                .map (Object :: toString)
                .collect (Collectors.joining (","));
    }

    @Override
    public T get (int index) {
        return list.get (index);
    }

    @Override
    public int size () {
        return list.size ();
    }
}
```

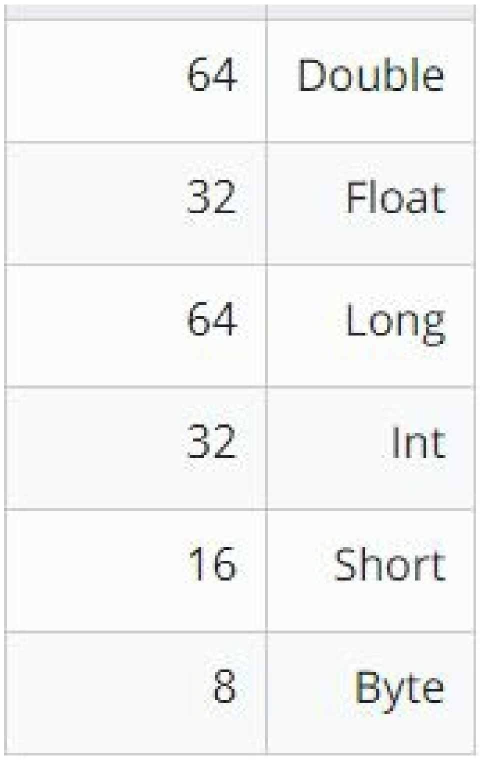
Chapter II
Data Types in Kotlin

Every element in Kotlin is an object, since member functions and properties can be called over any variable, and some have their own internal representation; for example, boolean is represented as basic values during runtime) But for the user just ordinary varieties. This page discusses the main types of data in Kotlin: numbers, chars, boolean, arrays, and text strings.

Numbers

Kotlin uses numerical data in a similar way to Java but with simple differences. For example, Kotlin does not support implicit (from the smallest to widening) and the literals differ in some cases.

Kotlin provides the following basic types of expression (similar to Java):



64	Double
32	Float
64	Long
32	Int
16	Short
8	Byte

The characters are not numbers in Kotlin.

Directal Constants

It has the following types of integers:

Decimals: 123 The long decimal numbers are characterized by the letter L (in its large case only) to form: 123L

* Hexadecimal numbers such as 0x0F

* Binaries such as 0b00001011

Kotlin does not support eight octal values, but it supports the conventional notation of floating-point numbers:

* Double (default) such as: 123.5 and 123.5e10

* Float type (f or F) such as: 123.5f

* Underscore in Numeric Literals starting with version 1.1

Use the underscore to make it easier to read large numbers such as:

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

Representation

The numbers are physically stored in the Java platform as basic types in the JVM unless there is a nullable reference to the number (eg Int?) Or the generics is not included, then the boxed numbers are not necessarily preserved and this does not necessarily preserve their identity, In the code:

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

But it maintains the equal value (with the symbol ==):

```
val a: Int = 10000
print (a == a) // will print true
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print (boxedA == anotherBoxedA) // also will print true
```

Explicit Conversion

The smaller species in Kotlin are not subtypes of larger species due to the different methods of representing numerical data among them, but they will also cause the following problems:

```
// This is a fake code and will not work correctly
val a: Int? = 1 // Envelope integer (java.lang.Integer)
val b: Long? = a // An implicit conversion that results in an envelope type type Long (java.lang.Long)
print (a == b) // This will result in a false result
```

The reason for the appearance of a false result in the last line of the previous code is that the equals function realizes that the other variable is also of the Long type. This is not true. Thus, we have not only lost identity, but also equality, And therefore a Byte value, for example, can not be assigned to a value of Int type without declaring that conversion, such as:

```
val b: Byte = 1 // No problem here
val i: Int = b // False attribution
```

By adding the explicit conversion, the code becomes:

```
val i: Int = b.toInt () // An explicit conversion of the smallest type to the largest type
```

Any of the following conversions can be used:

toByte (): Convert to Byte type

toShort (): Convert to Short type

toInt (): Converts to type Int

toLong (): Convert to Long type

toFloat (): Convert to Float type

toDouble (): Convert to Double Type

toChar (): Convert to Char type

The absence of implicit conversions is not a problem; because the type guesses through the context, as well as the overloading of arithmetical operations in line with the most appropriate conversion, such as:

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

Operations

Kotlin supports a set of standard calculations on numbers that are defined as members in the appropriate categories. The compiler complies with the instructions according to the approved instructions. (See [Operator overloading])

In bitwise operations, there are no special characters, but only functions that can be invoked in the internal form, such as:

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

Below is a list of binary operations at the box level (available for Int and Long types only)

* shl (bits) offset with a left-hand signal (such as a parameter >> in Java)

* shr (bits) offset with a right-hand signal (such as the << operator in Java)

* ushr (bits) The offset without a right-hand signal (such as the parameter <<< in Java)

* AND (bits) AND logical operation at the binary cell level

* Or (bits) OR logical operation at the binary cell level

* xor (bits) The logical operation XOR at the binary cell level

* inv () to reverse the case of binary cells (NOT)

Comparison of Floating Point Numbers

This paragraph discusses comparisons (between floating-point numbers):

Check the equality of a == b and not equal a! = B

Comparison coefficients a <b and a> b and a <b> and a> = b

Represent and validate fields such as a..b, x in a..b, and x! In a..b

If both operands at the ends of a (a and b) of a Float type, or a double or the equivalent of a nullable type (or the type declared as a result of conversion) [[Kotlin / typecasts | smart cast]]) On numbers and fields shall be in accordance with the IEEE 754 standards in calculations of floating-point numbers.

To support generic use cases and to provide total organization when transactions on both ends of the lab are not defined as floating-point numbers (such as Any, Comparable <...>, or the [type parameter]), operations depend on equals and compareTo For the two types Float and Double, this does not comply with standard standards, thus:

NaN is equal to the same

NaN is larger than any other element that contains POSITIVE_INFINITY

0.0) with a negative value (smaller than 0.0) in positive value)

Characters

The type in which the characters are stored is Char and can not be handled directly as in the numerical data, as in the following code:

```
fun check (c: Char) {
   if (c == 1) {// The error here is the incompatibility between the two types
       // ...
```

```
    }
}
```

The character is always surrounded by two semicolons such as the value '1', and some of the characters become especially meaningful if preceded by the symbol \ in order to make it among the characters escape sequence The Kotlin language supports the special characters: \ t, \ b, \ n and \ r, \, \ and \ $, and to express the Unicode code to any other character using the \ u character such as '\ uFF00'. The character can be converted to int integer value Explicitly, as shown in the following code:

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

The boxed character - as in numerical values - may be locked when a nullable reference is required, as this encapsulation does not preserve identity.

Booleans

The Boolean data type expresses logical data that has one of the following values: true or false.

The boolean can be encapsulated when a nullable reference is needed.

Core operations available for binary data:

- Separation process

- Connection &&

- The process of exile!

Arrays

A matrix in the Kotlin language is represented by the Array class, which contains the get and set functions (which are replaced by the arcades [] via overload overloading) plus the property of the length of the array size and many other functions Like:

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

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

The arrayOf () function is used to create a new matrix if the values of its elements are passed as arguments for this function. For example, to create the matrix [3, 2, 1] the function call is: arrayOf (1, 2, 3) (Whose values are null) along a specified length (size) that uses the arrayOfNulls () function. The second way to create a matrix is by relying on the constructor in the Array class, which requires the size of the array and the function that returns the initial value of each element in the matrix through its index,

```
val asc = Array (5, {i -> (i * i) .toString ()})
```

This code creates a matrix of text strings: [16 "," 9 "," 4 "," 1 "," 0 "]. As mentioned earlier, the use of brackets [] is only to invoke the functions get () and set () .

Unlike the Java language, the matrices in the Kotlin language are invariant. This means that Array <String> is not allowed to Array <Any>, which in turn prevents any failure during execution, but can use Array <out Any>

Kotlin has assigned some items to represent primitives without boxing, such as ByteArray, ShortArray, IntArray, and so on. There is no inheritance relationship for these Array items, but they contain the same set of properties and properties, each with a factory function, for example the use of matrices:

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

Text strings (Strings)

String is a text string that is immutable and contains characters that are accessible through the index (eg index), such as s [i], and a for loop with the characters in the string can be used as follows:

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

Text strings values

There are two types: raw text strings, which may include new line navigation, and escaped text strings that are very similar to text strings in Java, such as:

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

If the next line goes through the "n" character and to see all of those characters, you can refer to the Character paragraph on the same page as the current one. The raw text strings have "" three-point quotes and do not contain any escape characters, but can include a transition to a new line such as:

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

It is easy to remove the help spaces (left side of each new line) by calling the trimMargin () function such as:

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

The symbol is used The default case as an alias prefix, and any other character can be chosen and passed as a parameter for the parameter such as: trimMargin (">").

Text Template Templates

Text strings sometimes contain some custom expressions (which are parts of the code that add value to the text string), starting with $ and consists of:

* Either a simple name like:

```
val i = 10
val s = "i = $ i" // will be calculated as "i = 10"
```

* Or an expression in brackets {} such as:

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

These templates are used in both textual strings, and when the $ character is needed as a direct value in the raw string (not to denote an expression template), the following formula can be adopted:

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

"" "

Using the Kotlin language in server application development

Kotlin is a powerful tool in the development of server applications. The codes are accurate and expressive, and have full compatibility with Java and are easy to learn.

Features of using Kotlin language to develop server applications

Expressiveness: The advantages of the innovative Kotlin language (such as its support for the type-safe builder and delegated properties) help to create easy-to-use abstractions.

Scalability: Kotlin supports the help of coroutines, which allows - in server applications - the ability to expand to a larger number of clients and simple hardware requirements.

Interoperability: Kotlin is fully compatible with the Java operating environment, making it possible to take advantage of the techniques we used to use while using a more modern language.

Easy to move from Java: Kotlin provides a step-by-step transition from Java to complex code. It is possible to write new code and keep other parts of the Java environment.

Tools: The Kotlin language in addition to its integrated development environment (IDE) provides custom tools (as in the case of the Spring environment, for example) through the addition of the IntelliJ IDEA Ultimate environment.

Learning Curve: Starting in Kotlin will be very easy for the Java developer, and the convertor will help you get started with the Kotlin (Kotlin Plugins) See a number of interactive exercises.

Work environments available to develop server applications using the Kotlin language

Spring uses the benefits of Kotlin to get more precise API APIs (starting with version 5.0), and the Project Generator allows you to create a new project in Kotlin.

The Vert.x (used to build interactive web applications based on JVM) provides dedicated support for the Kotlin language.

The Ktor environment is an original Kotlin operating environment and is available online from JetBrains. It allows for extensive expansion based on routines (in Kotlin) and provides an easy-to-use software interface.

Kotlinx.html is a domain-specific language (DSL) that is used to build HTML pages in web applications. It offers many alternatives to traditional templating systems such as JSP and FreeMarker.

The options available (in the case of stored information) include direct access to JDBC and JPA interfaces, as well as the use of NoSQL databases in Java. JPA uses the addition of Kotlin-JPA in the compiler with some adapt Classes to suit the requirements of the working environment.

Deployment of server applications in Kotlin

Apps are published in Kotlin within any host that supports Java Web applications including AWS (Amazon Web Services), Google Cloud Platform and many other services. Applications can be deployed in Kotlin via the Heroku platform. AWS Labs provides a model project Use the Kotlin language to write AWS Lambda functions.

Kotlin users on the server side

The open and distributed Corda platform, which is used for records and supported by major banks, is built entirely through Kotlin. JetBrains Account, which is responsible for all licensing and verification sales at JetBrains, is written in Kotlin as well, and has been in service since 2015 without major problems.

Chapter III

the conditions

As in any programming language, Kotlin contains expressions to control flow: if, when, loop, and while. It also supports the words continue and break used in loops (see returns and jump).

Expression if

In the Kotlin language, the if condition is an expression that returns a value, and therefore does not need the formula condition? then: else because if does this role as in the following code:

```
Normal use
var max = a
if (a <b) max = b

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

// Use the condition as an expression
val max = if (a> b) a else b
```

If several instructions can be grouped in a block so that the value of that part is the value of the last expression in it, such as:

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

When you use if as an expression rather than an instruction (that is, it may return a value or assign its value to a variable), this expression must have an else condition.

Expression when

The expression "when" is an alternative to the switch, found in programming languages such as C and its equivalent, with the following formula:

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

A comparison is made between the value between the brackets and the values in all cases and in the sequence in which the condition of equivalence is achieved in one of them. The use of the condition is then either as a statement or as an expression; if used as an expression, The last case is equal, but if an instruction is irrelevant to the values of any existing situation, as in if, several instructions can be grouped under the same block and the value of that part is then the value of the last expression in it.

The format of the program is transferred to the case else if none of the above conditions are met, and when the condition is used as an expression, the existence of the else condition is mandatory or may be optional if the compiler can ensure that the existing cases meet all possible conditions.

It is noted that if the instructions are in more than one case, the two cases can be merged together and separated by a comma, such as:

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

The use of expression can be used in situations when instead of fixed values, such as:

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

Or it can be a domain or collection using the in or! In, such as:

```
when (x) {
    in 1.10 -> print ("x is in the range")
    in validNumbers -> print ("x is valid")
    ! in 10..20 -> print ("x is outside the range")
    else -> print ("none of the above")
}
```

It is also possible to take advantage of the is or deny! Is with a type, and there is no need for further verification to access the methods or properties. This is due to the smartcasts, such as:

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

When the string is substituted by the if-else if clause, and if there is no argument between the brackets, the conditions of the case will be boolean expressions and the case instructions whose binary value is true , Like:

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

Expression for

The for loop is used with any set of iterations, which is similar to the foreach loop in programming languages such as C # and for the following formula:

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

Several instructions can be implemented with one frequency, such as:

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

The loop for having something that makes iterator such as:

A function of a member function or an extension function, such as iterator (), which has a re-type:

Contains a child function of the type or function next ()

Contains a child function of the class or an additional function that hasNext () which returns a boolean value,

These three functions are defined as operators.

When you use the for loop with the matrices to become an index-based interface, you do not need to create an iteration object. The loop is repeated through the index such as:

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

Or withIndex may call from the library to become the code as follows:

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

Expression while

Both while and do..while as in any other programming language, such as:

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

do {
    val y = retrieveData ()
} while (y! = null) // The variable can be accessed here
```

Chapter III

Categories and objects

Classes and Inheritance in Kotlin

Use the class keyword to declare the category as follows (Invoice name):

```
class Invoice {
}
```

The label contains the name of the class name, the class header, and the class structure enclosed by the brackets. There is no need for brackets, such as:

```
class Empty
```

Albany (Constructor)

Each type in Kotlin has one primary and one or more-secondary. The main constructor is part of the header header where it is added immediately after the name of the product (and the parameters may be added) :

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

If the master builder does not have an annotation or a visibility modifier, then delete the constructor keyword, such as:

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

It is also noted that the main constructor contains no code because the initialization code is written in initializer blocks preceding the init keyword. These parts are executed in the same order as The class is interlaced with property initializers, such as:

```
class InitOrderDemo (name: String) {
    val firstProperty = "First property: $ name" .also (:: println)

    init {
        println ("First initializer block that prints $ {name}")
    }
    val secondProperty = "Second property: $ {name.length}". Also (:: println)

    init {
        println ("Second initializer block that prints $ {name.length}")
    }
}
```

Basic parameters can be used in the initialization parts and can also be used to initialize properties declared in the class structure, such as:

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

Kotlin supports a concise version of the declaration and initial configuration of the basic constructs:

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

As with properties in general, properties in the underlying constructor may be defined as variable mutable or read-only properties. If there is an annotation or a modifier for the constructor, the constructor must be present. The description or delimiters should be preceded by the formula:

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

Secondary Constructors

The secondary constructor in the class is declared to begin with the constructor, such as:

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

If the class contains a primary constructor, each secondary bane will eventually lead to it either directly or indirectly via another secondary bane. Either way, the main constructor will have access to this keyword, such as:

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

Note that the code in the initializer blocks becomes part of the primary construct where the main constructor is the first instruction in the secondary constructor, so all the codes in the initialization parts will be executed before the secondary constructor, even if there is no master element of the class, The transition will be implicit and the configuration parts will also be executed, such as:

```
class Constructors {
    init {
        println ("Init block")
    }

    constructor (i: Int) {
        println ("Constructor")
    }
}
```

If the non-abstract class does not contain any syntax (primary or secondary), it will automatically generate a default without arguments and the access parameter is of the public type. If you want to not be generic, Key blank and appropriately delineated, such as:

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

In the JVM environment, it is noted that if each constructor parameter has default values, the compiler will generate an additional parameter without parameters, which will use the default values. This makes the Kotlin language easy to use with libraries such as Jackson or JPA Which create objects using the pots without transactions. Like:

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

Create Instances of the class

To create an object of a class, the constructor is called as a normal function, such as:

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

Kotlin does not rely on the existence of the keyword new as in Java.

To create objects from nested or internal or anonymous inner.

Item (Class Members)

Category may include:

* Protections and initializer blocks

* Functions

* Properties

* Nested and internal classes (nested and inner classes)

* Object declarations

* Inheritance

All classes in Kotlin share the Any category as the default superclass for any category for which no higher category is declared, such as:

class Example // Inheritance of Any

Note: Any is not java.lang.Object; it contains only three functions: equals (), hashCode () and toString (). To declare the topclass clearly, type after the symbol: in the class header, such as:

open class Base (p: Int)

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

The keyword open (in front of the product name) expresses the final meaning of the final word in Java, which allows (open) the process of inheritance of this class because the default status of the items in the Kotlin language is final. If the derived class contains a basic element, the base class must be initialized there based on the parameters of that constructor. If the class is not master, each secondary element must create the base type using the keyword super, or to transfer that task to another, so it is possible to call several secondary colors of many of my basic type, such as:

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

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

Define Dependents (Overriding Methods)

Unlike the Java language, Kotlin requires a clear declaration that the element can be redefined by adding the open keyword before it and by adding the override keyword when redefining it, such as:

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

It is necessary to add override to the Derived.v function. If you do not add a compilation error, if you do not add the open keyword in the Base.nv () function, it is not allowed to redefine it using override or Without them, because access to the members in the categories of final type (without open) is not allowed, and the element is preceded by the keyword override of the open type by the default state (that is, it can also be redefined in subclasses) To prevent this, add the final keyword as in the code:

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

Overriding Properties

The redefinition of properties is similar to the method of redefining the methods in the preceding paragraph; the override must be preceded by the attributes that are redefined in the derived class and originally found in the superclass provided that the two properties are computable, , And redefined either by using the initializer or through get (), such as:

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

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

It is possible to redefine the property of the type val as a var property, but to prevent the inverse state. This is because the property of type val is originally declared as a getter, redefined as a var type, and also declares a setter in the derived class. The keyword override may be used as part of the definition of the property in the primary constructor, such as:

```
interface Foo {
    val count: Int
}

class Bar1 (override val count: Int): Foo

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

Derived class initialization order

When creating derived class objects, the initial class configurations will be executed first (preceded by only the arguments in the underlying construct), and will therefore occur before any configuration in the derived class, such as:

```
open class Base (val name: String) {

    init {println ("Initializing Base")}

    open val size: Int =
    name.length.also {println ("Initializing size in Base: $ it")}
}

class Derived (
    name: String,
    val lastName: String
): Base (name.capitalize (). Also {println ("Argument for Base: $ it"))} {

    init {println ("Initializing Derived")}

    override val size: Int =
    (super.size + lastName.length) .also {println ("Initializing size in Derived:
```

```
$ it")}
}
```

This means that the defined or overridden properties of the derived class are not prepared at the time the constructor processes are performed in the base class, and therefore a runtime failure may occur if none of these characteristics in the class Derivative is used in the configuration of the base class either directly or indirectly (via the definition of the implementation of any other element of the type of overtrid). To avoid this, elements of the open type must not be defined in the base class in any of the Or any of the configuration parts Arafa tagged with the word init.

Call the definition of usage of the top class (superclass)

The code in the derived class can invoke functions of the higher class and access functions by using the super keyword, such as:

```
open class Foo {
    open fun f () {println ("Foo.f ()")}
    open val x: Int get () = 1
}

class Bar: Foo () {
    override fun f () {
        super.f ()
        println ("Bar.f ()")
    }

    override val x: Int get () = super.x + 1
}
```

In order to reach the topclass of the outer class of the inner category, the keyword super is used as the super @ outer as in the following code:

```
class Bar: Foo () {
    override fun f () {/ * ... * /}
    override val x: Int get () = 0
    inner class Baz {
        fun g () {
            super@Bar.f () // Call the usage definition of the function f ()
            println (super@Bar.x) // Use the usage definition for the function to
access the x property
        }
    }
}
```

Overriding Rules

The inheritance of Kotlin's implementation inheritance is governed by the following rule:

"If a class inherits several definitions of implementation from the same member of the immediate superclasses, it must contain an override of that element to define its use (perhaps using one of the definitions it inherits) ", And the super keyword is constrained by the name of the supertype and is enclosed in parentheses <> in the form: super <Base> to refer to the higher type from which the usage definition is used,

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

interface B {
    fun f () {print ("B")} // The interface elements are in the default state open
    fun b () {print ("b")}
}
```

```
class C (): A (), B {
    // The compiler requires a redefinition of the function f ()
    override fun f () {
        super <A> .f () // call A.f ()
        super <B> .f () // call B.f ()
    }
}
```

There is no problem in inheriting the functions a and b of each of classes A and B (respectively) because category C inherits only one definition of each of these functions, and for function (f), category C is defined for use It should therefore contain an override of the function f () to define its use to eliminate that confusion.

