**Android:**

The primary programming language for Android app development is Kotlin, although Java is still widely used. Kotlin is now the recommended language for Android development, with Google encouraging developers to leverage its modern features and benefits.

Kotlin is a modern, statically typed language that is increasingly favoured for Android development due to its conciseness, expressiveness, and ability to reduce common code errors. It also integrates well with existing Java code.

While Kotlin is the preferred language, Java remains a valid and popular choice for Android development. Many existing Android projects are written in Java, and it still provides a strong foundation for learning app development.

* Feature vs Smart Device:

Any device that is designed to do minimal operations is called as feature device

e.g.-> remote, mobile phones, washing machine etc.

Any device that has got its own Operation system is called as smart device

e.g.-> smartphones, smart TV, smartwatch, smart Refrigerators

* History:

Android is an operation system that is designed on top of Linux OS with some changes

It is designed for mobile devices

It is free and open-source OS

It is developed by Andy Rubin in 2003

It was acquired by google in 2005

In 2007, google announced the development of Android OS

In 2008, HTC launched its first Android device

* Versions of Android:

1.0 -> Android 1.0(September 2008)

1.1 -> Android 1.1 (Feburary 2009)

1.5 -> Android Cupcake (April 2009)

1.6 -> Android Donut (September 2009)

2.0 -> Android Eclair - 2.0 (October 2009) - 2.0.1 (December 2009) - 2.1 (January 2010)

2.2 & 2.2.3 -> Android Froyo (May 2010)

2.3 -> Android GingerBread - 2.3 - 2.3.2 (December 2010) - 2.3.3 - 2.3.7 (Feburary 2011)

3.0 -> Android Honeycomb - 3.0 (Feburary 2011) - 3.1 (May 2011) - 3.2-3.2.6 (July 2011)

4.0 -> Android Ice Cream Sandwich - 4.0 - 4.0.2 (October 2011) - 4.0.3 - 4.0.4 (December 2011)

4.1 -> Android Jelly Bean - 4.1 – 4.1.2 (July 2012) - 4.2 – 4.2.2 (November 2012) - 4.3 – 4.3.1 (July 2013)

4.2 -> Android KitKat - 4.4 – 4.4.4 (October 2013) - 4.4W – 4.4W.2 (June 2014)

5.0 -> Android Lollipop - 5.0 – 5.0.2 (November 2014) - 5.1 – 5.1.1 (March 2015)

6.0 & 6.0.1 -> Android Marshmallow (September 2015)

7.0 -> Android Nougat - 7.0 (August 2016) - 7.1 - 7.1.2 (October 2016)

8.0 -> Android Oreo - 8.0 (August 2017) - 8.1 (December 2017)

9.0 -> Android Pie (August 2018)

10 -> Android 10/ Android Q / Quince Tart (Sepetember 2019)

11 -> Android 11 / Red Velvet Cake (Sepettember 2020)

12 -> Android 12 - 12 (October 2021) / Snow Cone - 12.1 (March 2022) / Snow Cone V2

13 -> Android 13 / Tiramisu (August 2022)

Android Architecture: Android is an open source, Linux-based software stack created for a wide array of devices and form factors.



* Applications:
* It is the top layer of android architecture.
* The pre-installed applications like home, contacts, camera, gallery etc and third-party applications downloaded from the play store like chat applications, games etc. will be installed on this layer.
* Application Framework:
* Application Framework provides several important classes which are used to create an Android application.
* It includes different types of services like activity manager, notification manager, view system, package manager etc. which are helpful for the development of our application
* Android Runtime:
* It is JVM for android, faster than desktop jvm.
* It uses 30% native instructions and 70%-byte codes
* It is possible because of double compilation that android uses.
* The byte code is given to dex compiler which compiles and produces, 70%-byte code and 30% native code in. dex(dalvik executables) file.
* For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART). - ART is written to run multiple virtual machines on low-memory devices by executing Dalvik Executable format (DEX) files, a bytecode format designed specifically for Android that's optimized for a minimal memory footprint.
* Some of the major features of ART include the following:
* Ahead-of-time (AOT) and just-in-time (JIT) compilation
* Optimized garbage collection (GC)
* On Android 9 (API level 28) and higher, conversion of an app package's DEX files to more compact machine code.
* Prior to Android version 5.0 (API level 21), Dalvik was the Android runtime. If your app runs well on ART, then it can work on Dalvik as well, but the reverse might not be true.
* Android also includes a set of core runtime libraries that provide most of the functionality of the Java programming language, including some Java 8 language features, that the Java API framework uses.
* Platform Libraries:
* The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.
* Android has its own C runtime library.
* The name for the Android’s runtime library is Bionic.
* Hardware abstraction layer (HAL):
* The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework.
* The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or Bluetooth module.
* When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.
* Linux Kernel:
* Linux Kernel is heart of the android architecture.
* For android operating system the Linux kernel is customized.
* Changes added to android from main line kernel is called as Androdism.
* It manages all the available drivers such as display drivers, camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc. which are required during the runtime.
* The Linux Kernel will provide an abstraction layer between the device hardware and the other components of android architecture.
* The features of Linux kernel are:

1. Security: The Linux kernel handles the security between the application and the system.

2. Memory Management: It efficiently handles the memory management thereby providing the freedom to develop our apps.

3. Process Management: It manages the process well, allocates resources to processes whenever

they need them.

