**Kotlin**

Last Updated: Dec 2021

1. Kotlin is a modern multi-platform language meant to make tasks easier. It has very little boilerplate, is type-safe and interoperable with Java. It’s nnbd as well. It requires JDK and JRE to be worked on, Java Runtime Environment is needed to run Kotlin apps and Java Development Kit is needed to write them.
2. Semicolons are optional.
3. Data Types:
   1. Primitive:
      1. Int, Float, Double, Long, Byte, Short, Char, String
      2. All of these have their Unsigned counterparts. UInt and so on.
      3. Unsigned types values append u.

val a= 42u; //UInt inferred

val b = 1\_000\_000u //ULong inferred

val c: UShort= 1u; //UShort

val d= 30UL ; //ULong

* 1. Non-primitive:
     1. List, Array
  2. Any, Unit and Nothing: Any is the super type of all types in Kt. Any can hold any type. Unit, same as void, but can be passed around and also assigned to template types. Nothing is used with functions that never return anything, such as when all their codepaths throw exception. Nothing is the subtype of all types in Kt.

If Any object holds a Float or Double then the comparison used when comparing 2 values is non-IEEE 754 standard, meaning -0.0 is not equal to 0.0, NaN is greater than POSITIVE\_INFINITY and NaN is equal to NaN.

* 1. Boxing and unboxing is supported. It isn’t recommended since it consumes more memory.
  2. To declare a variable: var is reassignable, val is final. const val is compile time constant val

val <name>:<type> = <value>;

var …

The type can be omitted.

val and var with T need a value of T type, but if we assign them later then that works as well, just like final and var in Dart.

val x: Int;

x=20;

x=30 //error

const val aabc= 2;

const val can only be at global scope or in singleton classes.

* 1. lateinit var … : Just like late var in dart, except the variable can’t be a val. It means that the variable itself will not be null but it won’t be initialized in ctor or directly, it will be initialized before it is used. Not doing so, is a runtime error.
  2. ‘\_’ can be used to separate long numbers for the sake of readability.

For example:

val oneMil= 1\_000\_000

* 1. Strings same as C++, ‘ for chars and “ for strings.
  2. String template: “I’m a happy $someVariable ${someVar.someProp}”

String template can use “ inside { } without escaping it.

val str= “a car is ${if (20 < 30) “fast” else “slow” }”;

stores ‘a car is slow’ in str.

* 1. nnbd: For type T, T? is its nullable variant.

var a: Int= null; //error

var b: Int?=null; //works

Like dart, nullable values can use ? to access props. a?.someProp.

Similarly, ! ! is used to enforce the value will be non-null, else an exception. a!!.someProp.

* 1. To create list,

val x= listOf(“a”,”b”,”c”);

creates an immutable list

val y= mutableListOf(“x”,”y”);

y.remove(“y”); //works

Types must always be same.

For empty list,

val items= listOf<Int>();

* 1. To create array,

val x= arrayOf(“a”,”b”);

arrays are always immutable.

Types can be different, arrays use boxing. To avoid boxing,

val y= intArrayof(1,2,3);

Arrays can be added (concat) and returned into new var,

val z = x + y;

* 1. Arrays and Lists can nest each other. While Lists can be printed out directly, arrays must be printed out using a java utlity otherwise their memory address is printed

val arr= Array (5) {it\*2};

println(java.util.Arrays.toString(arr));

Here, ‘it’ means index, starts at 0, it multiplies each value by 2 for 5 sized array.

* 1. ++ and – work
  2. Tuples: Created as Pair objects.

val obj: Pair<T,K>= 1 to “abc”;

The ‘to’ operator is used to create Pair objects. It can also help in creating maps.

val map = mapOf(1 to "x", 2 to "y", -1 to "zz")

println(map) // {1=x, 2=y, -1=zz}

* 1. Pairs and Triples: Just like KeyvaluePair in c#.

val somePair= “abc” to “xhz”;

println(“{somePair.first} is {somePair.second}”);

val someTriple= Triple(2,3,43);

Access using ..first, second, third.

Both pair and triple can be ..toString() to get a string of a list, or ..toList() to get a list.

Destructure as expected.

* 1. HashMap: Takes pair objects and maps them.

val someMap= hashMapOf(“a” to “b”, “c” to “d”);

someMap[“a”] //or someMap.get(“a”) returns “b”; if not found then null

mutableMapOf() returns a mutable hashmap.

* 1. null: null also has a type, the Nothing? type. null is a value. Nothing means no return while Nothing? means either no return or null. Nothing? has only 1 value, null.
  2. Volatile: Volatile variables guarantee linearizable, i.e., reads and writes to the variable but not to massive atomic operations.

@Volatile

val abc= 2

This variable won’t be cached and any r/w will directly be performed to the value in memory.

* 1. Spread operator: Arrays can be spread in-place using the ‘\*’ operator.

like

val a = arrayOf(1, 2, 3)

val list = arrayOf(-1, 0, \*a, 4)

* 1. Reified Types: These types allow passing a type as an argument. A normal type is erased at runtime, unless we do a compile time cast. However, if the object is obtained at runtime then we can’t do a compile time cast due to type erasure. We can use reified types in that case, as reified types, aka real types, preserve type information even in runtime which allows us to compare the type of the erased object to these types. If the passed object is the same type as the reified type then no casting is needed and compiler will know the type, this also allows the static code analyzer to know the type.

For example:

inline fun <reified T> TreeNode.findParentOfType(): T? {

var p = parent

while (p != null && p !is T) {

p = p.parent

}

return p as T?

}

If we don’t use inline with reified type argument functions then reflection is used, i.e., reified types have no use and this is why it is an error as well.

Non-reified types (types not having a ‘reified <type variable>’) cannot be used as an argument for a reified type parameter.

1. :: operator: Reflection operator. It’s used to reflect on some property/class/function.

Function: ::<funcname> decomposes a function into a lambda and can be used to pass around.

Property: ::<propname> reflection props on props, ..isInitialized can be used to return a bool, useful with lateinit props.

Class: ::<Classname> reflection on class, useful with methods that require a java class as kotlin makes java bytecode, at runtime it is almost like running java hence they can get a reflected kotlin class as java class.

1. Bools and Conditional statements same as C++, if-else if-else.
   1. when: Like a switch.

For example:

when (numberOfFish)

{

0 -> println(“yo”);

in 1..20 -> println(“yo range”);

else -> print(“yo else”);

}

when automatically adds break; after each option.

* 1. Elvis Operator: ‘?:’ Used as shorthand for, if x is not null.

For example

var someVal= x ?: 0;

Just like null-coalescing operator ?? in other langs.

* 1. Single line if-else:

val isHot =if (20 < 30) true else false;

stores false in ‘isHot’

* 1. is and !is for type check. These also perform smart cast as in C#, i.e., in an if else with ‘is’, the object is casted if it is assignable. Works in when and while blocks as well.

1. Typecast:
   1. ‘as’:

val x: String? = y as String?

if T were non-nullable it would lead to exception on null, or any incompatible value of y.

val x: String?= y as? String

doesn’t throw if incompatible, instead returns null.

* 1. Type Erasure and Generic Type check: At runtime, instances of generics don’t hold information about their actual type args. I figure this is the same as in Dart. List<Foo> is erased to List<\*> at runtime, this is why there is no way to check whether an instance belongs to a generic type with type args, i.e., like something is List<int>. This is why compilers prohibit ‘is’ on generics as they are types that can’t be compared due to type erasure.

But we can compare against star projected types,

if (something is List<\*>) {

something.forEach { println(it) } // The items are typed as `Any?`

}

However, if the check is being performed under a compile time cast then it can be done.

For example:

fun handleStrings(list: List<String>) {

if (list is ArrayList) {

// `list` is smart-cast to `ArrayList<String>`

}

}

* 1. Inline functions with reified types have their actual type args inlined so they can use args is T checks. But if an argument is an instance of a generic type itself, then it faces erasure.

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>>() // Compiles but breaks type safety!