Abstract Categories (Abstract Classes)

The class or some of its members may be declared by the abstract keyword so that these elements are not defined for implementation within the category in which there is no need to add the open keyword to the class or abstract elements.

It is possible to redefine an open element that is not abstract in another element, such as:

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

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

Companion Objects

There are no static methods in Kotlin - unlike Java and C # - and replace them with package-level functions.

When a callable function is needed without creating an instance of the class but having access to what is inside the class (for example, the factory method) it can be made as a member of the object's own declaration within that class.

In more detail, if the associated object is declared within the class, then its elements can be invoked in the same format as the static call in Java and C # depending on the class name only.

3.2 Properties and Fields fields in the Kotlin language

Declaring properties

Items in Kotlin may contain attributes defined either as variable values via the var keyword or as read-only values via the val keyword, such as:

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

The properties can be accessed by their name (as if they were a field in Java), such as:

```
fun copyAddress (address: Address): Address {
   val result = Address () // does not exist here for the keyword new
   result.name = address.name // The access functions are called,
   result.street = address.street
   // ...
   return result
}
```

Access properties using Getter and Setter

The full wording of the declaration of characteristics is:

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

The existence of both the setter and the setter is optional, and the existence of the type is also optional if it can be determined by the initialization (or the return type in the getter function) as in the following examples:

```
var allByDefault: Int? // Problem occurs: Configuration must be clearly present, and access functions
// implicit default
var initialized = 1 // The type is the type of integers and default access functions
```

The full version of the declaration of read-only characteristics is different from its predecessor in two ways: first it begins with the word val instead of var, and second that there is no function of the setter, which is as follows:

```
val simple: Int? // For the type of integer type, you must perform the
```

initialization in the constructor and the default access function getter

```
val inferredType = 1 // has an integer type and a default access function getter
```

Access functions, such as the process of modifying any other function, can be customized in the property definition, such as the getter assignment in the following code:

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

The setting of the setter function is as in the code:

```
var stringRepresentation: String
    get () = this.toString ()
    set (value) {
        setDataFromString (value) // Convert the text string and assign the value to another property
    }
```

The value-formula expresses the parameter name in the setter and can be modified. Starting from version 1.1, the property type can be deleted if it can be determined by a getter such as:

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

Sometimes any access functions can be defined without specifying their body structure when they need to change their visibility or add annotation without changing the default implementation definition, The following code:

```
var setterVisibility: String = "abc"
    private set // The access specifier is of a special type and has a default usage definition
```

```
var setterWithAnnotation: Any? = null
    @Inject set // Add the profile to it as Inject
```

Fields (Backing Fields)

The fields are not directly defined in the Kotlin language classes. Therefore, if an auxiliary field is needed, Kotlin is provided automatically, referred to in the field access functions, such as:

```
var counter = 0 // The value is also assigned to the helper field directly
    set (value) {
        if (value> = 0) field = value
    }
```

The use of the field selector is only available in the accessors for that property. If you use the default usage definition for access functions even once or if it is indicated by the field identifier when an access function is assigned, there is no helper field in the following code:

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

Backing Properties

In doing what is contrary to the implied help field mechanism (described in the previous paragraph), use the auxiliary properties as follows:

```
private var _table: Map <String, Int>? = null
```

```
public val table: Map <String, Int>
    get () {
        if (_table == null) {
            _table = HashMap () // Specifies the type parameters automatically
        }
        return _table?: throw AssertionError ("Set to null by another thread")
    }
```

This is similar in all respects to what is supported in the Java language, because access to properties of private type using accessorization functions is designed so that it can not be overridden by any call to functions.

Compile-Time Constants

The properties defined as known during the translation process are defined as "translation constants" using the const constants and have the following requirements:

To be a member of an object or a top-level event,

Set to String or any other primitive type

There is no custom getter

Such features are used in annotations such as:

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

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

Later-Initializing of Properties and Variables

Properties are defined as a natural state of type with no null values in the constructor, but this is not always appropriate. For example, it can be created through a dependency injection or a setup method to test the structure test, and a non-null configuration can not be relied upon in the constructor, although it is desirable to avoid verifying blank values when referring to properties within the class structure.

To address this situation, the subsequent configuration depends on the lateinit parameter such as:

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

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

   @Test fun test () {
      subject.method () // Direct indexing
   }
}
```

This parameter is used with var attributes defined within the class structure (not in the primary constructor and only when there is no allocation in the accessors), and since Kotlin 1.2 is also used for top-level properties, And for local variables, since the type must be primitive and non-null.

An attempt to access the property before its subsequent configuration results in a special exception that states that the property has been accessed before it is configured.

Check the subsequent configuration of variable var (starting with version 1.2)

Use isInitialized with reference The property specified by the lateinit var identifiers to verify the completion of the subsequent initialization process, such as:

```
if (foo :: bar.isInitialized) {
    println (foo.bar)
}
```

This allows verification only when access to this property is available, that is, a knowledge of the same type, an outer type, or a top-level in the same file.

Delegated Properties

The most common type of properties is readable (and perhaps written in) the helper field. In the case of access or setter, the use of the property can be defined. There are specific patterns for how the property works, such as lazy values value, reading from the map through the key, access to the database, notification when trying to access, etc. These cases can be defined as libraries using delegated properties.

Visibility Modifiers in the Kotlin language

Accessibility settings specify access to each of the classes, objects, interfaces, functions, properties, and setters (because the access parameter is similar to the access to the property itself), and there are four types Of the parameters are: private, protected, internal, public, and the default type is public (used when no one is authorized). The following is an explanation of how the determinants apply to different types of declarations.

Packages

Functions, properties, classes, objects, and interfaces can be defined at the top-level, directly within the package, such as:

```
package foo

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

It notes the following:

- If you do not specify Visible access, it is public by default, which means that the item will be available anywhere.

- Specifying the private type will make the element visible within the file that contains its permission only.

- Selecting the internal type will make the element visible within the module itself.

- The protected type is not available in top-level statements.

NOTE: To use any top-level visual declaration in another package, you can use the import command to obtain it.

Example:

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

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

public var bar: Int = 5 // The property is visible everywhere
    private set // This access point is only available in the example.kt file

internal val baz = 6 // Visible within the limits of the module (module)
```

Classes and Interfaces

When declaring the elements defined within the categories, one of the following cases:

- Private: means that the item is visible only within the category (including the other items in it).

- Protected access: such as visible private and added that the element is also visible in the subclasses of this category.

- Internal access: Any client that can access the class containing the statements (within this module) can also see the internal elements in it.

- public access: Any client that has access to the class that contains the statements can also see the public elements in it.

Note for Java programmers: The command is different in Kotlin since external classes can not access any of the private elements in their internal classes.

When redefining a specific item with protected access without explicitly specifying its access point, the specified type will also be protected.

Example:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4 // General by default
    protected class Nested {
        public val e: Int = 5
    }
}

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

    override val b = 5 // of the protected type
}

class Unrelated (o: Outer) {
    // oaa, o.b is not visible
    // o.c, o.d are visible because they are in the same unit
    // Outer.Nested is not visible
    // Nested :: e is also invisible
}
```

Constructors

The following formula is used to identify the basic constructor in the class (note that there is no constructor):

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

The type of constructor in the previous code is private because of the clear statement that the default state of the pan is public, making it available wherever the class is visible (ie, the constructor in the class is internal) In the same module [module] only).

Local Declarations

Visible access to local statements, functions, or items can not be determined.

Modules

The internal access point means that this element is visible in the same unit, and in more precise terms, the unit is a set of files written in Kotlin that are compiled together, such as:

IntelliJ IDEA Module

Project Maven

Source set in Gradle (except that the source group can access the internal statements in main)

The compilation of files is compiled through a single call to an Ant task.

Extensions in Kotlin

Extensions (Extensions)

As in programming languages such as C # and Gosu, Kotlin provides the ability to add classes to new functions without resorting to inheritances or using any design patterns such as Secorator, through special statements called extensions). Kotlin supports extension functions and extension properties.

Extension Functions

To define an additional function, its name must precede the receiver type, ie the type to be added. In the following code, the swap function is added to MableList <Int> as:

```
fun MutableList <Int> .swap (index1: Int, index2: Int) {
    val tmp = this [index1] // The list is represented by this keyword
    this [index1] = this [index2]
    this [index2] = tmp
}
```

This keyword is inside the add-in function of the receiving object (pre-point). After the addition, it can be called from any MableList <Int> such as:

```
val l = mutableListOf (1, 2, 3)
l.swap (0, 2)
```

In the previous code, this keyword, within the swap () function, expresses the variable l, and for the generalization of that function, the code becomes:

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

The generic type is declared before the function name to become available in the receiver type expression. (See generic functions)

Extensions are resolved statically

Additions do not actually modify the classes to which they apply, as new elements do not enter the category, but add some callable functions via the point format. In the variants of this type.

The additional functions are statistically distinct, ie, they are not virtual of the type of receiver. This means that the additive function determines the type of expression that is called, rather than the type of result it produces during execution, such as:

```
open class C


class D: C ()


fun C.foo () = "c"


fun D.foo () = "d"


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


printFoo (D ())
```

This code will show the output: "c" because the added function called depends on the type defined only for parameter c, which is class C. If the class contains two functions of the same name and type of receiver and accept the same arguments (the first one is a function of the member function) and the other extension function, the priority will be the original function in the member function, as in the following code:

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


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

Calling the c.foo () function from any C-type C object will show "member" rather than "extension". But overloading is allowed on the original function in the member function by the add function if they have the same name but with two different definitions (in terms of the number of transactions or their types), such as:

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


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

The call (c). Foo (1) will then display the phrase "extension".