4. Network Stack: It effectively handles the network communication.

5. Driver Model: It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

File Structure:

Manifest folder-

* Manifests folder contains AndroidManifest.xml of our android application.
* This file contains information about our application such as the Android version, metadata and other application components.
* It acts as an intermediator between android OS and our application.

java Folder-

* The Java folder contains all the java source code (.java) files that we create during the app development, including other Test files.

res (Resource Folder)-

* The resource folder is the most important folder because it contains all the non-code sources like images layouts, and UI strings for our android application.
* the sub folders inside this res folder are

1. Drawable

It is used to store the images/logos used in our application

1. Layout

It is used to create the UI of our application

1. mipmap

It is used to have icon for our application

1. values

It is used to keep constants that we can use across our application

1. anim

It is used to keep the animation xml files

1. menu

It is used to keep the menu xml files for the menus that we create

Gradle Scripts-

In build. Gradle (Project) there are build scripts. In build. Gradle (Module) plugins and implementations are used to build configurations that can be applied to all our application modules.

Gradle:

It is a build automation tool It is an open-source tool It is used to build the software’s that we design It can build any type of software’s Android studio uses this Gradle for the automated build system for its application that we design.

Fundamental Components of Android:

The fundamental components are also called as Pillars of Android

There are 4 fundamental components for Android application:

Activity:

An activity is the entry point for interacting with the user.

It represents a single screen with a user interface.

Services:

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons.

It is a component that runs in the background to perform long-running operations or to perform work for remote processes.

A service does not provide a user interface.

Content Provider:

A content provider manages a shared set of app data that you can store in the file system, in a SQLite database or on any other persistent storage location that your app can access.

Through the content provider, other apps can query or modify the data, if the content provider permits it.

Broadcast Receiver:

A broadcast receiver is a component that lets the system deliver events to the app outside of a regular user flow so the app can respond to system-wide broadcast announcements.

Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running.

Activity:

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View). While activities are often presented to the user as full-screen windows,

they can also be used in other ways: as floating windows (via a theme with R.attr.windowIsFloating set), multi-window mode or embedded into other windows. There are two methods almost all subclasses of Activity will implement:

onCreate(Bundle) is where you initialize your activity. Most importantly, here you will usually call setContentView(int) with a layout resource defining your UI, and using findViewById(int) to retrieve the widgets in that UI that you need to interact with programmatically.

onPause() is where you deal with the user pausing active interaction with the activity. Any changes made by the user should at this point be committed (usually to the ContentProvider holding the data). In this state the activity is still visible on screen.

To be of use with Context.startActivity(), all activity classes must have a corresponding <activity> declaration in their package's AndroidManifest.xml.

Activity Lifecycle:

Activities in the system are managed as [activity stacks](https://developer.android.com/guide/components/activities/tasks-and-back-stack). When a new activity is started, it is usually placed on the top of the current stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits. There can be one or multiple activity stacks visible on screen.

An activity has essentially four states:

* If an activity is in the foreground of the screen (at the highest position of the topmost stack), it is active or running. This is usually the activity that the user is currently interacting with.
* If an activity has lost focus but is still presented to the user, it is visible. It is possible if a new non-full-sized or transparent activity has focus on top of your activity, another activity has higher position in multi-window mode, or the activity itself is not focusable in current windowing mode. Such activity is completely alive (it maintains all state and member information and remains attached to the window manager).
* If an activity is completely obscured by another activity, it is stopped or hidden. It still retains all state and member information; however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
* The system can drop the activity from memory by either asking it to finish, or simply killing its process, making it destroyed. When it is displayed again to the user, it must be completely restarted and restored to its previous state.



There are three key loops you may be interested in monitoring within your activity:

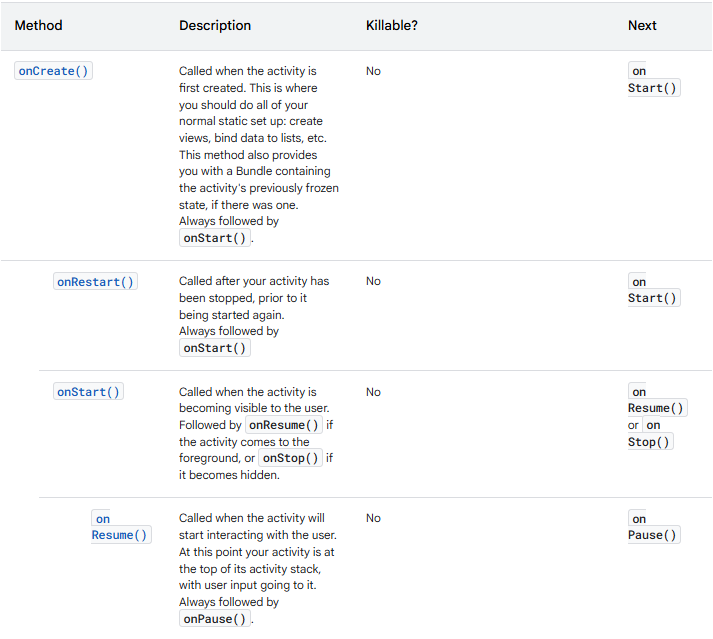
The **entire lifetime** of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread in onDestroy().

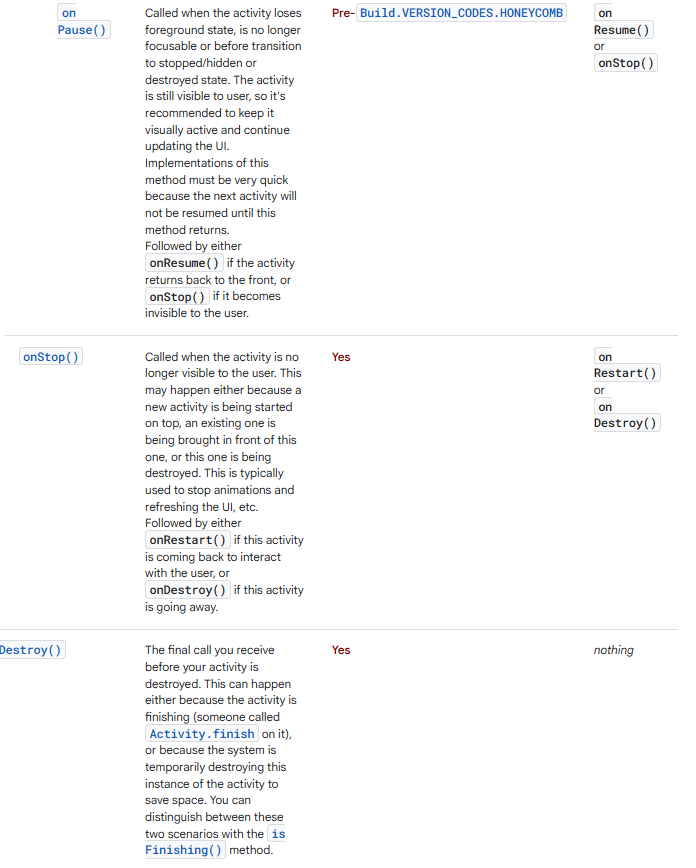
The **visible lifetime** of an activity happens between a call to onStart() until a corresponding call to onStop(). During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user. For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop() when the user no longer sees what you are displaying. The onStart() and onStop() methods can be called multiple times, as the activity becomes visible and hidden to the user.