This is to say, for a Type T even if the function is providing type safety at compile time, if its types have their own args then they will be erased.

* 1. We can use Suppress to hide warnings if we are sure of a type.\

inline fun <reified T> List<\*>.asListOfType(): List<T>? =

if (all { it is T })

@Suppress("UNCHECKED\_CAST")

this as List<T> else

null

1. Loops: for(x in y){…}
   1. We can loop over array elems and indices at same time.

for((x,y) in aList.withIndex()) {…}

* 1. Steps and ranges: Ranges are inclusive of both borders.

for(x in 1..5) {…}

for (x in 5 downTo 1) {…}

for (x in 1..5 step 2) {…}

for (x in ‘a’..’e’) {…} //gets a b c d e

* 1. There’s no for (var;limit;inc/dec) {…} in Kt.

There’s while, do {…}while and repeat(<count>) {…}

* 1. a

1. Functions: main() is the entrypoint for any project.
   1. Syntax:

fun <name> (<params: type>): <return type> {…}

* 1. Every function returns something, even void as in main returns kotlin.Unit, which is automatically inserted if not explicitly.

This also means, everything in Kt has value. val some=println(“yo”); will store kotlin.Unit in ‘some’. The only exceptions are for and while loops, which don’t return anything.

* 1. main with args:

fun main(args: Array<String>) {…}

* 1. Default value:

fun xyz(aVar: String = “lol”) {…}

All params are named params in Kt. Hence this func can be called in 3 ways,

xyz();

xyz(aVar=”something”); //named

xyz(“seeeee”); //positional

It is advised to declare default params after normal params.

Default params can be retrieved from other functions too.

fun car(speed: Int = getSpeed()) {…}

* 1. Required params: Any param without a default value and a non-nullable T type is a required param.
  2. Compact function: aka Single-Expression Functions, just like in dart.

fun xyz(temp: int) = temp > 20;

returns bool.

* 1. Lambdas: Declared inside {…}

val aLambda= {car: String -> println(“vroom $car”)}

then just call aLambda.

val verboseLambda: (Int) -> Int = { carID -> carID/2 };

Here, we declare the type of verboseLambda to be a function that has 1 arg of type int and returns an int.

* 1. Higher-Order Function: A func that accepts another func or lambda.

fun someFunc (carID: Int, anotherFunc: (Int) -> Int ) {…}

We can call using a lambda or pass another func. Unlike passing a lambda, another func is passed by using ‘::’.

fun carLol(id: Int): Int {…}

someFunc(20, ::carLol);

:: passes carLol as reference.

* 1. If higher-order functions have func param as last then they can use last parameter call syntax;

fun someFunc (carID: Int, anotherFunc: (Int) -> Int ) : Int {…}

var id=20;

id= someFunc(id) {carID -> carID/2};

* 1. Extension lambda: A lambda can be turned into an extension function.

fun someFunc(name: String, someLambda: String.()-> Unit){

name.someLambda(); //works

}

called with

someFunc(“yolo”) {

println(this.capitalize())

}

* 1. Inline: Normal functions require their own closure and memory allocations, but sometimes that isn’t necessary for performance reasons, inline works like C++ where the function’s contents are spilled out at the call site . An inline function, will convert normal lambdas passed to it to inlined lambdas. To prevent a lambda from being inlined, used noinline <lambda sign>. G/Setters can be inlined too.

When a function is inlined, lambdas inside it can return, called non-local returns. But when it is not then,

fun foo() {

ordinaryFunction {

return // ERROR: cannot make `foo` return here

}

}

To disallow from a lambda returning in an inlined function, use ‘crossinline’

inline fun foo(crossinline boy: ()-> Unit)

{

}

This is used in tandem with passing ‘boy’ to another execution context such as a nested function. An inline function wouldn’t be able to give its own context and hence boy can’t return in it, crossinline indicates that it won’t, as it would be an error.

* 1. Return: By default, return returns the outermost context, this is why return inside a lambda is prohibited, as it would be returning the parent context when it shouldn’t have the right to do so. To return a given context, we use labeled returns (explained in its own separate point in this doc).

Syntax:

someFunc{

return@someFunc <T type value>

}

Just like a normal return, this return also returns a value.

For ex.:

//Extension function higher order function, turns lambda into an extension function for Int.

private fun abc(lambda: Int.()-> Int) :Int{

return 2.lambda()

}

private fun callThis(){

val someValue= abc{

return@abc this\*2

}

// = 4

val anotherValue=someValue+2

}

Anonymous functions can use normal returns though,

val someValue= abc(fun Int.(): Int{

return this\*2

}) //=4

//works

i.e., anonymous func != lambda as the context is different in both of them

* 1. let: Used to chain function call chains. Just like apply but it returns a copy of the object, this is useful in chaining multiple lets together.

println(abc.let { it.name.capitalize()}

.let{it + "fish"}

.let{it.length}

.let{it + 31})

prints 38.

It is also helpful in working with nullables,

fun aabc(item: Int?) {

item?.let {

//here item is not null.

}

}

* 1. Varargs: Accept variable number of args using vararg.

fun <T> asList(vararg ts: T): List<T> {

val result = ArrayList<T>()

for (t in ts) // ts is an Array

result.add(t)

return result

}

i.e., while the type of variable is T as denoted in the function signature the type of variable becomes List<T> automatically if we use varargs.

* 1. Infix functions: Call functions using the infix notation (no dots/curved brackets).

infix fun Int.shl(x: Int): Int { ... }

1 shl 2

//or

1.shl(2)

They must have 1 param, be ext. func or member func and the param must not be vararg nor should it have a default value. They have lower precedence than arithmetic operators, type casts and rangeTo operator. If used as member func and called in the same class, syntax is

this someMethod “someValue”

//this is needed.

1. Things are imported as in c++.

import java.util.\*;

Here \* implies import everything.

1. Filter: Just like in any other lang, filter returns a new list after running a bool-returning code on it.

val cars= listOf(“lik”,”vroom”);

val newList= cars.filter{ it[0] == c };

‘it’ refers to each element.

This is an eager function.

* 1. Sequence: This collection enables filter to work lazily.

val newList2= cars.asSequence().filter{…}

Naturally, the elements aren’t processed until called. Use ..toList() to squeeze the sequence of lazy elements.

Another use for sequence is in yield,

fun simple(): Sequence<Int> = sequence { // sequence builder

for (i in 1..3) {

Thread.sleep(100) // pretend we are computing it

yield(i) // yield next value

}

}

fun main() {

simple().forEach { value -> println(value) }

}

* 1. map: Just like in dart, applied to a list it executes a code block on every element and returns a value for it.

val newCars= cars.asSequence().map{

println(“map $it”);

it //returns it automatically

}

Since this is a sequence, map won’t work on any element until it is accessed.

map in extension methods can be used directly,

fun List<A>.asB(): List<B> {

return map { //it: A

AasB(it) //method that converts A to B and returns B.

}

}

1. Class: Blueprint for objects, objects are instance of classes. Properties are characterisitcs, like length, width etc.. Methods, aka member functions are what can be done with an object. Interface is a specification that a class can implement. Packages are way to group related code, or be made into a library and be imported.

Works similar to other langs.

* 1. Packages can be declared with package keyword

package xyz;

…

And then imported with

import xyz;

All classes/methods in package class file are imported.

* 1. ‘new’ isn’t needed to create a new instance of a class.
  2. Properties created with var can be edited later, unlike val.
  3. Constructor: Unlike other langs, default constructor isn’t a different method, but declared alongside classname. If base ctor is a parameterized ctor then default unparameterized ctor is disabled.

class Car(length: Int) {

var length: Int = length; //works

}

The getters and setters for props are created automatically.

val car = Car() //wouldn’t work

Props can be declared in the ctor itself,

class Car(var length: Int)

{

…

}

The difference between props and normal variables in ctor is that normal ctor variables are only accessible by props and init block but not methods.