The future is Nullable

Extensions may define a nullable type (that is, the type may contain a null value), such additions are called via object variable even if the value is null, because it can be verified using this condition: == null , This explains calling the toString () function in the Kotlin language without null checking because the check occurs in the add-in function as:

```
fun Any? .toString (): String {
    if (this == null) return "null"
    return toString ()
}
```

The type automatically converts this after the verification of this == null to the non-null type, so the toString () function will lead to the function in the class Any.

Additional Features

Kotlin also supports additional features, as well as extensibility functions, as shown in the following code:

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

Because additions do not include any new elements in classes, there is no effective way to create a backing field for the additional property. Therefore, it does not allow for the configuration of additional properties, and explicitly defines the behavior through the getter and setter,

```
val Foo.bar = 1 // This instruction is false because it does not allow for
```

additional properties

Companion Object Extensions

When a class contains an object, it can then define additional functions and properties for that object, such as:

```
class MyClass {
    companion object {} // "Companion"
}
```

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

It is called by using the name of the product as an identifier (as in any of the accompanying objects):

```
MyClass.foo ()
```

Extension areas (Scope of Extensions)

Top-level add-ons are often defined immediately after the package is declared, as in the following code:

```
package foo.bar


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

To use such an add-on outside the defined package, you must import it to the place of the call, such as:

```
package com.example.usage
import foo.bar.goo // import all extensions by the name "goo"
                // 
import foo.bar. * // Import all that exists in "foo.bar"


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

Declare extensions as members (Declaring Extensions as Members)

Kotlin allows the possibility of declaring additions to one of the items in another, in which case there are several implicit receptors (non-qualifier), and the object of the class containing the additional statements is called the dispatch receiver) An object that is the receiver type of the extension method is called the extension receiver, such as:

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

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

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

    fun caller (d: D) {
        d.foo () // calls the add-in function
    }
}
```

When there is a discrepancy between the receiver receiver and the receiver receiver, the priority is for the additional receiver, but to access the separate receiver elements, use this restricted keyword, such as:

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

If additions are known to be open, they can be redefined in subclasses. This means that the separation of these functions is virtual for the separate but static receiver type for the additional receiver type, such as:

```
open class D {
}
```

```
class D1: D () {
}

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

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

    fun caller (d: D) {
        d.foo () // call the add-in function
    }
}

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

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

C () .caller (D ()) // The phrase "D.foo in C"
C1 () .caller (D ()) // The statement "D.foo in C1"
                  // The chapter is the default for the separate future
C () .caller (D1 ()) // The phrase "D.foo in C"
                  The static chapter for the additional future
```

Motivations

In Java, many types of "Utils" are used, such as FileUtils, StringUtils, and the like. The java.util.Collections category is the same, but the negative aspect of using these utils is that Its codes are like:

```
// Java code
Collections.wap (list, Collections.binarySearch (list, Collections.max
(otherList)), Collections.max (list));
```

The names of these categories impede the programming process, and if the programmer uses static import, it will receive the code as:

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

It looks better than its predecessor, but it does not have the feature to complete the code (as we like) in the IDE environment. It would be better if we could type it:

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

However, it is difficult to define the implementation of all possible methods within the List class, and here is the importance of the additions in Kotlin.

Object Declarations and Expressions in the Kotlin language

Sometimes you may need to create an object by making minor modifications to a class without declaring a subclass. Java uses such cases to rely on anonymous inner classes and generalizes them by introducing a concept Authorization of objects and their expressions.

Object Expressions

To create an object of an anonymous type inheriting one or more types, the code is:

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

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

If the superclass has a constructor, it must pass appropriate parameters, and many supertypes are defined as a list (separated by a comma), written after the two vertical points: as in the following code:

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

```
interface B {...}


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

When needed only for an object without the complexities of the higher species, the code simply becomes as follows:

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

Anonymous objects may be used as types in the case of local or private statements only. If the object is used as a wild type in a generic function or as a general property type, the actual type of that function or property will be the higher type For the anonymous object, or Any type if no higher type is known, and the added elements in the anonymous object will not be accessible as in the following code:

```
class C {
    // The function is therefore special, and the returned type is the anonymous object type
    private fun foo () = object {
        val x: String = "x"
    }
    // The generic function is therefore the re-type is Any
    fun publicFoo () = object {
        val x: String = "x"
    }

    fun bar () {
        val x1 = foo (). x // The code is correct and works
        val x2 = publicFoo (). x // Wrong code since it can not be accessed via
reference x
    }
}
```

As with anonymous inner classes in Java, the object expression code can access all variables in the same enclosing scope (this is not limited to the final variables as in Java), such as the following code:

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

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

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

Object Declarations

The Singelton design model is used in many cases, and Kotlin (after Scala) makes the Singelton statement as easy as the code:

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

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

This is called an object declaration, and is always followed by a label after the object. As in the declaration of variables, this statement is not an expression and can not be used on the right side of the assignment statement. Initialization is problem free The thread-safe (ie does not produce software problems in the cross-configuration across different threads) and the object is referred to by using its name directly, such as:

```
DataProviderManager.registerDataProvider (...)
```

This type of organism may have higher types (supertypes) such as:

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

Note: Objects can not be localized (as if they are in a function), but can be nested in statements about other objects or any non-internal items.

Companion Objects

Add the keyword companion before declaring the object if it is in the category, such as:

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

The elements of the accompanying organism are called through the name of the species as a qualifier, such as:

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

The name of the accompanying object can also be dispensed with to use the name Companion, such as:

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

Although the elements of the accompanying object are similar to the static elements in other programming languages, they remain normal object elements during runtime, so they can define the implementation of interfaces, such as:

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



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

However, JVM associated object elements can be generated as real static methods and fields using the @JvmStatic annotation.

The differences between the statement of objects and their expressions

There is one fundamental difference between them:

* Execute (and initialize) object expressions in the place where you are using it immediately

* The setting is lazy to declare objects when they are accessed for the first time

* Companion object is created when loading the corresponding class, similar to the static configuration in Java.

Delegated Properties in Kotlin

Use generalized features

In Kotlin you can define the use of properties manually over and over again every time you need them, but it is easier to define their use once and store this definition in the library for use whenever needed. This includes:

Lazy property: It is calculated once only when it is first accessed.

* Observable property: a waiter is called when a change occurs in the property.

* Store the properties in the map instead of a separate field for each.

Kotlin includes all such cases by supporting the delegated properties in the following general format:

```
val / var <property name>: <Type> by <expression>
```

As in the following code:

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

The expression after the keyword will be generalized because the get () and set () functions will be used for the getValue () and setValue () functions, so there is no need to define the implementation of any interface, It is sufficient to have the getValue () function and the setValue () function in the case of variables of the var type, such as the following code:

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

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

When reading the value from p and generalizing any instance of Delegate, the getValue () function is called from Delegate where the first parameter is the object from which the value of p is read and the second contains a description (for example, for its name)

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

This will show the result:

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

Similarly, the setValue () function is called when assigning to variable p, where the first and second coefficients are identical while the third contains the assigned value, when the following code is executed:

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

The result will appear:

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

The requirements for such a circular will be explained in a subsequent paragraph on the current page.

Note: From Kotlin 1.1, you can declare the generalized property within the function or part of the code block and do not require that it be a member in the class. You will find an example under the heading localized features on this page.

Standard Delegates

The standard Kotlin Library contains a number of factory methods for the various types of generalizations:

Lazy

The lazy () function takes a lambda expression and returns a lazy <T> instance object to define the use of the lazy property. The lambda expression of the function is calculated at the first call (ie, use get ()) to use the same value for all calls Subsequent, such as:

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

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

The execution of the previous code will result in:

```
computed!
Hello
Hello
```

The calculation of the lazy property is synchronized. It is calculated in a single thread and all other threads will have the same value. If it is not important that the initialization process is synchronized, it will be allowed to execute more than thread at the same time. LazyThreadSafetyMode.PUBLICATION for the first parameter in the lazy function) The LazyThreadSafetyMode.NONE is used when it ensures that the configuration process will not occur more than thread.

Observation

The Delegates.observable () function contains two arguments: the initial value and the handler. The second argument is called each time the attribute is assigned (after the assignment process, not before), and the function contains its structure body has three parameters: the property to which the value is assigned, the previous value and the new value, as in the following code:

```
import kotlin.properties.Delegates

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

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

Which will result in:

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

Sometimes the function is called "veto" instead of "observable", where the handler will call the passable function before executing the attribute assignment.

Store the properties in Map

This feature is used in dynamic applications or JSON parsing, in which case the map object itself uses a generalized type of property, such as:

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

In the previous code, the constructor contains the map as follows:

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

The generalized properties get the values from the map via the key, which are the names of properties as in the code:

```
println (user.name) // The statement "John Doe"
println (user.age) // The value will appear 25
```

This also works for var properties using MableMap instead of read-only maps, so that the code is as follows:

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

Localized features (starting with version 1.1)

Kotlin allows local variables to be declared as generalized properties, such as making the local variable of the lazy type, such as:

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

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

The value of the memoizedFoo variable will only be calculated when it is accessed for the first time, and if there is a defect in the condition someCondition will never be counted.

Property Delegate Requirements

If the property is read-only, that is, type val, the circular must contain the getValue () function and have the following parameters:

thisRef: has the same type as the original property or any type of supertype, and in the case of extension properties the extended type.

property: It has the type KProperty <*> or its highest type.

This function can return the same type used in the property (or subtypes).

However, if the property is mutable, it should also contain the setValue () function and have the parameters:

thisRef: As in the getValue ()

property: as in the getValue () function

New value: It must be of the same property type or any higher type (supertype).

The two previous statements (getValue) and setValue () are either elements in the class class or as extension functions. The second case is used when a property of an object that does not contain these functions is used, Use the operator keyword before either.

The circular class can contain the definition of the implementation of either the ReadOnlyProperty and ReadWriteProperty interfaces, which have the required operator functions. The two interfaces are defined in the standard library in Kotlin as follows:

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

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

Translation Rules

The compiler generates an auxiliary property for each generalized property, generating the prop property, for example the prop $ delegate, and circulating the access code for this additional property to form:

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

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

It provides the Kotlin interpreter with all the necessary information about the prop property through arguments: this which refers to the external class instance C and this :: prop, which is a reflection object of the type KProperty that describes the property prop itself.

Note: The format: this :: prop (used to refer to the bound callable reference in the code directly) is available from Kotlin 1.1.

Delegate (starting with version 1.1)

It is possible to extend the creation logic of the objects to which the definition of usage is being propagated through the definition of provideDelegate. If the object (user on the right side of the) contains the function provideDelegate as a function in the class or extension function, This property is used if the property consistency is verified when it is created and not only when it is used (either through getter or setter).

For example, to check the property name before binding, the code becomes:

```
class ResourceDelegate <T>: ReadOnlyProperty <MyUI, T> {
   override fun getValue (thisRef: MyUI, property: KProperty <*>): T {...}
}

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

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

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

   val image by bindResource (ResourceID.image_id)
```

```
    val text by bindResource (ResourceID.text_id)
}
```

The provideDelegate () function (as in the getValue () function contains transactions:

thisRef: has the same type as the original property or any type of supertype, and in the case of extension properties the extended type.

property: It has the type KProperty <*> or its highest type.

The function is called for each property in the process of creating the instance of MyUI and directly executes the necessary checks.

Without the ability to achieve the connection between the property and its generalization, the name of the property must be explicitly passed to accomplish the same task as before, and difficult as in the following code:

```
// Verify the name of the properties without relying on
// provideDelegate function

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

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

The provider provideDelegate in the generated code to create the prop $ delegate. Compare the following code to the property declaration: val prop: Type by MyDelegate () and the code generated earlier (as explained in the translation rules clause where there is no existDelegate):

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

// The compiler generates this code with the function
// 'provideDelegate'
class C {
    // call the function to create circularization of the add-in property
    private val prop $ delegate = MyDelegate (). provideDelegate (this, this ::
prop)
    val prop: Type
        get () = prop $ delegate.getValue (this, this :: prop)
}
```

This function affects only the creation of the auxiliary property and does not affect the generated code for either the getter or the setter.

Chapter IIII

Patterns of codes commonly used in Kotlin

Apply the Style Guide

To set the IntelliJ formater in accordance with this guide, it is recommended to install the Kotlin extension in version 1.2.20 (or later) and set the editor by going to Settings, Editor, Code Style, "Set from ..." "in the upper right corner and from the menu select the Predefined style / Kotlin style guide option.

To ensure that the code is formatted according to the Kotlin directory, go to the inspection settings, do the property in Kotlin, then Style issues, and then file is not formatted according to project setting. (Such as naming conventions) are enabled by default.

Organization of Source Code

Directory Structure

The Kotlin source files are placed in the programming projects written in several programming languages on the same root as the source files in Java and follow the same folder structure (each file [folder] is stored in the [directory] folder by package configuration [package] Contained therein).

In software projects written entirely in Kotlin, it is recommended to follow the package structure with the deletion of the shared root packet. If all the code in the project exists in the "org.example.kotlin" package, for example, or in any of the sub-packages in it, In the "org.example.kotlin" package are placed directly in the root folder of the source files. The files in the "org.example.kotlin.foo.bar" package are placed in the "foo / bar" subfolder starting from the root folder of the source files without mentioning Shared package "org.example.kotlin.foo.bar".

Source File Names

If the file contains only one class (and may be associated with higher-level declarations), it will be labeled with the same label plus the .ket. If the file contains more than one category or only higher level statements, then you must choose a name that expresses the contents of this file and name it using the style of naming humps by starting with a large letter for each word such as: ProcessDeclarations.kt, An indication of what Chaferta is doing, and it is better to avoid arbitrary expressive names such as "Util".

Organize source files (Source File Organization)

Multiple declarations (such as classes, top-level functions, or properties) are placed in the same source file as long as these statements are inter-related and the total file size is acceptable (the number of lines can not exceed several hundred) .

In the particular case, when defining the extension functions for a class, which is related to all clients of this category, it is best to be placed in the same file as the same category. When it is intended for one customer, it is recommended to place it next to the client code without creating Files of their own.

Class Layout

Items are generally arranged in the following order:

*** Properties statements and initializer blocks**

*** Secondary constructors**

*** Method statements**

*** Companion objects**

It is not common to arrange the statements of the methods in a alphabetical or accessible way, and the normal functions are not separated from the extension methods. Rather, the associated elements are placed together so that those who read the code can follow the logical sequence to clarify what And must be adhered to in the same order (from elements located at the upper level of elements located at the lower level or vice versa) in all codes.

Nested classes are preferred next to the code they use, and when there are items prepared externally without reference to them within the class, they are eventually placed after the companion objects.

Interface Implementation Layout

When defining the interface, the items remain in the same order as the elements in the basic interface (and may be interspersed with some [private] methods used in the [implementation] process.

Overload Layout

Overloads are always contiguous in the class.

Naming Rules

The Kotlin language follows the same naming rules in the Java language; the names of packages should always be lowercase letters without the underscore (such as the org.example.myproject label), the label is not preferred in multiple words, Label the humps by starting with a small character (such as org.example.myProject).

The names of classes and objects begin with a capital letter and follow the pattern of naming humps, such as:

```
open class DeclarationProcessor {...}
```

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

Function Names

The names of the functions, the properties and the local variables start with a small character and the style of naming humps is adopted without the underscore, such as:

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

The factory functions, which are used to create an instance of the items, have the same label as the related category, such as:

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

Test Methods Names

When the test (and test only) attributes are used, the Kotlin language allows the use of spaces provided that they are placed between the backticks (`code`), but this label is not supported in the Android system.

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

   @Test fun ensureEverythingWorks_onAndroid () {
   }
}
```

Properties Names

Constants, which actually contain static data, are generally capitalized, and the words underscore are defined. These are the properties defined by the const type, the properties of the object or top-level object variables without a function to obtain On them (such as get), such as:

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

The properties of objects, top-level objects that contain objects and their operations, or editable data, are called camel humps by starting with a small character, such as:

```
val mutableCollection: MableSet <String> = HashSet ()
```

The naming of properties that refer to singleton objects follows the same pattern of statements as objects, such as:

```
val PersonComparator: Comparator <Person> = ...
```

When naming multiple constants, there are two methods; they are called in capital letters (and their words are separated by underscore) in several words (eg num class RED, GREEN) Large, and the label type is selected according to use in the codes.

Backing Properties Names

If the class contains two similar properties; one is part of a general API and one of the implementation details, then the underscore is used as a prefix before the name of the private property, such as:

```
class C {
  private val _elementList = mutableListOf <Element> ()
  val elementList: List <Element>
      get () = _elementList
}
```

Good choice of label

The name of the class is a noun or a noun that expresses what it is, such as List or PersonReader, and the name of the method is often a verb or actual phrase that indicates the task of this function, such as close or readPersons, The name indicates whether the substitutions change the object or return another object. For example, the sort expression means switching the order of the items in the same group, and the name sorted represents the child that returns a set (copy) of the group; To clarify the target names of the named program entity, it is best to avoid names without meaning such as : Manager, Wrapper, etc. When there are acronyms, the full name is written in uppercase letters if it is composed of only two characters, such as IOStream. It starts with a capital letter and continues in lowercase if it is more than two characters, such as XmlFormatter and HttpInputStream.

Formatting

In the format of the format, the language is similar to the Java language, such as using 4 spaces at alignment instead of using tabs. When using parentheses, the opening brace is placed at the end of the line where the constructor starts and the closing bracket is placed in a separate line Vertically aligned to the beginning of this block, such as:

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

The addition of a semicolon (;) is optional in the Kotlin language. Thus, the chapter by moving to a new line appears clear. The general layout here depends on the parentheses used in the Java language and may be malfunctioning if dependent on a different format.

Horizontal white spaces

Spaces are added on both sides of the binary transactions such as: a + b. The fields are specified in the form: 0.i. In the case of unary operators, no spaces are added such as: a ++.

Add a space between the flow control words (such as when, when, for, and while) and the next parenthesis.

Do not add a space before the primary arc in the basic definition of the code structures or define the methods or call them as shown in the code:

```
class A (val x: Int)


fun foo (x: Int) {}


fun bar () {
    foo (1)
}
```

*** No spaces are added after the start brackets [and] or before the ending brackets [and].**

*** Do not add any spaces on either side. Or.? Such as: foo.bar (). Filter {it> 2} .joinToString () or foo? .Bar ().**

*** Insert a space after the comment symbol // such as // This is a comment.**

*** No space is added on either side of the <and> brackets used to specify parameters of a particular type, such as: class Map <K, V> {...}.**

*** No space is added on both sides of the symbol :: such as: Foo :: class or String :: length.**

*** Do not add any distance before the code? Which is used to denote nullable type such as: String?.**

In general, it is best to avoid horizontal alignment of any kind so that if an identifier is renamed, for example, the general format of the definition or use is not changed.

Colon (:)

Insert a space before the two vertical points: in the following cases:

* When used for separation between type and top type (supertype).

* When generalized to a superclass constructor or another brick within the same class.

* After the keyword object.

No space is added before separating between the declaration and its type.

The distance is always added after them.

Example:

```
abstract class Foo <out T: Any>: IFoo {
    abstract fun foo (a: Int): T
}
class FooImpl: Foo () {
    constructor (x: String): this (x) {
        // ...
    }
    val x = object: IFoo {...}
}
```

Format Header (Class Header)

It is possible to write constructor parameters in a single line if their numbers are a bit like:

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

In the case of long headers, they must be coordinated so that each basic parameter of the builder falls in a separate line with the offset (4 spaces) and the termination arc is placed in a separate line and added - in the case of heredity - The topclass or the list of implemented interfaces, as follows:

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

    // ...
}
```

If multiple interfaces, the superclass constructor is placed first, and each interface is placed in a different line, such as:

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

    // ...
}
```

If the list of supertype is long, it separates a line between the vertical points and the types, which are written in vertical alignment, such as:

```
class MyFavouriteVeryLongClassHolder:
    MyLongHolder <MyFavouriteVeryLongClass> (),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo () {}
}
```

For a clear separation between the long header and the body, either a blank line should be placed after the header (as in the previous example) or the starting bracket should be typed in a separate line as follows:

```
class MyFavouriteVeryLongClassHolder:
    MyLongHolder <MyFavouriteVeryLongClass> (),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo () {}
}
```

The normal alignment (4 spaces) is added before the constructor's coefficients, so as to align the properties defined in the underlying construct with the same properties as the class structure.

Modifiers

If the same declaration is more than specific, it is placed in the following order:

public / protected / private / internal

expect / actual

final / open / abstract / sealed / const

external

override

lateinit

tailrec

vararg

suspend

inner

enum / annotation

companion

inline

infix

operator

data

If there is any annotation, it is written before the delimiters, such as:

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

It is best to delete any redundant settings (such as public), but this does not apply if libraries are working.