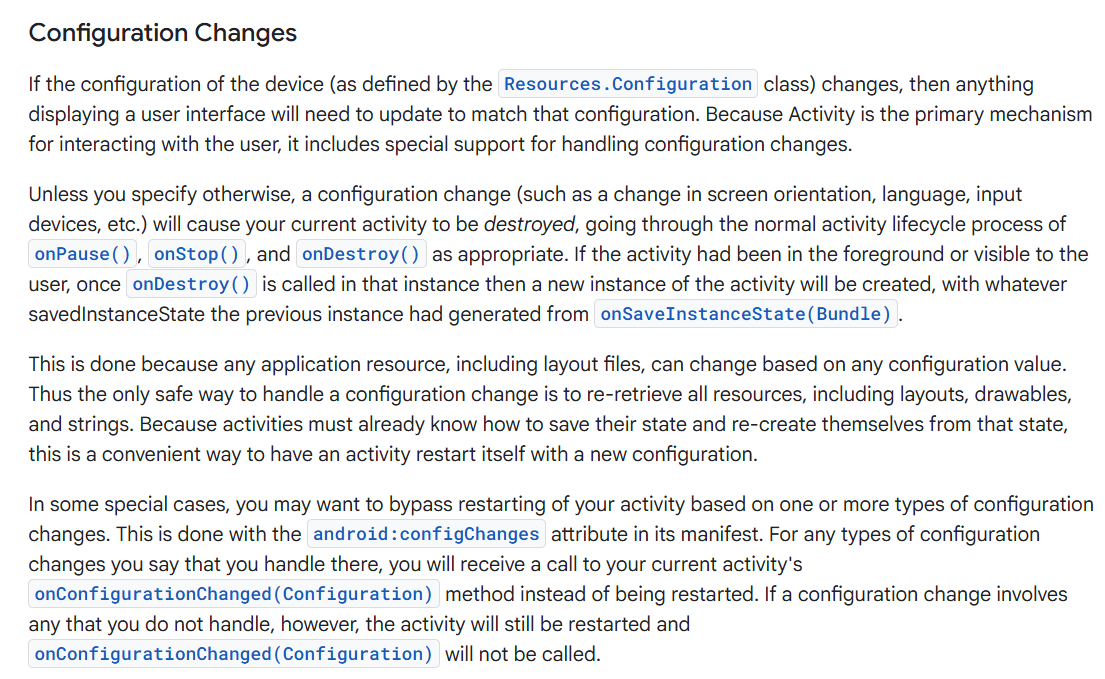
The **foreground lifetime** of an activity happens between a call to onResume() until a corresponding call to onPause(). During this time the activity is visible, active and interacting with the user. An activity can frequently go between the resumed and paused states -- for example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

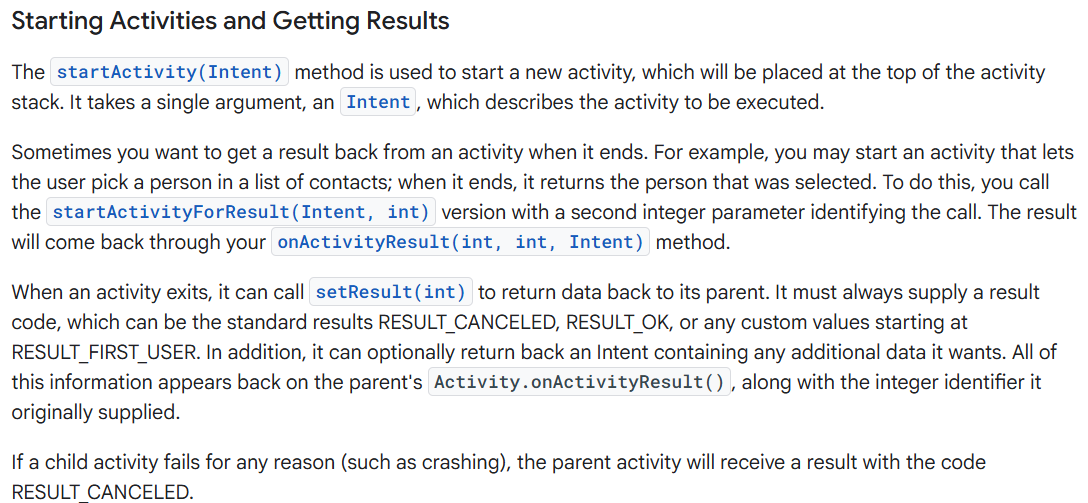
The entire lifecycle of an activity is defined by the following Activity methods. All of these are hooks that you can override to do appropriate work when the activity changes state. All activities will implement onCreate(Bundle) to do their initial setup; many will also implement onPause() to commit changes to data and prepare to pause interacting with the user, and onStop() to handle no longer being visible on screen. You should always call up to your superclass when implementing these methods.











public class MyActivity extends Activity {

...

static final int PICK\_CONTACT\_REQUEST = 0;

public boolean onKeyDown(int keyCode, KeyEvent event) {

if (keyCode == KeyEvent.KEYCODE\_DPAD\_CENTER) {

// When the user center presses, let them pick a contact.

startActivityForResult(

new Intent(Intent.ACTION\_PICK,

new Uri("content://contacts")),

PICK\_CONTACT\_REQUEST);

return true;

}

return false;

}

protected void onActivityResult(int requestCode, int resultCode,

Intent data) {

if (requestCode == PICK\_CONTACT\_REQUEST) {

if (resultCode == RESULT\_OK) {

// A contact was picked. Here we will just display it

// to the user.