For more code, an ‘init’ block can be used.

class … {

init { … }

}

This block is executed automatically at instance creation, just like a ctor. There can be multiple init blocks and all are ran sequentially. init block has access to ctor vars and obviously props (but only to props declared before them, an init block cannot access class variables declared after them in the class)

The difference between an init block and a ctor in other language is that an init block is ran in the initialization phase whereas a ctor is ran after an initialization phase.

* 1. Secondary Ctor: Allows ctor overloading.

constructor(someProp: Int): this() {

…

}

this() calls the default ctor. It is necessary to call this(). We are delegating call here. This ctor would be used if the instance is created with params matching its signature. The init block is a body for the default ctor and would only be executed whenever default ctor is invoked.

* 1. ctor can be made private, this is done when we are creating factory ctor using companion object. This disables default and base ctor.

class ABC private constructor(val lol: Int) {

companion object {

fun from(value: Int) {

return ABC(value);

}

}

}

and called with

val abc= ABC.from(24);

For unparameterized default ctor,

class LOL private constructor() {

…

}

val abc= ABC()

val lol= LOL()

are both erroneous.

* 1. Getter/Setter: Props can be assigned manual g/setter.

class … {

var volume: Int //can’t use ;

get() {return 2;} // or get() = 2;

set(value) = field = value;

}

Here, field is a pre-defined property that holds the value of the variable and we change it to change the variable’s value. If we instead used, set(value) {volume=value} then it would have been a recursive call.

val props can’t have setters.

* 1. All classes, props, objects, interfaces, ctors, methods, g/setter are public by default in Kt.

Just like C#, the visibility modifiers are public, internal, private, protected.

A private setter is an interesting use-case, it is modifiable only by members of the class.

* 1. Subclass:

class abc (override var length: Int): car(length= length){…}

We use override modifier to override the props

By default, classes can’t be subclassed. They need ‘open’ modifier.

open class …

* 1. Similarly, props of a class can’t be overridden by subclasses by default, however they can be accessed. We need to use ‘open’ on them to allow so.

open class (open var abc: Int) {….}

* 1. Abstract Class: Can have ctor and are always open. Props and methods are non-abstract unless marked with ‘abstract’, meaning subclass can use them as is. But if they are marked abstract, they must implement (override) them.

abstract class Abc {

…

}

* 1. interface: Just like abstract class but can’t have ctor and can’t store state, meaning props and methods can not be initialized.

interface EEE {

…

}

* 1. Kt doesn’t support multiple inheritance, but multiple interfaces can be implemented. Abstract classes can also implement interfaces.
  2. Object class: Singleton. There can only be a single instance of this class.

object someName {…}

works like a class but it can’t be instantiated and all props need to be initialized with the declaration.

This object isn’t initialized until the first time it is accessed.

One use of this is in defining static classes, unlike normal static classes in other langs like C#, we don’t have static classes in Kotlin, so we can use this to do the same

object someName{

const val home = "home"

const val plant\_detail = "plant\_detail"

}

* 1. object: : We can instantiate an object that implements an interface directly, to use it

val something= object: AnInterface {

override fun aMethod() {…}

}

‘something’ is an instantiation of an interface, which is illegal if done directly, but object: allows us to create an instantiation directly.

It can be used to instantiate other classes that are inheritable as well.

* 1. Function object: Function is an interface too, hence we can use the object: to define a custom Function.

For example:

val aFunc= object: Function1<String, Unit> {

override fun invoke(name: String)

{

…

}

}

//call aFunc(<string>);

This adds a bit of overhead, this is why inlining functions helps with performance as the contents are unpacked right there and this extra code isn’t generated.

* 1. Interface Delegation: In Kt, an interface can be implemented by another class and then this class can be used to provide implementation of the interface rather than using the interface directly and implementing it with new values. It is better to be done by using an object class,

interface Speed {

val speedInKm: Double;

}

object car : Speed {

override val speedInKm = 20.9;

}

class Race : Speed by car {

…

}

Here Speed will be implemented and its props/methods will be overridden as in car.

Another way of doing it is, allowing the implementation to be provided at instance creation of Race.

class Race(someVar: Speed = car): Speed by someVar {…}

object class isn’t necessary for interface delegation.

class bike (override val speedInKm) : Speed {…}

class Race: Speed by bike(20) {…}

* 1. data class: Like structs, but still an object.

data class someName {…}

They automatically implement equals method and copy() method. However, both methods only reference the variables declared in primary ctor.

* 1. sealed class: These classes can only be subclassed in the same file, nowhere else.
  2. companion object: It’s a singleton object within a class. Initialized whenever an object is created, created from the static ctor of the class.

class AB  
{

companion object {

const val abc= 2;

}

}

We can just get the companion object with

val obj=AB.Companion;

They can be used to create const vals for the class as const val can’t be in a class directly.

Companion objects can inherit and instantiate other classes/interfaces, this is because while they look and act like static members of the class they are still instance members of objects of the class.

class PP

{

companion object ABC: xyz() {

…

}

}

Since this is an object it can’t be invoked and the methods inside it are created when the class is.

To access ABC’s methods,

PP.ABC.someMethod();

or

PP.someMethod();

PP can also be written as,

class PP

{

companion object : xyz<someClass>() {

…

}

}

and access it with PP.someMethod().

or

val pp: xyz<someClass>= PP //works.

Since companion objects are a part of instance as well, class methods can access them. They can also access private members of the companion object.

* 1. Just like in C++, ‘this’ keyword can be used to access members inside a class.
  2. Delegated properties: Properties in a class can be implemented each time by subclasses or we can delegate them to other methods and only implement them once. To do so we use delegated properties,

Syntax:

class abc {

val/var <propname>: <proptype> by <expr>

}

expr is a delegate, ‘by’ forwards all g/set calls to the object returned by the method, namely it needs a getValue and setValue method in the delegate.

For example:

class xyz{

var p: String by MyDelegate()

}

class MyDelegate() {

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.")

}

}

Here, thisRef holds the object we read p from, so an instance of xyz and property holds the description of p. getValue just needs to return a string which represents p, it also means that ‘p’s value is nothing in the getValue, if we want to use a backing property then that is up to us. If it were val we wouldn’t need to define setValue.  
  
Yes, it uses reflection for the same.

Standard delegates: Kt’s STL provides many delegates we can use instead of creating our own ones.

* + 1. lazy: A method that accepts a lambda which is run only once and its result is stored on the first get call on the object, returns a Lazy<T>.

val p: String by lazy {

println(“Hello”)

“Yo”

}

println(p)

println(p)

prints

Hello

Yo

Yo

We can also use lazy(LazyThreadSafetyMode.<a>), by default ‘a’ is synchronized which means the value is ccomputed in a single thread and all other threads see the same value, it can also be publication which allows each thread its own copy or none if threads aren’t being used.

* + 1. Delegates.observable(): Takes 2 args, an initial value and a lambda handler. The lambda handler takes 3 params, the property info param, old value and new value, and is called after a new value is assigned to the property. It observes ‘set’ operations.

class User {

var name: String by Delegates.observable("yaba baba") {

prop, old, new ->

println("$old -> $new with ${prop.name}")

}

}

will set the initial value to yaba baba and when we set new values on the prop it will print the given string after assignment.

* + 1. Delegates.vetable(): Same as observable but runs the handler before assignment.
    2. Delegating to another property: This is done by using the ‘::’ operator and can be used to delegate g/set to another prop which can be in its own class or be global.

var topLevelInt: Int = 0

class ClassWithDelegate(val anotherClassInt: Int)

class MyClass(var memberInt: Int, val anotherClassInstance: ClassWithDelegate) {

var delegatedToMember: Int by this::memberInt

var delegatedToTopLevel: Int by ::topLevelInt

val delegatedToAnotherClass: Int by anotherClassInstance::anotherClassInt

}

var MyClass.extDelegated: Int by ::topLevelInt

* + 1. map: Map/MutableMap can also be used like delegates.

class User(val map: Map<String, Any?>) {

val name: String by map

val age: Int by map

}

and set values using

val user = User(mapOf(

"name" to "John Doe",

"age" to 25

))

The property name is the key it searches for in the map object and assigns the value associated with the key to the property.