Formatting Annotation

The annotation is often placed in a separate line before the associated declaration and with the same vertical alignment as:

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

If there are more than one description and without arguments, it can be placed in the same line, such as:

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

They can also be placed if they are without arguments in the same line with the associated statement, such as:

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

File Annotation

The file annotation is written after the line of comment (if any) and before the package instruction is separated by a line (to confirm that it is not for the package), such as:

```
/ ** The license, copyrights, etc. * /
@file: JvmName ("FooBar")
package foo.bar
```

Format functions

The format of the function signature follows the following format if it can not be fully positioned in a single line:

```
fun longMethodName (
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType
): ReturnType {
    // Function structure here
}
```

The displacement is used in four spaces for parameter parameters in order to achieve similarity with constructor parameters. It is preferable to use the expression body of a function if it contains only one expression, such as:

```
fun foo (): Int {// bad format
    return 1
}

fun foo () = 1 // Good format
```

Expression Body Format

If the expression structure is not appropriate in one line, as in the declaration, then the attribution = in the first line and the offset in the next line can be marked by 4 spaces and the expression continues, such as:

```
fun f (x: String) =
    x.length
```

Format Properties

The line formatting style is based on read-only properties such as:

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

For more complex features, it is better to use the keywords get and set and separate lines as:

```
val foo: String
    get () {
        // ...
    }
```

In the case of initializer, if the attribution is long, a new line is placed starting from the following reference signal = and by displacing 4 spaces, such as:

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance (). GetDefaultCharsetForPropertiesFiles
(file)
```

Control flow statements

The parentheses {} are used for the instruction set if the expression in the if clause or in the WHERE statement is in a number of lines, which is removed by 4 spaces from the beginning of its parent instruction, and the parenthesis bracket and parenthesis bracket are placed together with a separate line,

```
if (! component.isSyncing &&
    ! hasAnyKotlinRuntimeInScope (module)
) {
    return createKotlinNotConfiguredPanel (module)
}
```

Thus, the separation becomes clear between the condition and the structure of the instructions. The keywords else, catch, finally, and the while keyword of the do / while loop on the same line as the closing bracket of the structure that precedes it are as follows:

```
if (condition) {
    // ...
} else {
    // ...
}

try {
    // ...
} finally {
    // ...
}
```

If a branch of the "when" directive extends over more than one line, it separates the other branches via a blank line such as:

```
private fun parsePropertyValue (propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue (propName, token.value)

        Token.LBRACE -> {// ...
        }
    }
}
```

Short sections are placed with the condition in the same line and without the use of brackets, such as:

```
when (foo) {
    true -> bar () // Good format
    false -> {baz ()} // Bad format
}
```

Call Format

If the parameter list is long, then a new line is added after the starting bracket and the transactions are moved by 4 spaces at the beginning of each line. The converged coefficients can be grouped together with the same line, with spaces on either side of the reference signal = The name and value of the parameter, such as:

```
drawSquare (
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

Chained Call Wrapping

When the serial call is encapsulated, the code is placed. Or.? With the following line and offset by 4 spaces such as:

```
val anchor = owner
    ? .firstChild !!
    .siblings (forward = true)
    .dropWhile {it is PsiComment || it is PsiWhiteSpace}
```

The first call is often separated by a new line but can be dispensed with.

Lambda format

Spaces are used on the ends of the brackets in the lambda expressions as well as on the ends of the arrow separating between the parameters and the body. If the call requires only one lambda, it should be passed out as far as possible, for example:

```
list.filter {it> 10}
```

When assigning a label to lambda, no space is placed between the label and the next start-up bracket, such as:

```
fun foo () {
    ints.forEach lit @ {
        // ...
    }
}
```

When declaring the names of the parameters in lambda in several lines, the labels are written in the first line and the following line is appended to it via the arrow symbol:

```
appendCommaSeparated (properties) {prop ->
    val propertyValue = prop.get (obj) // ...
}
```

If the list of parameters is long and does not fit in a single line, the arrow symbol is used in a separate line, such as:

```
foo {
   context: Context,
   environment: Env
   ->
   context.configureEnv (environment)
}
```

Documentary Comments

When writing long documentary comments it is agreed to put an additional star to start the comment as / ** in a separate line, and start each new line with the asterisk as shown in the following comment:

```
/ **
* Here's a line
* Here is another line
* /
```

Short comments are placed in one line as follows:

```
/ ** This short line comment * /
```

It is best to avoid using @param and @return tags, replacing them in the context of the documentary commentary, adding the transaction links used, but if the comment is long-term and difficult to formulate, it is possible to use the previous two tags. For example, instead of writing a comment in the form:

```
/ **
* Returns the absolute value of the given number.
* @param number The number to return the absolute value for.
* @return The absolute value.
* /

fun abs (number: Int) = ...
```

Best to write as:

```
/ **
* Returns the absolute value of the given [number].
* /

fun abs (number: Int) = ...
```

Avoid excess structures

It is helpful to delete any redundant structure if its existence is optional or specified in the IDE as unnecessary unnecessary; do not add any extra elements in the code in order to make it more explicit.

Type Unit

When you return the function Unit, there is no need to mention it in the code, such as:

```
fun foo () {// deleted here word type Unit
    ...
}
```

Semicolon semicolon (;)

Save the semicolon as much as possible.

Text Template Templates

You do not need to use brackets {} when you enter a simple variable in the text string template, as it is used for longer expressions, such as:

```
println ("$ name has $ {children.size} children")
```

Conventional use of language advantages

Immutability data stability

It is preferable to use static data on the variable, so the declaration of data is by keyword val instead of var if there is no need to convert the stored value later.

Always use the static collection interfaces such as: Collection, List, Set or Map to declare static groups. When using factory functions to create groups, it is best (if possible) to return these functions to static types, such as: (It is good to use Set instead of HashSet and use the function that returns the static type List <T> instead of the ArrayList <T> variable)

Always use the static collection interfaces such as: Collection, List, Set or Map to declare static groups. When using factory functions to create groups, it is best (if possible) to return these functions to static types, such as: (It is

good to use Set instead of HashSet and use the function that returns the static type List <T> instead of the ArrayList <T> variable)

```
// Bad use of variable type of value that will not change later
fun validateValue (actualValue: String, allowedValues: HashSet <String>)
{{}

// Good use as the type here is fixed
fun validateValue (actualValue: String, allowedValues: Set <String>) {...}

// ArrayList <T> Bad use of the variable type
val allowedValues  = arrayListOf ("a", "b", "c")

// List <T> is a good use of the fixed-type variable instead of the variable
val allowedValues  = listOf ("a", "b", "c")
```

Parameters default values

Declaring functions with default values for transactions is better than declaring them overloads such as:

```
// Bad
fun foo () = foo ("a")
fun foo (a: String) {...}
// Good
fun foo (a: String = "a") {...}
```

Type aliases

It is easier to use alias if the types are functional or in the case of parameters used several times under the code, as follows:

```
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map <String, Person>
```

Lambda transactions

If short lambda expressions are non-overlapping, it is best to use the term "it" instead of declaring the parameter. In lambda cases, overlapping and containing transactions must be explicitly declared.

Return commands in Lambda

It is best to avoid using multiple named commands to return in lambda, and it is easier to count it as a lambda reconstruction process, making it only one exit point. If not, lambda must be converted to an anonymous function.

The return order should not be used as a labeled return in the last instruction in lambda.

Named Arguments

The argument naming formula is used when there are more than one parameter of the same type, or when there are Boolean transactions, if not all transactions are clear from the general context, such as:

```
drawSquare (x = 10, y = 10, width = 100, height = 100, fill = true)
```

Use Conditional Statements

It is preferable to use the expression form for try, if and when such as:

```
return if (x) foo () else bar ()

return when (x) {
    0 -> "zero"
    else -> "nonzero"
}
```

Which is better than the following code:

```
if (x)
    return foo ()
```

```
else
    return bar ()

when (x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

when and if

It is preferable to use if instead of when only two conditions exist for the condition, ie instead of typing the code:

```
when (x) {
    null -> ...
    else -> ...
}
```

The code is written better:

if (x == null) ... else ...

Where is reserved for cases where there are more than two options.

Boolean values can be null (nullable) in the condition

In that case, the conditions if (value == true) and if (value == false) must be checked.

Loops

It is preferable to use a loop at a higher level (eg filter, map, etc.) instead of using loops, except forEach. It is best to use the for loop instead, unless the future of forEach is capable of containing a nullable value, Or if the forEach loop is used as part of a longer call-chain call.

When deciding whether to use the loop or using a complex expression based on several higher-order functions, the cost of the operations performed in each case should be estimated and a good level of performance maintained.

Using loops based on domains (Ranges)

It is advisable to use the until keyword in open-field loops, such as:

```
for (i in 0.nn - 1) {...} // bad use
for (i in 0 until n) {...} // Good use
```

Use Strings text strings

The use of string templates is recommended for interconnection.

It is best to use multiple-string strings instead of using \ n the string value.

In order to maintain alignment in text strings in several lines, the trimIndent function can be used if the internal alignment of the resulting string is not needed or the trimMargin function is used if its internal alignment is needed, such as:

```
assertEquals ("" "Foo
          Bar "" ". TrimIndent (), value)

val a = "" "if (a> 1) {
      | return a
```

```
        |} "" ". trimMargin ()
```

Functions

Sometimes, there are some stylistic considerations of preference between them. It is advisable to use functions instead of functions when the following conditions are met: In the built-in algorithm:

<u>* Does not contain a throw</u>

<u>* Inexpensive computations (or cached) during the first run)</u>

<u>* Returns the same results for different calls as long as the object state has not changed.</u>

Using Extension Functions

The function is used primarily on an object. It can be converted to an additional function by making the object in its future. In order to reduce the overload on the APIs, it is preferable to specify the visibility of the additional function as much as possible, It is recommended that the type be private when you need to use additional functions, whether local or as a member or a top-level.

Use the internal functions of Infix Functions

The declaration of the function as an internal function only when working on two objects with the same role is a good example of this: and and to and zip. It is a useless example: add, and the function is not declared as an internal function if it has a role to change the object Receiver receiver.

Functions Producing Factory Functions

The functions that are produced are not called the same name as the class, but are used by their name as a distinctive name indicating what this function does. In a few cases (if the function does not distinguish it), it can be the same as the name of the product. Example:

```
class point (val x: Double, val y: Double) {
    companion object {
        fun fromPolar (angle: Double, radius: Double) = Point (...)
    }
}
```

The functions produced can be used instead of overloaded if the object has several overloaded colors that do not require any superclass constructors and can not be reduced to one construct with default values.

Platform types

You must explicitly declare the type of any function or child of the public type if it reverts a type of platform as in the code:

```
fun apiCall (): String = MyJavaApi.getProperty ("name")
```

Similarly, a property is either a package-level or a class-level that has a platform-like expression as the code:

```
class Person {
    val name: String = MyJavaApi.getProperty ("name")
}
```

If the platform-type expression is assigned to a local variable, the type can be declared or ignored, such as:

```
fun main (args: Array <String>) {
    val name = MyJavaApi.getProperty ("name")
    println (name)
}
```

Using Scope Functions apply / with / run / also / let

Kotlin provides a number of functions that allow a part of the block to be executed within the context of the object code. To select the appropriate function, consider the following points:

* Are the satellites called by different objects within the block or passing the object as an argument? You use a function that allows access to the object in question, such as it (not also or let), and uses the function also if the receiver is not used in this part at all, such as:

```
// The object meaning is
class Baz {
    var currentBar: Bar?
    val observable: Observable

    val foo = createBar (). also {
        currentBar = it // Access the property in the class
        observable.registerCallback (it) // Pass the object as a medium in the
function
    }
}
// The future is not used in this section
val foo = createBar (). also {
    LOG.info ("Bar created")
}

// Object is this
class Baz {
    val foo: Bar = createBar (). apply {
        color = RED // Access only properties from Bar
        text = "Foo"
    }
}
```

* How will the result of the call? If the result is an object then use the apply or also function, or if the result is a value of this part then use the function

with or let or run, such as:

```
// The returned result is an object
class Baz {
    val foo: Bar = createBar (). apply {
        color = RED // Accessing only properties of Bar
        text = "Foo"
    }
}
// The returned result is a value
class Baz {
    val foo: Bar = createNetworkConnection (). let {
        loadBar ()
    }
}
```

* Is the object nullable or is the result of a call chain? If so, you must use the apply or let or run function, and use with or otherwise, such as:

```
// The object is nullable
```

```
person.email?.let {sendEmail (it)}


// Object here can be accessed directly and not null
with (person) {
    println ("First name: $ firstName, last name: $ lastName")
}
```

Libraries

When writing libraries, it is recommended that you adhere to some general rules to ensure API interfaces are stable:

* Visibly define the visibilty of the elements, so that you do not know - by mistake - of a public type.

* Declare the types of returned values in functions and properties to avoid a type error when changing the usage definition.

Providing documentation for all public items except for override operations that do not require any new documentation to support the creation of documentation for the library.

Programming structure in Kotlin language:

2.1 Packages in Kotlin

Declaring packages

The source file usually starts with a declaration of packages such as:

```
package foo.bar

fun baz () {}

class Goo {}

// ...
```

All the contents of this file (such as classes and functions) are then placed in the declared package. In the previous code, the actual full name of the baz () function is foo.bar.baz and the full name of the Goo type is foo.bar .Go.

If you do not specify the package at the beginning of the file, its contents follow the default package that is not named.

Import default packages (Default Imports)

A number of packages are automatically imported to be included in each file, namely:

```
kotlin. *

kotlin.annotation. *

kotlin.collections. *
```

kotlin.comparisons. * (starting with version 1.1)

kotlin.io. *

kotlin.ranges. *

kotlin.sequences. *

kotlin.text. *

Some additional packages are imported according to the target platform:

JVM:

java.lang. *

kotlin.jvm. *

JS:

kotlin.js. *

Additional Imports

Files may contain packages that import packages other than default packages, and have several states:

- import via one name, such as:

import foo.Bar // Access is available without restrictions for Bar

Import all content in scope (package, category, object, etc.), such as:

import foo. * // Access is available for all content in foo

There may be mixed between two similar names; then the alternative label is used depending on the keyword as:

import foo.Bar //

import bar.Bar as bBar // The alternative label means 'bar.Bar'

Import is not limited to importing items, but is also used to import other statements such as:

- functions or properties at the top-level

- Functions or properties declared in object declarations.

- constants constants

Kotlin does not support the "import static" import format (as in the Java language), importing these statements using the usual import keyword.

Visibility of Top-level Declarations

If the top-level declaration is set to private, it has only the file it contains.

Chapter II

Advanced topics

Declarations in Kotlin

Declaration of disassembly

You may sometimes need to break the object into a number of variables, such as:

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

The previous version is called disassembly, which creates more than one variable at the same time (the name and age variables) where they are allowed to be used independently as in the following code:

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

The decommissioning statement translates as follows:

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

The component1 () and component2 () are examples of the basic conventions used in the Kotlin language (see transactions such as +, * and for, etc.), and allows any object on the right side of the declaration to be disassembled as long as possible The required number of component functions through it (may be component3 and component4) and so on) which must be specified by the operator to allow use in this statement. The disassembly statement in the for loop is also used as:

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

Where variables a and b obtain values from calling the component1 () and component2 () functions in the collection elements.

Practical application: Reset two values of the function

When you need to return two elements of the function (for example, return the result object and a condition), the agreed method for doing so is to create a data class and return an object, as in the following code:

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

    return Result (result, status)
}

// Now .. Use the function
val (result, status) = function (...)
```

Permission to disassemble is permitted here because the data classes define the componentN () functions automatically.

Note: Standard Pair can be used to return function Pair <Int, Status>, but it is best to label the data as appropriate for use.

Practical application: Disassembly statements and Maps

The most appropriate way to navigate map elements is as in the following code:

```
for ((key, value) in map) {
   // Do instructions based on value and key
}
```

Here, the following must be observed:

Represent the map object as a sequence of values through the iterator () function

Represent all of the elements as a pair (pair) through the functions component1 () and component2 ()

The standard library provides such additions:

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

Making it easier to use statements by disassembling in loops based on maps (as well as objects of data class instances and the like).

Use underscore (_) for unused variables (starting with version 1.1)

If there is no need for the variable to declare disassembly can be dispensed with the name of the lower police such as:

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

In this way the component function will not call for its approval.

Disassembly in lambdas (starting with version 1.1)

If the lambda coefficient of Pair (or Map.Entry or any other type has an appropriate component function), then more than one coefficient (surrounded by parentheses) can be used. , As in the following code:

```
map.mapValues  {entry -> "$ {entry.value}!" }
map.mapValues  {(key, value) -> "$ value!" }
```

The difference between the authorization and disqualification of two workers is as follows:

```
{a -> ...} // one coefficient
```

```
{a, b -> ...} // parameters
{(a, b) -> ...} // Declined statement with two variables
{(a, b), c -> ...} // Declined statement with two variables and a third independent parameter
```

If a part of the clearance statement is not used, replace it with the underscore without having to name it, such as:

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

The total type of disassembly permit transactions or one of them can be determined separately as in the following code:

```
map.mapValues  {(_, value): Map.Entry <Int, String> -> "$ value!" }

map.mapValues  {(_, value: String) -> "$ value!" }
```

Collections: List, Set, and Map in Kotlin

Kotlin (unlike many programming languages) distinguishes between the mutable and immutable groups, and its ability to precisely control where groups are allowed to reduce bugs and better design APIs.

It is important to know the differences between the read-only view of the variable group and the actual fixed set. It is easy to create each but the Kotlin species system does not tell the difference between them.

The Type <out T> in Kotlin is an interface that provides read-only operations such as size, get and the like. As in Java, it inherits the collection <T>, which in turn inherits from Iterable <T> , And the methods that change the list are added via the MutableList <T> interface. This sample also includes Set <out T> / MutableSet <T> and Map <K, out V> / MutableMap <K, V>.

The following code shows the basic uses of group types:

```
val numbers: MutableList <Int> = mutableListOf (1, 2, 3)
val readOnlyView: List <Int> = numbers
println (numbers) // The result will appear
                  "[1, 2, 3]"
numbers.add (4)
println (readOnlyView) // The result will appear
                  "[1, 2, 3, 4]"
readOnlyView.clear () // will not be translated
val strings = hashSetOf ("a", "b", "c", "c")
assert (strings.size == 3)
```

Kotlin does not create custom structures to create lists or groups, but rather depends on the dependencies in the standard library, such as: listOf (), mutableListOf (), setOf (), mutableSetOf () , But when you create a map type, you use the conventional formula: mapOf (a to b, c to d), in the not-performance-critical codes. The readOnlyView variable refers to the same menu and changes as the undering list changes. If the only type of reference for the list is read-only, this group can be said to be static, Code:

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

The listOf implementation is defined using an array list at the moment, but the forward-type will depend on non-variable and more efficient types of memory.

The read-only types are covariant, which means that the <Rectangle> list can be assigned to List <Shape> (assuming that the rectangle [Rectangle] inherits from [Shape]), and this is not allowed in the variable group types because it will result in Malfunction during implementation.

In some cases, you may need to return to the caller a snapshot of the group at a specified point in time so that it does not change later, such as:

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

The toList plug-in copies only the list elements, so the returned list will not change. There are many useful additional functions in lists and sets, such as:

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

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

In addition to some utilities such as sort, zip, fold, reduce, and more. And follow the map to the same form where it is easy to create and access as described in the code:

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

Fields (Ranges) in the Kotlin language

Use domains

Range expressions are formulated by field-dependent rangeTo functions, which in turn complement the in and! In parameters. The field can be defined for any type of comparison. In the case of integral primitives types, Use optimized, these are some examples:

```
if (i in 1.10) {// equivalent to wording
              // 1 <= i && i <= 10
   println (i)
}
```

The Intrange, LongRange, and CharRange fields have an added advantage as iteration can be traversed by the compiler, which converts it to a corresponding format for the indexed loop in Java at no extra cost, such as:

```
for (i in 1..4) print (i) // "1234" The values  will appear
```

```
for (i in 4..1) print (i) // No value will appear
```

But what if the need to move items in reverse order? This is simple! Using the downTo () function defined in the standard library, is as follows:

```
for (i in 4 downTo 1) print (i) // "4321" The values will appear in reverse order
```

You can also move the value of a step other than one using the step () function as in the following code:

```
For (i in 1..4 step 2) print (i) // "13" the values  will appear

For (i in 4 downTo 1 step 2) print (i) // "42" the values  will appear
```

To create an open-ended domain (without access to the last element), the until function is used as in the code:

```
for (i in 1 until 10) {// 10 is excluded and is equivalent to the format
                         // i in [1, 10]
```

```
    println (i)
}
```

Mechanism of action areas

The fields define the use of the ClosedRange <T> shared interface in the library, which expresses the mathematical concept of the closed field (including the start and end values) and defines the comparable values, and has two points: start and endInclusive (included in the field) , And contain the basic process used in it and it depends on the formula using the parameters in and!