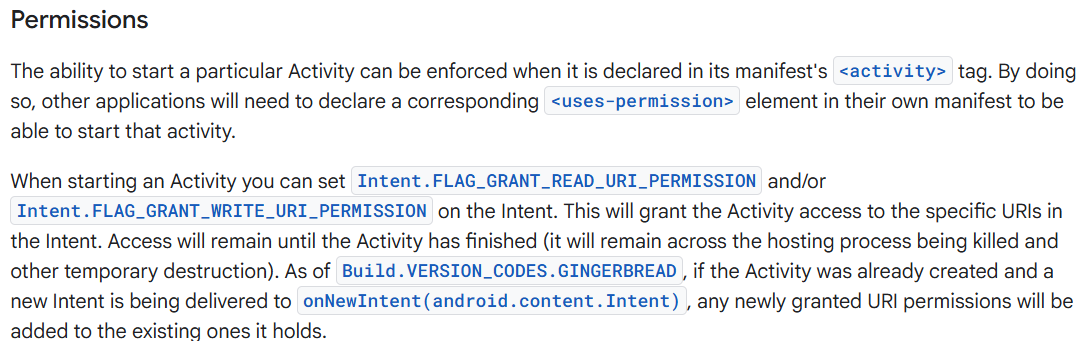
startActivity(new Intent(Intent.ACTION\_VIEW, data));