* + 1. Local delegated props: Local vars can have delegated props too, like

fun example(computeFoo: () -> Foo) {

val memoizedFoo by lazy(computeFoo)

if (someCondition && memoizedFoo.isValid()) {

memoizedFoo.doSomething()

}

}

* + 1. We can create custom delegates without classes using interface ReadOnlyProperty or ReadWriteProperty.

fun resourceDelegate(): ReadWriteProperty<Any?, Int> =

object : ReadWriteProperty<Any?, Int> {

var curValue = 0

override fun getValue(thisRef: Any?, property: KProperty<\*>): Int = curValue

override fun setValue(thisRef: Any?, property: KProperty<\*>, value: Int) {

curValue = value

}

}

val readOnly: Int by resourceDelegate() // ReadWriteProperty as val

var readWrite: Int by resourceDelegate()---

This also shows us that functions can be used as delegates. Taking it a step further,

* + 1. ProviderDelegate: By using this type of delegate, the logic for creating the object can be delegated to yet another delegate. It is automatically used if the delegate object defines provideDelegate method with same params as getValue.

For example:

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 delegate

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)

}

This can be streamlined by using PropertyDelegateProvider from the STL,

val provider = PropertyDelegateProvider { thisRef: Any?, property ->

ReadOnlyProperty<Any?, Int> {\_, property -> 42 }

}

val delegate: Int by provider

* 1. Inner class: Classes can nest other classes, interfaces or the other way around or any combination. But if we declare the nested class as inner class then it can access the members of the outer class.

class Outer {

private val bar: Int = 1

inner class Inner {

fun foo() = bar

}

}

val demo = Outer().Inner().foo()

* + 1. They can be declared anonymously as well, we use object expression for the same.

window.addMouseListener(object : MouseAdapter() {

override fun mouseClicked(e: MouseEvent) { ... }

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

})

1. SAM: Single Abstract Methods. An interface with exactly 1 method in it, it’s common for Java based APIs. Runnable with a method run() and Callable with a method call() are 2 common SAMs.

To use them,

<java file>

package example;

public class JavaRun {

public static void runNow(Runnable runnable) {

runnable.run();

}

}

<kt file>

fun runExample() {

val runnable = object: Runnable {

override fun run() {

println("I'm a Runnable")

}

JavaRun.runNow(runnable);

}

runs the above instantiation in runNow.

It can be also written as,

JavaRun.runNow({…});

//or

JavaRun.runNow {…}

Kotlin provides this ease to assist in working with SAMs.

* 1. SAMs are aka functional interfaces. They can have many non-abstract members but only 1 abstract member.

To declare one in KT,

fun interface IntPredicate {

fun accept(i: Int): Boolean

}

and we instantiate it normally using object,

val isEven = object : IntPredicate {

override fun accept(i: Int): Boolean {

return i % 2 == 0

}

}

or

val isEven = IntPredicate { it % 2 == 0 }

This uses kotlin’s SAM conversion. We just have to provide a method body in a lambda.

1. Structural Equality: ==, compares the values of 2 objects, or calls the equals method in class for complex types. equals(other: Any?): Boolean {…}
2. Referential Equality: ===, compares the references of 2 objects. When comparing obj to null, == is automatically inferred as === and for primitive types, === is automatically inferred as ==.
3. in: Used to check if value is in the given range, x in a . . b.
4. is: Used to check types.
5. with: Used to create a higher-order function easily.

data class Abc(val name: String)

val abc= Abc(“moo”);

with(abc.name)

{

println(this.capitalize());

}

//prints “Moo”

//It is similar to this func

fun myWith(name: String, block: String.() -> Unit) {

name.block()

}

myWith(abc.name) {

println(this.capitalize());

}

//Here, String.() accepts a normal lambda but turns it into a String extension function.

1. run: Run takes a lambda and returns the result of the lambda.

val res= abc.run {

name

}

returns name.

1. apply(): Just like run but applies changes onto the object and returns the object.

val res= Abc(“moo”).apply{

name= “yaka”

}

1. Destructuring: Same as in c#.

data class abc {

val a:String=”yo”;

val b: Int= 2;

val c= “da”;

}

val décor= abc();

val (a,b,c) = décor;

val (d,\_,f)= décor ; //skip b

1. enum: A bit more advanced in Kt than other languages.

enum class Direction(val degrees: Int) {

NORTH(0), SOUTH(180), EAST(90), WEST(270)

}

fun main() {

println(Direction.EAST.name)

println(Direction.EAST.ordinal)

println(Direction.EAST.degrees)

}

prints

EAST

2

90

ordinal is the index.

1. Extensions: Just like in dart, extensions are used to extend capabilities of a class.

fun String.hasSpaces(): Boolean {

val found= this.find {it == ‘ ‘};

return found;

}

//or

fun String.hasSpaces()= find {it == ‘ ‘} !=null;

‘this’ isn’t needed inside extension functions and all non-private methods/props can be accessed normally.

Ext. funcs are resolved at compile time. Meaning,

open class AquariumPlant(val color: String, private val size: Int)

class GreenLeafyPlant(size: Int) : AquariumPlant("green", size)

fun AquariumPlant.print() = println("AquariumPlant")

fun GreenLeafyPlant.print() = println("GreenLeafyPlant")

val plant = GreenLeafyPlant(size = 10)

plant.print()

println("\n")

val aquariumPlant: AquariumPlant = plant

aquariumPlant.print()

prints GreenLeafyPlant then AquariumPlant, i.e., aquariumPlant’s type is assigned at compile time to be AquariumPlant and hence it uses the extension method defined for it, instead of being assigned a different func at runtime (instances/objects are created at runtime hence plant isn’t initialized until later)

* 1. Extension Property: Just like functions, ext can be applied to props too, along with g/setter. They work just like normal props in a class.

val Car.boom: Boolean

get() = speedInKm==20;

* 1. Nullable receiver: The class being extended is called the receiver and it an be null, T?. We can define ext methods for them as well,

fun Car?.wow() {

…

}

We’ll use ‘this?’ here.

val car: Car?=null;

car.wow(); //executes the ext

car?.wow(); //doesn’t execute the ext

1. Generics: Just like everywhere else.

fun <T>abc(a:Int ): T {…}

T: type to specify type that can be used. By default it is Any? so T can be null too.

abc<Int>(); //to call it.

* 1. Covariance and Contravariance: Just like in C#, co means given class or base classes, contra means given class or its child classes.

We use ‘in’ for co and ‘out’ for contra here as well.

class ABC<in T: Car> {…}

Variance can only be used with classes and interfaces.

* 1. Generic types are replaced at compile time hence we cannot check if a generic type is <user-defined type> at compile time. It is an error as the generic types are erased, called Type Erasure. To counter this, we use inline function with reified types.

inline fun <reified T: Car> wow() = Car is T; //works

Reified tells the compiler the type will be generated at runtime and hence it should be assumed as a real type. Only inline functions can have reified types.

* 1. Star Projections: When we use a class/interface that uses co/contravariance with generics, it defines an upper bound/lower bound, i.e., this is most base class (Any?) or most child class (Nothing).

If we don’t define this limit, <in T> then any consumer that accepts an object of this class will not be able to access the methods/props of T provided through the class since it doesn’t know what they are, hence we use <in T: Car>.

At the consumer site, if the class is being provided another generic then we can directly use <in K: Car> but it is not always possible to know which type it is that T needs. Here, we can use ‘ \* ’, which ensures the type will be in T or out T with limit same as the class. For classes with no limits in T: Any? and out T: Nothing is used.

So there are 2 imp points here, have limits on variance and use \* to ease up boilerplate.