The types of integral sequences such as IntProgression, LongProgression, and CharProgression are the mathematical sequences, known as the first element, the last element, and the non-zero step. The field begins with the first value and produces each subsequent value from the previous value after adding the step step to the last last element. Unless the sequence is empty.

The sequences are a subtype of Iterable <N> where type N is an type Int or Long or Char, and therefore can be used in for and some functions such as map, filter, and so on. Progression is similar to the indexed loop, In Java or JavaScript, such as:

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

For example, the IntRange field defines the use of ClosedRange <T> and extends the introsection of the introspection. , So all processes defined for IntProgression will also be available in IntRange, and the result of the downTo () and step () functions is always the Progression type. Sequences are generated using the defined fromClosedRange function in its companion objects as:

IntProgression.fromClosedRange (start, end, step)

Where the last last element of the sequence is calculated to find the largest value that does not exceed the end value if the step step is positive, or to find the smallest value of at least the end value in the case of the negative step step so that it is (last - first)% step == 0.

Some help functions

The rangTo () function

The coefficients of the rangTo function () - in the integrated types - call the objects in the Range * items such as:

```
class Int {

   // ...

   operator fun rangeTo (other: Long): LongRange = LongRange (this, other)

   // ...

   operator fun rangeTo (other: Int): IntRange = IntRange (this, other)

   // ...

}
```

The floating-point numbers (such as Float and Double) do not know the rangeTo argument, but instead use the generic type found in the standard library, such as:

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

The domain (iteration) elements that result from this function can not be traversed.

DownTo () function

The downTo () function is defined for any pair of integrative types. The following are two examples:

```
fun Long.downTo (other: Int): LongProgression {
    return LongProgression.fromClosedRange (this, other.toLong (), -1L)
}

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

The reversed function ()

The extension function (reversed) is defined for all sequences of progresses, resulting in reverse sequences, ie,

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

Step () function

The extension function is known as step () for all the Progression sequences, resulting in modified step sequences, where the step value is always positive and therefore does not change the function of the direction of passage to the domain elements, such as:

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

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

The last value of the returned sequence may differ from that in the original sequence to maintain the condition (last - first)% step == 0, such as:

```
(1.12 step 2) .last == 11 // [1, 3, 5, 7, 9, 11] The consecutive values
(1.12 step 3) .last == 10 // [[1, 4, 7, 10] The sequence of values
(1.12 step 4) .last == 9 // [1, 5, 9] The consecutive values
```

Type Check and Switching in Kotlin

The coefficients is and! Is

Kotlin supports the object compatibility check with one of the types during execution, depending on the is or its! Is as in the code:

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

if (obj! is String) {//! (obj is String) is equivalent to the formula
   print ("Not a String")
}
else {
   print (obj.length)
}
```

Smart Casts

There is often no need to make the conversion explicit in Kotlin because the compiler tracks the checks (through parameter) and explicitly converts the immutable values and converts them automatically and securely when needed, as in the code:

```
fun demo (x: Any) {
    if (x is String) {
        print (x.length) // x will convert the parameter
                        // String automatically to type
    }
}
```

Where the conversion is "safe" if externally checking leads to a return order such as:

```
    if (x! is String) return
    print (x.length) // x will convert the parameter
                    // String automatically to type
```

Or on the right side of the coefficients && || As in the code:

```
    // Automatically converts the variable to the text string type
    // To the right side of the lab || In the condition
    if (x! is String || x.length == 0) return

    // Automatically converts the variable to the text string type
    // to the right side of the coefficient && in the condition
    if (x is String && x.length> 0) {
        print (x.length) // x is automatically cast to String
    }
```

Such smart conversions can be used in both the when and the while expressions, as in the code:

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

Smart conversions are not performed when the compiler is unable to ensure that the variable does not change between the verification and usage process. In more detail, smart conversions occur in the following cases:

Local variables of type: always with the exception of local delegated properties.

Properties of the type val: only if the properties are private or internal, or verified in the same module that has the property declaration, and conversions are not made in the case of open properties or those that are open, Have a custom getter.

Variable local variables: Only if the variable does not change between verification and use and lambda is not converted from value or not localized.

Variables of the var type: Never, because the variable may change at any given moment and across any other code.

Unsafe conversion factor

The use of the conversion factor will result in an exception if the conversion is not available. This is what we call unsafe conversion. In Kotlin, it depends on the parameter as in infix, as in the code:

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

Null can not be converted to String because it is not a null type, so the value of variable y is null and will result in an exception. In order to achieve parity with Java language, the nullable type must be in the right side of the conversion as in the code:

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

Safe conversion factor (Nullable)

The safe conversion parameter uses as? (Which returns the null when a defect occurs) to prevent the previous exception, such as:

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

Although the right side of the parameter as? Is of the string type (which does not accept null), the result of the total conversion may be null (ie nullable).

Remove Type (Erasure) and Check for Generic Type

Kotlin ensures the safety of species (type-safety) - including generalized species - during compilation, but during runtime some objects of generalized species may not contain information about their actual type media, <Foo>, for example, list type <*>, since there is no way to verify that the organism belongs to a general type (species media) during implementation.

Thus, the compiler prevents the validation of the parameter is during execution because of the removal of the type erasure types, such as ints is List <Int> or list is T, but it is possible to check the objects that are in violation of the star-) As in the code:

```
if (something is List <*>) {
   something.forEach {println (it)} // `Any?` will make elements of type
}
```

Similarly, since the type arguments of the object are validated during translation, it is also possible to check the is parameter or to perform any conversion containing the non-generic part of this type. In the code:

```
fun handlestrings (list: List <String>) {
    if (list is ArrayList) {
        // will convert the object list
        // to type `ArrayList <String>`
    }
}
```

The same mode of handling deleted type media is allowed for conversions that do not take type arguments into account, such as: list as ArrayList. In the case of inline functions with reified operators (inline) in each call site, the function is checked: arg is T, but if arg is a generalized instance itself, , Like:

```
inline fun <reified A, reified B> Pair <*, *>. asPairOf (): Pair <A, B>? {
    if (first! is A || second! is B) return null
    return first as A to second as B
}

val somePair: Pair <Any ?, Any?> = "items" to listOf (1, 2, 3)
val stringToSomething = somePair.asPairOf <String, Any> ()
val stringToInt = somePair.asPairOf <String, Int> ()
val stringToList = somePair.asPairOf <String, List <* >> ()
val stringToStringList = somePair.asPairOf <String, List <String >> () //
Breaks type safety!
```

Unchecked Casts

The concept of type erasure in Kotlin prevents - as mentioned earlier - the actual type media checks of generic type instances during runtime, and the generalized species may not be sufficiently interconnected to ensure the safety of the species, However, some high-level programs involve the safety of species, such as:

```
fun readDictionary (file: File): Map <String, *> = file.inputStream () .use {
   TODO ("Read a mapping of strings to arbitrary elements.")
}
// Save map
// with valid values  in that file
val intsFile = File ("ints.dictionary")

// Warning: Unverified conversion
// `Map <String, *>` to `Map <String, Int>`
val intsDictionary: Map <String, Int> = readDictionary (intsFile) as Map <String, Int>
```

The compiler issues a warning at the last line of code, which can not be fully verified during execution, and the values in the map object are not necessarily int.

In order to avoid this type of remittance, the program is restructured. Two interfaces can be added: DictionaryReader <T> and DictionaryWriter <T> in the previous program, which have a safe implementation definition of different types, abstractions can also be added to transfer non-conversion Unchecked cast from the calling code to the definition of implementation, and it is also useful to use the generic variance.

In the case of generic functions, the use of reified type coefficients makes the conversion, such as: arg as T, checked unless the arg type has its own mutants to be removed.

To bypass the unverified conversion warning, add the "annotation" as @Suppress ("UNCHECKED_CAST") to the instruction or statement in which the warning occurs, as in the code:

```
inline fun <reified T> List <*> .AsListOfType (): List <T>? =
    if (all {it is T})
        @Suppress ("UNCHECKED_CAST")
```

```
this as List <T> else
null
```

The array types in the JVM environment (eg Array <Foo>) retain information about the erased type of their elements, and the conversion process is partially achieved by the matrix type, where the containability of the null value and the actual type media of the elements (eg erasaed), for example, foo as Array <List> String>?> will be successful if foo is a matrix that holds either <*> whether nullable or not.

Exceptions in Kotlin

}

Exception Classes

All categories of exceptions fall in the Throwable Kotlin language, where each exception has a message for notification, a stack trace, and an optional reason.

It uses the throw expression to throw an exception object as:

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

Try is used to detect the exception as in the code:

```
try {
   // Code
}
catch (e: SomeException) {
   // exception handlers
}
finally {
   // Optional section
}
```

The exception may contain more than one section of a catch or may not be written originally, and the partition can be permanently deleted, but it is necessary to have at least one partition of catch or finally in the exception.

Try expression

The try section is an expression in Kotlin and therefore it is possible to return a value as in the code:

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

The value returned from the try expression is either the last expression value in the try section or the last expression value in the catch section (or in all sections if there is more than one section of catch), because the presence of the partition finally does not affect the overall result of the expression.

Checked Exceptions

Kotlin does not support this type of exception for a number of reasons. We present it with a simple example; the following code is from the JDK interface that is implemented in the StringBuilder class:

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

This code means that each time a string is attached to an object (such as StringBuillder, log, console, etc.), exceptions are treated like IOException, but why? Because many IO operations can occur (also known as Appendable in Writer), so there will be a lot of this code:

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

It is not good to do such things, nor should exemptions be ignored, especially in large projects. This would reduce productivity without a significant improvement in quality.

The Nothing Type

The throw section is also an expression in the Kotlin language and may therefore be used as part of the Elvis expression (see Security Null Safety) as in the code:

```
val s = person.name?: throw IllegalArgumentException ("Name required")
```

The throw type is a special type in Kotlin which is nothing and has no value. It is used to identify the parts that are not accessible in the code. This type can also be used with functions that do not return, In the code:

```
fun fail (message: String): Nothing {
   throw IllegalArgumentException (message)
}
```

When the function is invoked, the interpreter (for example, the Nothing) will know that the implementation will not continue with what follows, such as:

```
val s = person.name?: fail ("Name required")
println (s) // It is known that the variable is formatted here
```

This type is used in another case when the type inference is assumed where the type Nothing null (nullable) has only one possible value, which is null, so when using the null value for the initial initialization of the value of a type of evaluator without any other information useful in determining the type , Then the compiler will make the type of Nothing as in the code:

```
val x = null // variable of type `Nothing?`
val l = listOf (null) // variable of type `List <Nothing?>
```