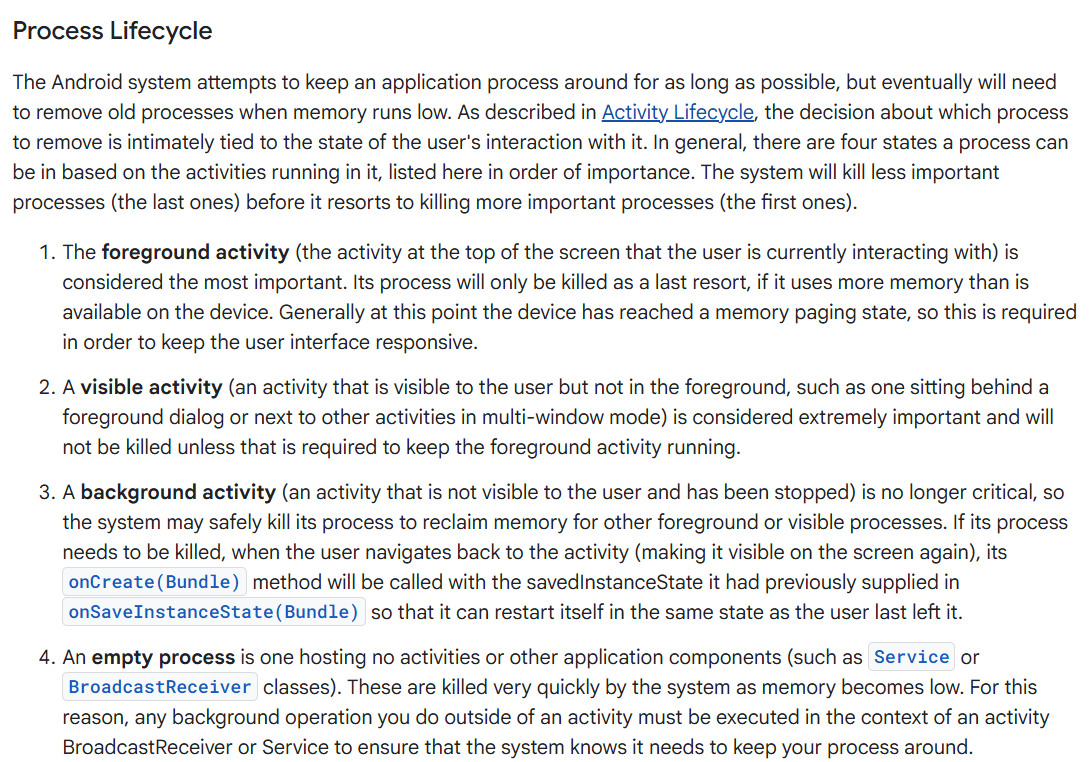
}

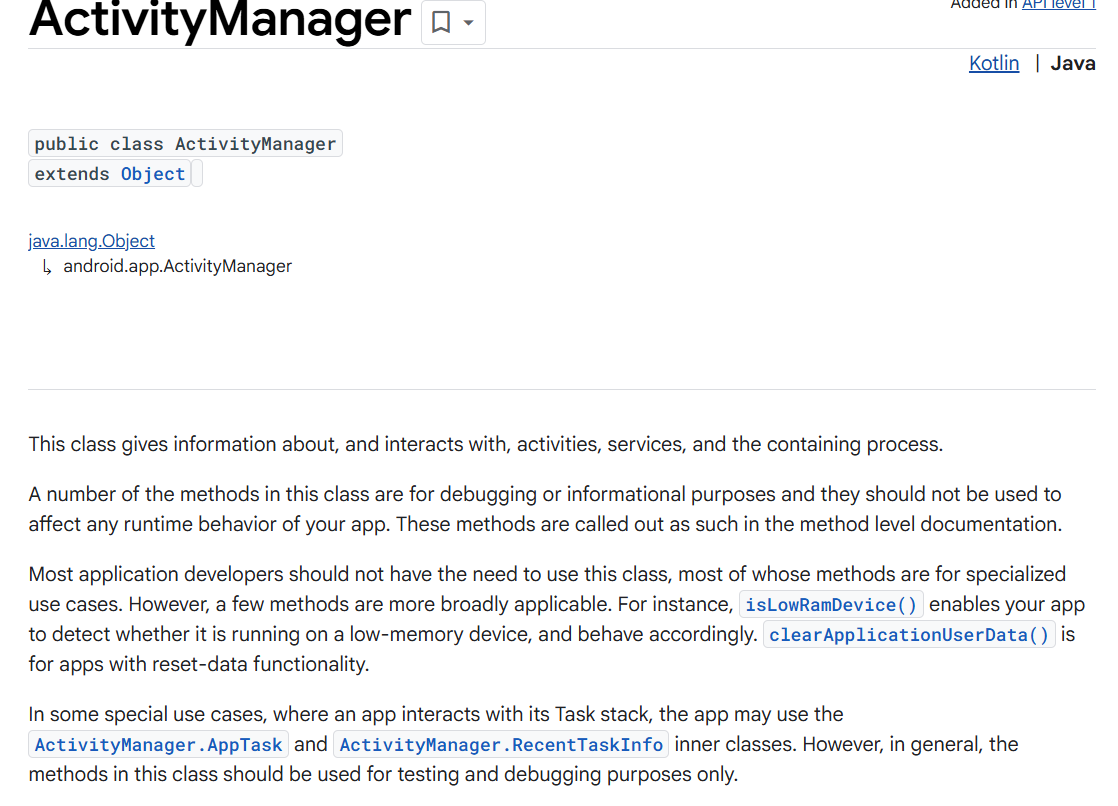
}

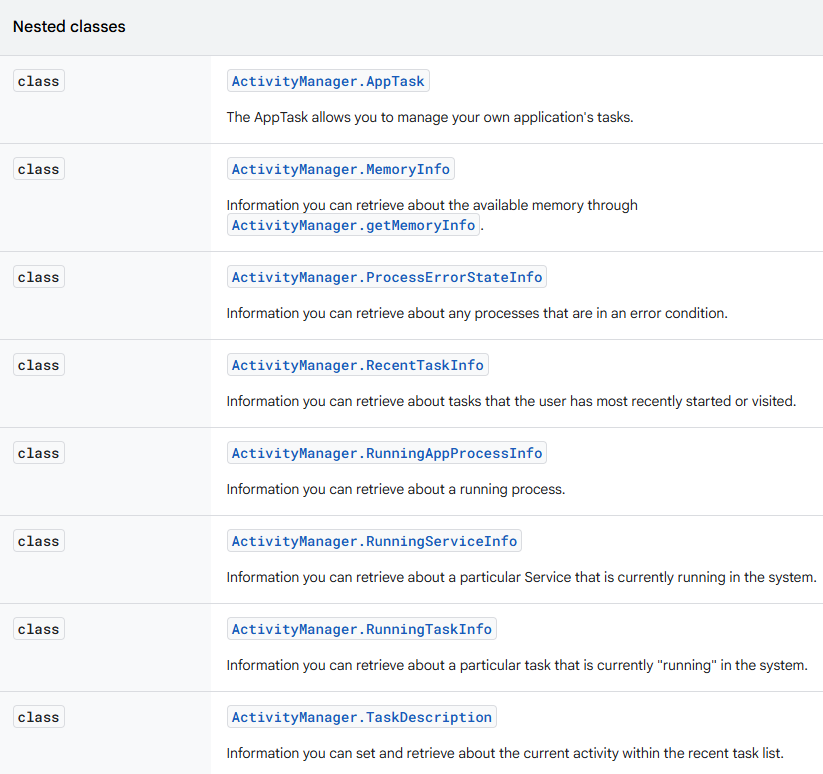
}

}









**Comparison to Java﻿:**

**Kotlin fixes a series of issues that Java suffers from:**

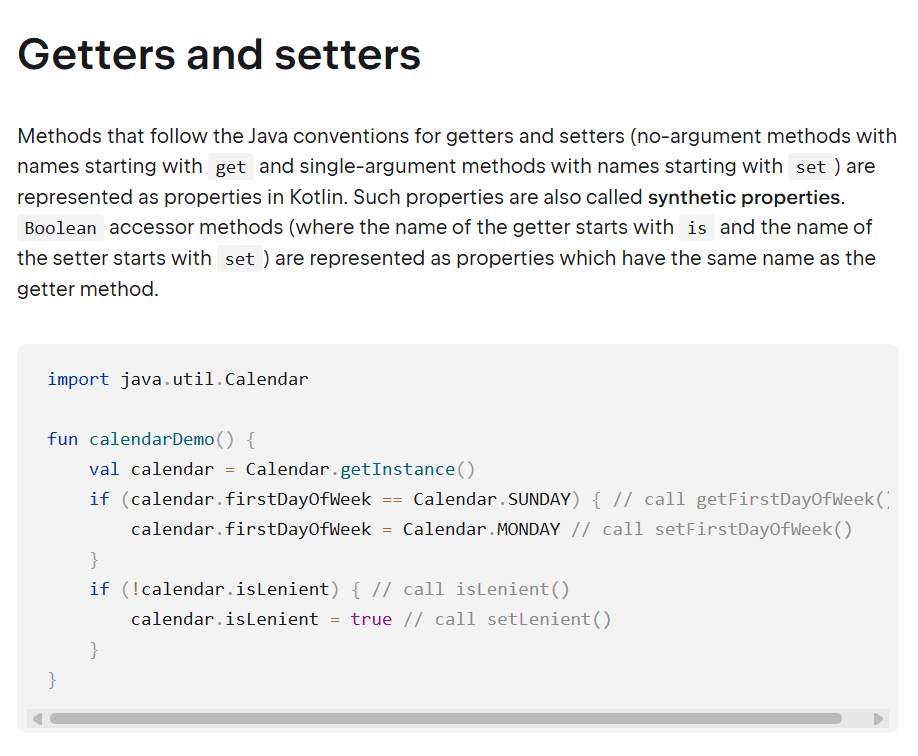
* Null references are [controlled by the type system](https://kotlinlang.org/docs/null-safety.html).
* [No raw types](https://kotlinlang.org/docs/java-interop.html#java-generics-in-kotlin)
* Arrays in Kotlin are [invariant](https://kotlinlang.org/docs/arrays.html)
* Kotlin has proper [function types](https://kotlinlang.org/docs/lambdas.html#function-types), as opposed to Java's SAM-conversions
* [Use-site variance](https://kotlinlang.org/docs/generics.html#use-site-variance-type-projections) without wildcards
* Kotlin does not have checked [exceptions](https://kotlinlang.org/docs/exceptions.html)
* [Separate interfaces for read-only and mutable collections](https://kotlinlang.org/docs/collections-overview.html).