For example:

class Producer<out T> {

private val items = listOf<T>()

fun produce() : T = items.last()

fun size(): Int = items.size

}

fun useProducer(star: Producer<\*>) {

// Produces Any?, a Pet is not guaranteed because T is unknown

val anyNullable = star.produce() // Not useful

// Can't use functions and properties of Pet.

anyNullable.cutenessIndex // Error

// Can use T-independent functions and properties

star.size() // OK

}

but

class Producer<out T : Pet> {

private val pets = listOf<T>()

fun produce() : T = pets.last()

}

fun useProducer(star: Producer<\*>) {

// Even though we use \* here, T is known to be at least a Pet

// because it's an upper bound at the declaration site.

// So, Pet is guaranteed.

val pet = star.produce() // OK

// Can use properties and functions of Pet.

pet.cutenessIndex // OK

}

1. check(<bool>); Like assert in other langs.
2. Annotations: Just like other langs, use these to define some property about a class/method. Annotations are based on reflection so they decrease perf.

For example:

@file:JvmName("InteropFish")

class InteropFish {

companion object {

@JvmStatic fun interop()

}

}

JvmName gives the name of the file as InteropFish and JvmStatic turns the method static.

* 1. To create annotation:

annotation class abc

{

@Target(AnnotationTarget.PROPERTY\_GETTER)

annotation class OnGet

@Target(AnnotationTarget.PROPERTY\_SETTER)

annotation class OnSet

}

To use it:

@abc class bbc {

@get: OnGet

val …

@set: OnSet

val …

}

Applies the given annotation class and it’s annotations on these members.

1. Reflection: Check properties of objects at runtime.
   1. ::class :

import kotlin.reflect.full.\*

class someClass{…}

val reflectedObj= someClass::class

val aList= reflectedObj.declaredMemberFunctions

…

::class gets class props/methods and their details and similarly declaredMemberFunctions gets details about methods.

* 1. reflectedObj.annotations gets annotations applied on a class. Or ..findAnnotation<type>() to get one directly.

1. Labeled Break: Kt has break, continue return and also a ‘goto’ replacement.

outerLoop@ for(x in 1..10)

{

for(j in 20..30)

{

break@outerLoop

}

}

breaks the loops and returns control to after the loop.

Similarly, labeled continue and labeled return exists, labeled return returns the given lambdas control. Unlike goto these can’t be used everywhere except in loops.

1. Dependencies: Dependencies in kotlin are provided by Gradle, maven, or through Gradle Kotlin DSL. These are build tools and unlike many other languages, we can choose multiple of these to build a kotlin project. A sample dependency looks like

org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0

And to add it we use,

Maven

<dependency>

<groupId>org.jetbrains.kotlinx</groupId>

<artifactId>kotlinx-coroutines-core</artifactId>

<version>1.6.0</version>

</dependency>

Gradle

dependencies {

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'

}

Gradle Kotlin DSL

dependencies {

implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0")

}

1. Exception: throw Exception(“someMessage”); to throw it,

try{…}

catch(e: SomeException)

{…}

finally {…}

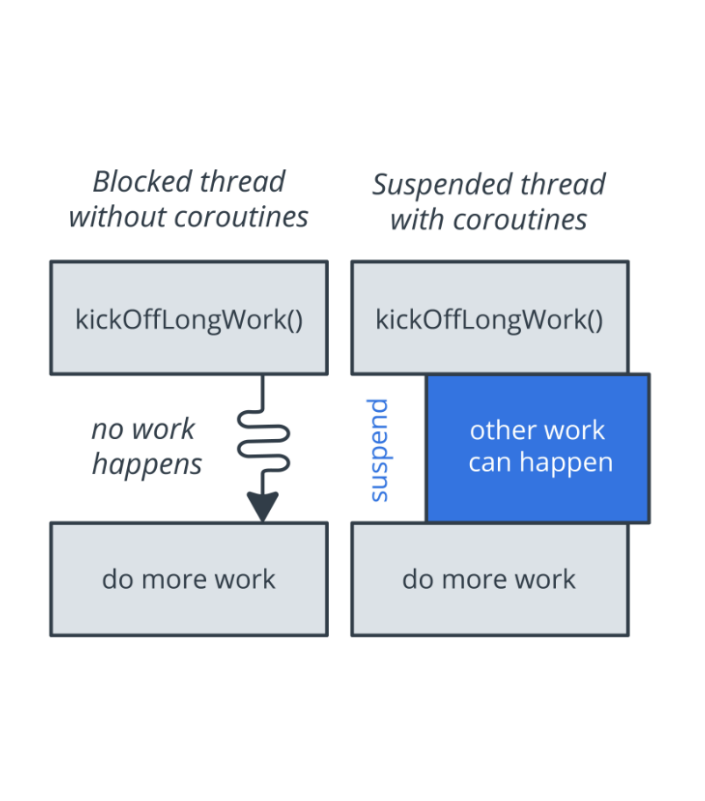
to catch it. Can have multiple catches.

* 1. Just like everything else, try catch is an expr too. It returns the last expr of either try block or of last catch block.

val abc: Int = try { xyz.toInt() } catch (e: Any) { 10 }

* 1. @Throws can be used in Java when calling Kt code in it since Kt doesn’t have checked exceptions (forced exceptions).

1. Concurrency: Kotlin has its own way of working with async code, unlike traditional threading, callbacks, Future/Promises or Rx, called Coroutines. Coroutines in Kt simplify async programming to a higher level, there’s very little change in syntaxes and normal blocking code can easily be made async.



* 1. Coroutines aren’t packaged with Kotlin, they need to be added separately. From the org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0 package.
  2. Sample Code:

import kotlinx.coroutines.\*  
  
fun main() = *runBlocking* **{** *launch* **{** delay(4000);  
 *println*("wooo");  
 **}** *launch* **{** delay(5000)  
 *println*("nooo")  
 **}** return@runBlocking;  
**}**

runBlocking takes a lambda which can declare multiple coroutines using launch, launch takes a lambda that will be executed asynchronously. All coroutines are executed concurrently inside runBlocking. After all the coroutines are completed, runBlocking exits. It is only needed in main since it blocks control until all the coroutines inside are finished processing.

* 1. suspend fun: Just like Future in dart, these functions allow async code like delay(…). They can be passed to launch directly. For example:

suspend fun abc(){…}

fun main = …

launch{abc()}

* 1. coroutinescope: This is the heart of the runBlocking function, it returns a suspend function and releases underlying threads until control is needed again. runBlocking uses it and returns a kotlin.Unit. It can be used directly as well, to build our own scope.

fun main = runBlocking {

abc()

}

suspend fun abc() = coroutineScope {

launch {…}

return@coroutineScope;

}

* 1. CoroutineScope can be created using either coroutineScope or MainScope(). The latter uses Dispatchers.Main and is used for UI of an app.

The recommended way of using scopes is:

class Activity {

private val mainScope = MainScope()

fun destroy() {

mainScope.cancel()

}

}

then use mainScope.launch {…} everywhere to launch coroutines, when the Activity is to be destroyed, simply call destroy().   
This bundles all coroutines into a single scope and hence managing lifecycle of coroutines is simple.

* 1. Explicit Job: We can use ‘launch’ coroutine builder explicitly as well.

For example:

fun main() = runBlocking {

abc();

val abx = launch {

delay(1000);

println("yo");

}

println(“yee”);

abc.join();

}

.join waits until the coroutine is complete.

* 1. Cancellation: The job returned by launch can be cancelled as well, using abc.cancel(), then calling join to let the coroutine be stopped completely. Or call cancelAndJoin() to do the same. However, for a coroutine to be cancellable it must either suspended or manually check for cancellation. In an infinite while loop it will keep executing hence even if we call cancel it will only cancel once it suspends/exits the loop.
  2. To check for cancellation manually, use ‘isActive’, it returns a bool which is false if cancellation is called. Only available in a coroutinescope.
  3. When cancellation is called, CancellationException is thrown by the function as soon as it acknowledges the cancellation. We can leave it as-is or use try catch inside the function. The catch/finally is executed after the cancellation.
  4. Non-cancellable block: The catch/finally can’t have another suspended fun inside it as the coroutine is cancelled, it will throw the same exception again. But we can allow suspended func inside it if we use withContext(NonCancellable) {…}

val job = launch {

try {

repeat(1000) { i ->

println("job: I'm sleeping $i ...")

delay(500L)

}

} finally {

withContext(NonCancellable) {

println("job: I'm running finally")

delay(1000L)

println("job: And I've just delayed for 1 sec because I'm non-cancellable")

}

}

}

delay(1300L) // delay a bit

println("main: I'm tired of waiting!")

job.cancelAndJoin() // cancels the job and waits for its completion

println("main: Now I can quit.")