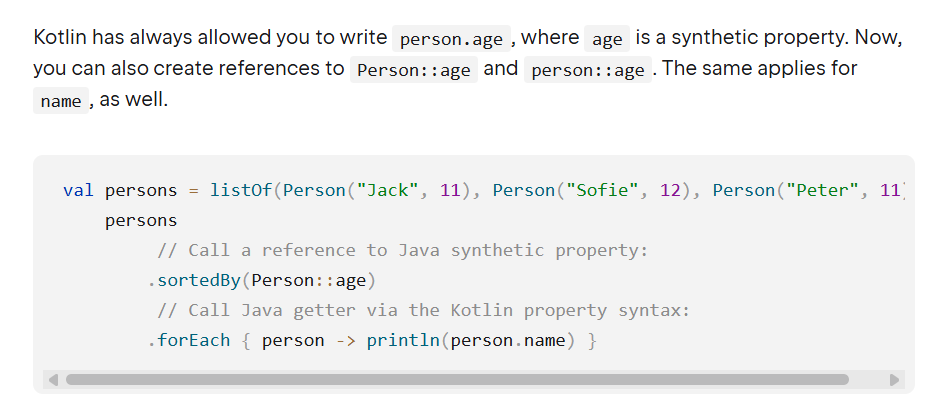
**What Java has that Kotlin does not﻿?**

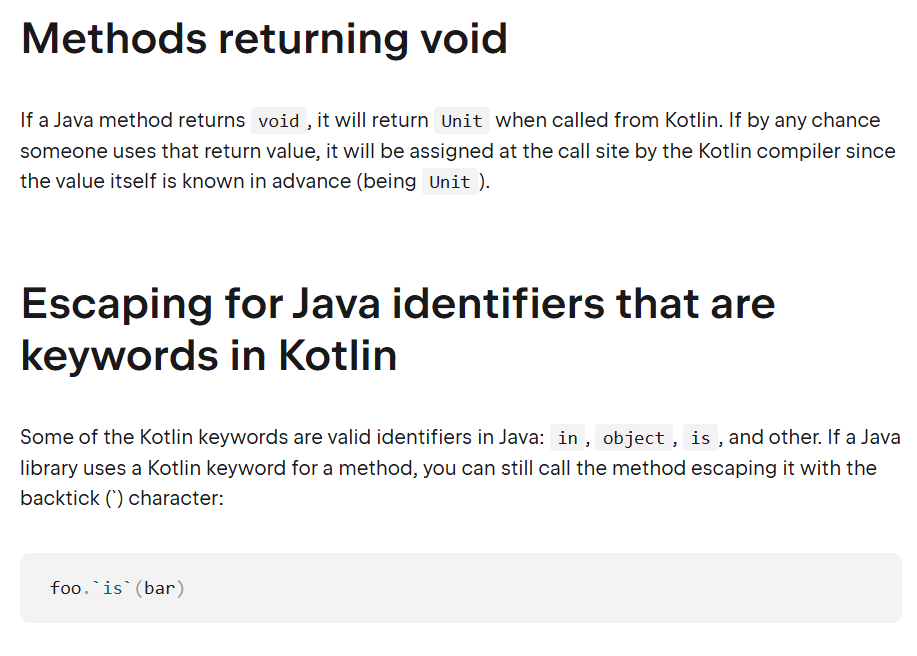
* [Checked exceptions](https://kotlinlang.org/docs/exceptions.html)
* [Primitive types](https://kotlinlang.org/docs/basic-types.html) that are not classes. The byte-code uses primitives where possible, but they are not explicitly available.
* [Static members](https://kotlinlang.org/docs/classes.html) are replaced with [companion objects](https://kotlinlang.org/docs/object-declarations.html#companion-objects), [top-level functions](https://kotlinlang.org/docs/functions.html), [extension functions](https://kotlinlang.org/docs/extensions.html#extension-functions), or [@JvmStatic](https://kotlinlang.org/docs/java-to-kotlin-interop.html#static-methods).
* [Wildcard-types](https://kotlinlang.org/docs/generics.html) are replaced with [declaration-site variance](https://kotlinlang.org/docs/generics.html#declaration-site-variance) and [type projections](https://kotlinlang.org/docs/generics.html#type-projections).
* [Ternary-operator a ? b : c](https://kotlinlang.org/docs/control-flow.html#if-expression) is replaced with [if expression](https://kotlinlang.org/docs/control-flow.html#if-expression).
* [Records](https://openjdk.org/jeps/395)
* [Pattern Matching](https://openjdk.org/projects/amber/design-notes/patterns/pattern-matching-for-java)
* package-private [visibility modifier](https://kotlinlang.org/docs/visibility-modifiers.html)

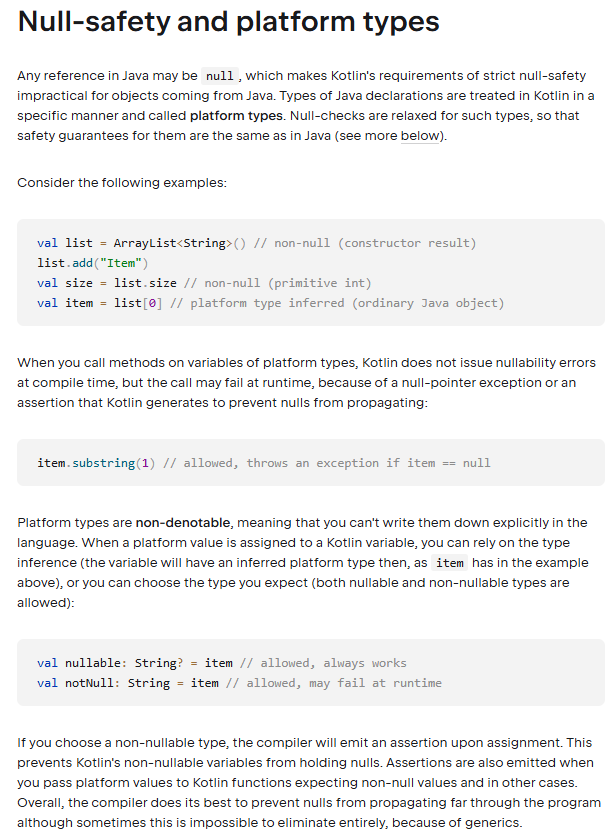
**What Kotlin has that Java does not﻿?**

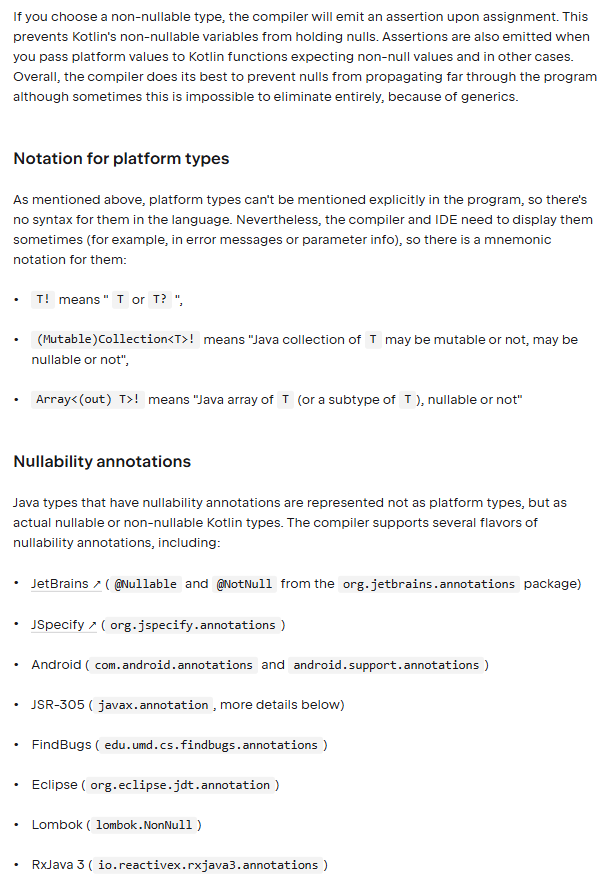
* [Lambda expressions](https://kotlinlang.org/docs/lambdas.html) + [Inline functions](https://kotlinlang.org/docs/inline-functions.html) = performant custom control structures
* [Extension functions](https://kotlinlang.org/docs/extensions.html)
* [Null-safety](https://kotlinlang.org/docs/null-safety.html)
* [Smart casts](https://kotlinlang.org/docs/typecasts.html) (Java 16: [Pattern Matching for instanceof](https://openjdk.org/jeps/394))
* [String templates](https://kotlinlang.org/docs/strings.html) (Java 21: [String Templates (Preview)](https://openjdk.org/jeps/430))
* [Properties](https://kotlinlang.org/docs/properties.html)
* [Primary constructors](https://kotlinlang.org/docs/classes.html)
* [First-class delegation](https://kotlinlang.org/docs/delegation.html)
* [Type inference for variable and property types](https://kotlinlang.org/docs/basic-types.html) (Java 10: [Local-Variable Type Inference](https://openjdk.org/jeps/286))
* [Singletons](https://kotlinlang.org/docs/object-declarations.html)
* [Declaration-site variance & Type projections](https://kotlinlang.org/docs/generics.html)
* [Range expressions](https://kotlinlang.org/docs/ranges.html)
* [Operator overloading](https://kotlinlang.org/docs/operator-overloading.html)
* [Companion objects](https://kotlinlang.org/docs/classes.html#companion-objects)
* [Data classes](https://kotlinlang.org/docs/data-classes.html)
* [Coroutines](https://kotlinlang.org/docs/coroutines-overview.html)
* [Top-level functions](https://kotlinlang.org/docs/functions.html)
* [Default arguments](https://kotlinlang.org/docs/functions.html#default-arguments)
* [Named parameters](https://kotlinlang.org/docs/functions.html#named-arguments)
* [Infix functions](https://kotlinlang.org/docs/functions.html#infix-notation)
* [Expect and actual declarations](https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-expect-actual.html)
* [Explicit API mode](https://kotlinlang.org/docs/whatsnew14.html#explicit-api-mode-for-library-authors) and [better control of API surface](https://kotlinlang.org/docs/opt-in-requirements.html)