* 1. Timeout: We can use withTimeout(<duration>){…} inside a coroutine scope to let a computation execute with a timeout.

fun main() = runBlocking {

withTimeout(1000L)

{…}

}

When timeout is reached the coroutine throws a TimeoutCancellationException, which is a subtype of CancellationException. If a withTimeout is used inside another coroutine then it’s exception will be caught by the coroutine as CancellationException automatically and we won’t see it bubble outside.

* 1. withTimeoutOrNull: Returns null instead of an exception when timeout is reached.

val abc= withTimeoutOrNull(…){…}

* 1. withTimeout runs the timer event asynchronously to the actual code inside the block, hence even if it appears timeout isn’t reached, it is possible it may fire off and exception be raised. If any resource is acquired in a withTimeout, be sure to wrap the coroutine in try finally and release it on finally.

try {

withTimeout(60) { // Timeout of 60 ms

delay(50) // Delay for 50 ms

resource = Resource() // Store a resource to the variable if acquired

}

} finally {

resource?.close() // Release the resource if it was acquired

}

* 1. Sequential: Suspending functions inside a coroutine are executed sequentially by default.

val time = measureTimeMillis {

val one = doSomethingUsefulOne()

val two = doSomethingUsefulTwo()

println("The answer is ${one + two}")

}

println("Completed in $time ms")

will give the time taken by both.

* 1. Async: We can make the suspending functions async by using this keyword.

val time = measureTimeMillis {

val one = async { doSomethingUsefulOne() }

val two = async { doSomethingUsefulTwo() }

println("The answer is ${one.await() + two.await()}")

}

println("Completed in $time ms")

will give the time taken by the longer suspended function.

Async is just like a launch, but it returns a Deferred instead of a Job. Deferred is exactly like Future in Dart, it actually returns a Future which we await to consume. Since Deferred is a subtype of a Job it also has cancel.

* 1. Lazy async: By default async starts the execution right away and await is equivalent to join. But it can be made lazy, i.e., only start when asked or awaited.

val time = measureTimeMillis {

val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }

val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }

// some computation

one.start() // start the first one

two.start() // start the second one

println("The answer is ${one.await() + two.await()}")

}

println("Completed in $time ms")

If we don’t use start in this case, await starts the exec but blocks until it finishes and hence the behavior is almost akin to normal sequential exec.

* 1. Async style functions can easily reduce all this code but their use is discouraged in Kt. <https://kotlinlang.org/docs/composing-suspending-functions.html#async-style-functions>
  2. CoroutineContext: All coroutines execute in a context and the context is defined by this type, it has many elements but Job and Dispatcher are the main ones. Job is a cancellable background task which is returned by launch and async.

All coroutines in a context can be cancelled by,

coroutineContext.cancelChildren();

* 1. Dispatcher: Dispatcher defines which thread will the coroutine work on. By default, launch {…} takes the parent coroutine which runBlocking provides.

launch { // context of the parent, main runBlocking coroutine

println("main runBlocking: ${Thread.currentThread().name}")

}

launch(Dispatchers.Unconfined) { // not confined -- will work with main thread

println("Unconfined: ${Thread.currentThread().name}")

}

launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher

println("Default ${Thread.currentThread().name}")

}

launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread

println("newSingleThreadContext:${Thread.currentThread().name}")

}

Unconfined: Run the code in the caller thread until first suspend function, then switch to a different thread for consuming the suspend function and the rest of the body.

newSingleThreadContext(‘name’): Runs the coroutine in a user named thread, it can even be used to give a different thread to runBlocking and withContext.

Dispatchers.Default: Runs the coroutine in a thread taken from a shared pool of threads. This is different from not assigning a dispatcher, as then, the parent thread is used.

Dispatchers.IO: Available in Android, uses a separate pool of threads meant to be used for IO bound tasks.

* 1. Debugging Coroutines: <https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html#debugging-coroutines-and-threads>

log(‘message’);

prints the thread and message.

* 1. We can get Job status using coroutineContext.

fun main() = runBlocking<Unit> {

println("My job is ${coroutineContext[Job]}")

}

* 1. When a coroutine is launched in the CoroutineScope of another coroutine, its Job becomes a child of the parent coroutine and hence if we cancel parent job it gets cancelled too. This also means that the parent coroutine isn’t complete until all its children are.

GlobalScope.launch;

can be used to prevent this.

launch(Job()) {…}

does the same, as a new Job is passed to it instead of getting inherited from the parent.

* 1. CoroutineName: Assign custom names to coroutines.

async(CoroutineName(“yee”)){…}

* 1. We can define multiple context elements with +.

launch(Dispatchers.Default + CoroutineName("test")){…}

* 1. ThreadLocal: Sharing data across coroutines isn’t simple, but Kt provides ThreadLocal to alleviate the issue.

import kotlinx.coroutines.\*  
  
val *threadLocal* = ThreadLocal<String?>() // declare thread-local variable  
  
fun main() = *runBlocking*<Unit> **{** *threadLocal*.set("main")  
 *println*("Pre-main, current thread: ${Thread.currentThread()}, thread local value: '${*threadLocal*.get()}'")  
 val job = *launch*(Dispatchers.Default + *threadLocal*.*asContextElement*(value = "launch")) **{** *println*("Launch start, current thread: ${Thread.currentThread()}, thread local value: '${*threadLocal*.get()}'")  
 yield()  
 *println*("After yield, current thread: ${Thread.currentThread()}, thread local value: '${*threadLocal*.get()}'")  
 **}** job.join()  
 *println*("Post-main, current thread: ${Thread.currentThread()}, thread local value: '${*threadLocal*.get()}'")  
**}**

The value is main in the parent coroutine (even in post-main) but in job’s coroutines it is launch across all of them. If value is set inside a child coroutine then it isn’t propagated outside. A way to counter it is, use withContext with parent coroutine’s context and mutate it in there and then all the child coroutines will observe the changed value.

* 1. Flow: Just like Stream in dart.

fun simple(): Flow<Int> = flow { // flow builder

for (i in 1..3) {

delay(100) // pretend we are doing something useful here

emit(i) // emit next value

}

}

fun main() = runBlocking<Unit> {

// Collect the flow

simple().collect { value -> println(value) }

//or (1..4).asFlow().collect {…}

//or val flowTest= flowOf (1,2,3)

}

* + 1. Flow doesn’t start until it is collected. It respects cancellation just like normal coroutines. As such, an exception is thrown inside the flow when cancellation is requested.
    2. Flow can use map or filter and these take a lambda which can call suspend functions. Flop.map.collect{…}; applies map to an upstream flow and collect brings the downstream,

i.e, the execution is sequential.

(1..5).asFlow()

.filter {

println("Filter $it")

it % 2 == 0

}

.map {

println("Map $it")

"string $it"

}.collect {

println("Collect $it")

}

prints:

Filter 1

Filter 2

Map 2

Collect string 2

Filter 3

Filter 4

Map 4

Collect string 4

Filter 5

* + 1. Flow can also use ..transform on its upstream to run a function on each value, it can imitate map/filter but be used for other complex ops too.
    2. <flow>.take(<int>).collect{..}; takes given elements then sends a cancellation.
    3. Flow can use methods like collect, reduce, first, toList, fold toSet. These are called Terminal flow operators and one of them is needed to start flow collection.

launchIn(<coroutine>); // or use launchIn(this) for current coroutine

is a terminal operator that launches the collection on the given coroutine.

Terminal operators can use cancel() func inside to cancel a flow. But it will wait until the flow acknowledges cancellation. To explicitly check cancellation,

flow.onEach{currentCoroutineContext().ensureActive()}.collect{…}

//or flow.cancellable()

can be used

* + 1. Flow context: Flows are executed in the caller’s context,

withContext(context) {

simple().collect { value ->

println(value) // run in the specified context

}

}

will run simple and collect in this context.

simple…{

kotlinx.coroutines.withContext(Dispatchers.Default) {

for (i in 1..3) {

Thread.sleep(100) // pretend we are computing it in CPU-consuming way

emit(i) // emit next value

}

}

}

gives an error because the body tried emitting in a different context.

* + 1. flowOn: This function can be used on the flow body to change the context of the flow.

simple…= flow {…}.flowOn(Dispatchers.Default);

flowOn also applies .buffer automatically.

* + 1. buffer: Runs collection on each element concurrently as opposed to sequentially which is default.

simple().buffer().collect{…}

* + 1. conflation: Conflate is synonym to merge. When we apply this operator, only the first element and then the most recent elements are collected.

val flow = flow {

for (i in 1..30) {

delay(100)

emit(i)

}

}

flow.conflate().collect {value=> delay(1000);

println(value);

}

will print 1,10,20,30

* + 1. collectLatest or <terminal-operator>Latest: collects all values, processes them and the last element is the only one collected.
    2. flow.Zip(anotherFlow) {a,b-> <return val>}.collect{…} combines 2 flows. The time taken by each element is equal to the max time taken between the 2 flow’s emission.

val nums = (1..3).asFlow().onEach { delay(300) } // numbers 1..3 every 300 ms

val strs = flowOf("one", "two", "three").onEach { delay(400) } // strings every 400 ms

val startTime = System.currentTimeMillis() // remember the start time

nums.zip(strs) { a, b -> "$a -> $b" } // compose a single string with "zip"

.collect { value -> // collect and print

println("$value at ${System.currentTimeMillis() - startTime} ms from start")

}

1 -> one at 449 ms from start

2 -> two at 849 ms from start

3 -> three at 1251 ms from start

flow.combine is the same as zip except it runs on each emission, of either of the flows, so the result can include multiple elements.

* + 1. We can combine flows, as in take ones processing into another and then flatten it using any of the flatten operators.
    2. Exception: All exceptions can be caught by a single try catch on the collector, or by using flow.catch{elem => …}.collect{…}. Only upstream exceptions are caught using .catch, not the ones in collect. To workaround this, flow.onEach{…}.catch{…}.collect(); can be used. There’s onCompletion {catch->…} as well which is always ran after the flow is completed, it gets a null value if there was no exception.
    3. SharedFlow: A hot flow that shares emitted values to all its collectors in a broadcast fashion. Kinda like an Observable in any language, It’s a hot flow, as it exists independently of its collectors and can run without any collector unlike a normal flow, which is called cold. It never completes, calling Flow.collect on it or Flow.launchIn never completes normally, any collector is called a subscriber which can be cancelled and hence complete. Similarly, Flow.toList wouldn’t complete, but ..take and ..takeWhile would.

For ex:

class EventBus {

private val \_events = MutableSharedFlow<Event>() // private mutable shared flow

val events = \_events.asSharedFlow() // publicly exposed as read-only shared flow

suspend fun produceEvent(event: Event) {

\_events.emit(event) // suspends until all subscribers receive it

}

}

It will give all the events to any subscriber, the limit is defined in its replay cache and buffer.

* + 1. StateFlow: An implementation of SharedFlow which is faster and doesn’t store all the events, only the most recent/an initial value which it provides to new subscribers.
  1. Channel: Just like in go, send and receive values asynchronously from a channel.

val channel= Channel<Int>()

launch{

for (x in 1..5) channel.send(x);

}

repeat(5) {println(channel.recieve)};

//or

val channel…

launch {

for (x in 1..5) channel.send(x);

channel.close(); //to indicate no more elems are coming in.

}

for (y in channel) {…}

Channel<T> returns a SendChannel<T> object.

* + 1. Producer-Consumer: This can be achieved using a few convienently named features for channels.

import kotlinx.coroutines.\*

import kotlinx.coroutines.channels.\*

fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {

for (x in 1..5) send(x \* x)

}

fun main() = runBlocking {

val squares = produceSquares()

squares.consumeEach { println(it) }

println("Done!")

}

produce returns a RecieveChannel which can be used to consumeEach or receive() to return a single value. RecieveChannel can be passed around as one end of the channel to build complex multi-stage channels. We define producers as ext to CoroutineScope to easily stop all of them with coroutineContext.cancelChildren();

Produce is ran asynchronously, we can do the same with iterators with yield, next and Iterator but they aren’t asynchronous.

* + 1. Fan-out and Fan-In: We can pass around SendChannel and ReceiveChannel to build fan-out (1 sender multiple recievers )or fan-in (the opp.) channels.

Fan-out:

import kotlinx.coroutines.\*  
import kotlinx.coroutines.channels.\*  
  
fun main() = *runBlocking*<Unit> **{** val producer = *produceNumbers*()  
 *repeat*(5) **{** *launchProcessor*(**it**, producer) **}** delay(950)  
 producer.cancel() // cancel producer coroutine and thus kill them all  
**}**fun CoroutineScope.produceNumbers() = *produce*<Int> **{** var x = 1 // start from 1  
 while (true) {  
 send(x++) // produce next  
 delay(100) // wait 0.1s  
 }  
**}**fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = *launch* **{** for (msg in channel) {  
 *println*("Processor #$id received $msg")  
 }  
**}**

Fan-in:

import kotlinx.coroutines.\*  
import kotlinx.coroutines.channels.\*  
  
fun main() = *runBlocking* **{** val channel = *Channel*<String>()  
 *launch* **{** sendString(channel, "foo", 200L) **}** *launch* **{** sendString(channel, "BAR!", 500L) **}** *repeat*(6) **{** // receive first six  
 *println*(channel.receive())  
 **}** coroutineContext.*cancelChildren*() // cancel all children to let main finish  
**}**suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {  
 while (true) {  
 delay(time)  
 channel.send(s)  
 }  
}

Channels are fair, so the first one to request gets the first value and first one to send sends the first value from/to a channel.

* + 1. Buffered Channel: By default, when a sender sends or a receiver receives a message in the channel, they then wait until the other acknowledges processing it and suspends until then. But we can define a capacity to let it continue adding more elems or requesting more elems to create a buffer.

val channel = Channel<Int>(4) // create buffered channel

…

* + 1. Ticker Channel: Produces a kotlin.Unit in given delay repeatedly until cancelled.

**val** tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0) // create ticker channel

**var** nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }

println(**"Initial element is available immediately: $nextElement"**) // no initial delay

nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent elements have 100ms delay

println(**"Next element is not ready in 50 ms: $nextElement"**)

tickerChannel.cancel()

prints kotlin.Unit and then null.

* 1. Exceptions: They can be handled in a few more ways in Coroutines.

In the root coroutine scope, launch/actor propagate exceptions automatically while async/produce spill the exceptions outside.

* + 1. A root coroutine can be made with GlobalScope. GlobalScope’s use isn’t recommended hence we have to explicitly opt-in to use it.

@OptIn(DelicateCoroutinesApi::class)

fun main() = runBlocking {

val job = GlobalScope.launch { // root coroutine with launch

println("Throwing exception from launch")

throw IndexOutOfBoundsException() // Will be printed to the console by Thread.defaultUncaughtExceptionHandler

}

}

* + 1. Global Exception Handler: CoroutineExceptionHandler is used for this task, normally there’s one defined by android/jvm but we can define our own.

import kotlinx.coroutines.\*

@OptIn(DelicateCoroutinesApi::class)

fun main() = runBlocking {

val handler = CoroutineExceptionHandler { \_, exception ->

println("CoroutineExceptionHandler got $exception")

}

val job = GlobalScope.launch(handler) { // root coroutine, running in GlobalScope

throw AssertionError()

}

val deferred = GlobalScope.async(handler) { // also root, but async instead of launch

throw ArithmeticException() // Nothing will be printed, relying on user to call deferred.await()

}

joinAll(job, deferred)

}

The CoroutineExceptionHandler is called at the last, normally, when an exception occurs in a coroutine it bubbles up until one handles it, if none handles it then the CoroutineExceptionHandler is invoked. It’s not invoked for async builder as async passes it to its deferred object.

* + 1. CancellationException isn’t caught by any default handler, and it has no use other than for debugging purposes.
    2. Aggregate Exceptions: All exceptions in coroutines while bubbling up are bundled together, the first to be caught wins and the rest of the exceptions are added to it in a list called suppressed exceptions.
    3. Supervision Job: Just like a normal Job, but when an exception occurs in it, it is propagated downwards to its children.

val supervisor = SupervisorJob()

with(CoroutineScope(coroutineContext + supervisor)) {

val firstChild = launch(CoroutineExceptionHandler { \_, \_ -> }) {

println("The first child is failing")

throw AssertionError("The first child is cancelled")

}

firstChild.join();

supervisor.cancel(); //to cancel it

* + 1. SupervisorScope: Just like normal coroutineScope but when a cancellation is requested, it cancels only its children and cancels all of them only if it failed itself.

The children in this scope must handle their own exceptions and can use CoroutineExceptionHandler because their exceptions don’t bubble up.

* 1. Shared Mutable State: Normal variables and volatile variables don’t help when lots of coroutines have to modify the same value. These are the possible ways to get a SMS.
     1. AtomicInteger(): This type can be passed to the coroutines and it’s thread-safe. It provides an .incromentAndGet() method which is like <var>++ but fully thread-safe.
     2. Thread-confinement: Create a newSingleThreadContext and with withContext let all the coroutines add in the same context. This is quite slow because of the context switching but it ensures the thread is confined to a context.
     3. Mutex: Just like in other langs, locks the variable then unlocks access to it.

val mutex= Mutex()

mutex.lock(();

try {…}

finally {mutex.unlock();}

//or

mutex.withLock {…}

This mutex is passed to all the coroutines.

* + 1. actor: This creates a coroutineScope that provides a channel, state and is better suited for complex classes.
  1. Raw Thread: We can use the Thread class from java.util.\*.

val thread= Thread {

Thread.sleep(50);

…

}

thread.apply {start()}; //start exec

thread.join(); //join main thread

* 1. synchronized: This comes from Java hence is only usable with Kotlin Android/JVM based targets.

@synchronized

<some method>

makes the access to method synchronized, i.e., locks and unlock the access to the method.

or

synchronized(<caller instance>)

{

…

}

This is called a synchronized statement, it takes any object as its lock. Usually we pass ‘this’,

* 1. Lock: We can create manual locks and unlock them from anywhere, locks prevent any thread from accessing the statements within when locked. These are like mutexes.

val someLock= ReentrantLock(); //a type of lock

someLock.lock()

…

someLock.unlock();

the unlock can be performed from anywhere in the code so the locked code need not have unlock at its end..

* 1. Semaphore: These are required when signalling is required along with synchronosity.

val somePhore= Semaphore(1) //binary semaphore

somePhore.acquire();

…

somePhore.release();

Kind of like lock but we can define more parameters, the 1 passed to Semaphore ctor stands allows only 1 caller to access the code, if it were more it would allow more. It also accepts a fairness arg, if set to fair the thread that waited the longest gets the access first, otherwise any thread can get it.

* 1. CyclicBarrier: These are used to barr execution of threads until a set number of threads are ready.

For example:

var cb= CyclicBarrier(3){

println(“yo”);

}

..some async code starter

{

cb.await()

{

println(“ready”)

}

}

Stops execution at cb.await until it is called 3 times.

Then it prints,

yo

ready

ready

ready

cb.reset() will reset the count back to 0, and the awaiting threads will get a BrokenBarrierException.

* 1. CountDownLatch: Just like CB but its value decrements and it has to be done manually.

var cdl= CountDownLatch(3)

{

println(“yo”)

}

…

{

cdl.countDown();

cdl.await();

}

countdown can be called anywhere.

* 1. ConcurrentCollections: Apart from ones introduced in Kt, java’s collections can be used too.

Like Collections.synchronizedList(List<T>), similarly named for Maps, Sets etc.

CopyOnWriteArrayList: Creates a list but the list is copied to a local variable before it is mutated then the changes are put at once on the actual list.

HashTable: Individual elements are thread safe, i.e., put/get is safe. The rest of the map isn’t locked so it isn’t safe though.

ConcurrentHashMap: Same as above but locks entire map.

1. @Suppress: Suppresses linter warnings. Like @Suppress("UNUSED\_VARIABLE")
2. BoM: Bill Of Materials, some libraries provide a BoM along with individual component libraries, this is used to automatically provide compatible library versions for the individual components. We use ‘implementation platform’ to use BoMs.

For example:

implementation platform('com.google.firebase:firebase-bom:29.1.0')

implementation 'com.google.firebase:firebase-analytics-ktx'

…

We don’t need to specify versions for other components of the library if we use BoM, as it deduces the most compatible versions (with both, the android project and its sibling components) automatically. If we wish to use our own defined version for a component then we only need to specify the version for the component and it overrides the BoM’s provided version.

1. Annotation: Create annotations with annotation class,

**@Target**(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,

AnnotationTarget.TYPE\_PARAMETER, AnnotationTarget.VALUE\_PARAMETER,

AnnotationTarget.EXPRESSION)

**@Retention**(AnnotationRetention.SOURCE)

**@MustBeDocumented**

**@Repeatable**

**annotation** **class** Fancy

and to use it

**@Fancy** **class** Foo {

**@Fancy** **fun** baz(**@Fancy** foo: Int): Int {

**return** (**@Fancy** 1)

}

}

@Target tells which types can be annotated.

@Retention has 3 values, SOURCE means annotation can be inspected by other annotation classes, binary means annotation is stored in binary and is visible to compiler and tools as well, runtime means it can be inspected at runtime along with other 2 sources as well.

@MustBeDocumented specifies annotation is part of a public API and must be visible in generated API documentation.

@Repeatable allows the annotation to be applied multiple times to same element.

For ex: @Inject

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

* 1. They can have params,

annotation class PP(val why: String)

@PP(“yo”) class wow{…}

param type can be anything, even other classes (abc::class or by using KClass, annotation class WOW(val abc: KClass<\*>) and call with @WOW(String::class)) and annotations (without @, annotation class DD(val pp: PP)) but not Nullable types.

and we can access value from its object with .value

* 1. We can call annotation classes normally too,

fun main(){

PP(“yo”)

}

* 1. They can be put on lambdas too, then they are applied to the invoke() method of the lambda.

val pvc= @PP { Thread.sleep(10)}

* 1. use-site Target:

class Example(@field:PP val foo, // annotate Java field

@get:PP val bar, // annotate Java getter

@param:PP val quux) // annotate Java constructor parameter

or for an entire file,

@file: PP(“yoo”)

Same property can get multiple use-site target annotations,

class mmm(@set:[WOW PP]val abc: ABC){…}

These are the use-site targets,

file

property (annotations with this target are not visible to Java)

field

get (property getter)

set (property setter)

receiver (receiver parameter of an extension function or property)

param (constructor parameter)

setparam (property setter parameter)

delegate (the field storing the delegate instance for a delegated property)

1. Calling Java from Kotlin: We can call Java methods from Kotlin.
   1. Backtick: If a Java method is named same as Kotlin keyword, we can call the Java method in Kotlin by escaping the keyword with ` (backtick).

For example:

Foo.`is`(bar)

in Kotlin code will call the is method of Java. But if we had used Foo.is(bar) then the is method of Kotlin would have been called.

The backtick escaped variables can be used as-is too, so

val `yoo`=22 is a valid variable name.