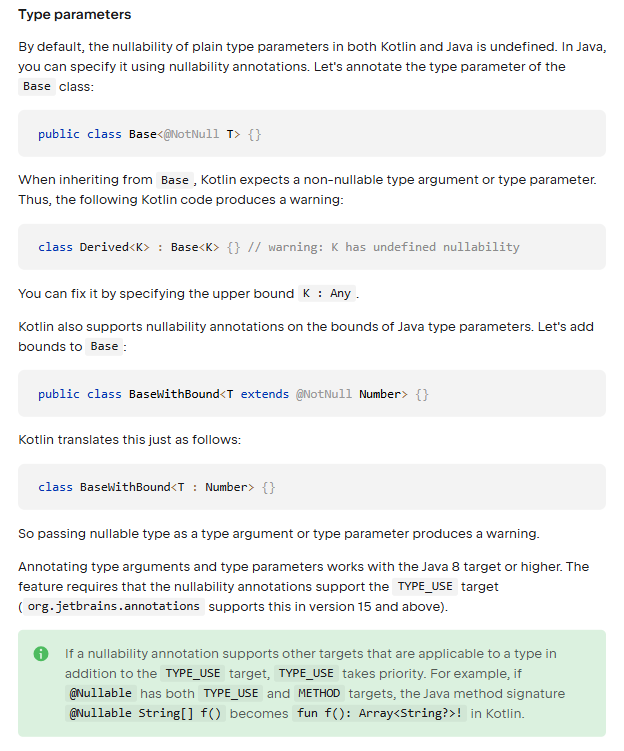






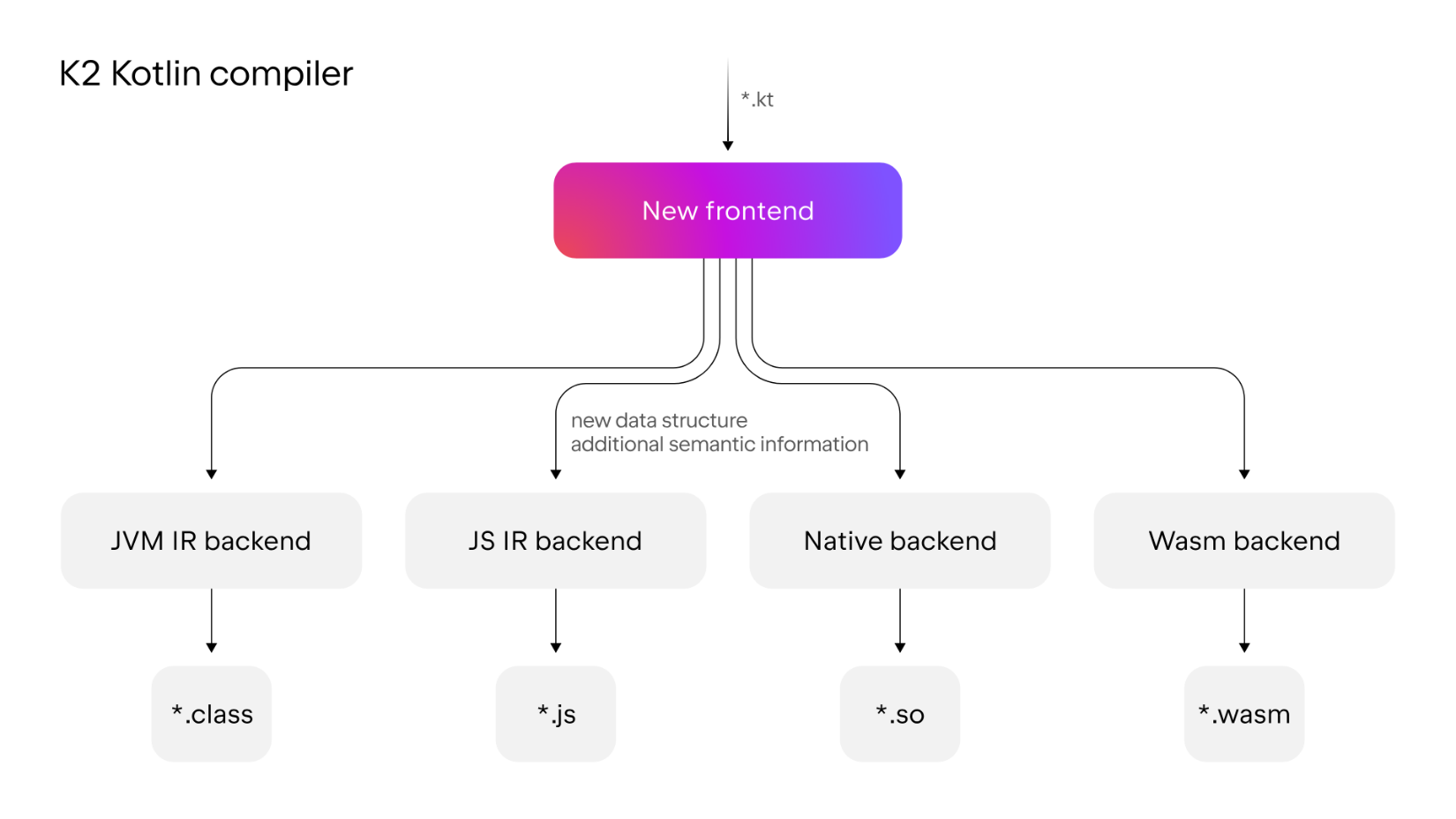






**K2 compiler:**

As the Kotlin language and ecosystem have continued to evolve, so has the Kotlin compiler. The first step was the introduction of the new JVM and JS IR (Intermediate Representation) backends that share logic, simplifying code generation for targets on different platforms. Now, the next stage of its evolution brings a new frontend known as K2.



With the arrival of the K2 compiler, the Kotlin frontend has been completely rewritten and features a new, more efficient architecture. The fundamental change the new compiler brings is the use of one unified data structure that contains more semantic information. This frontend is responsible for performing semantic analysis, call resolution, and type inference.

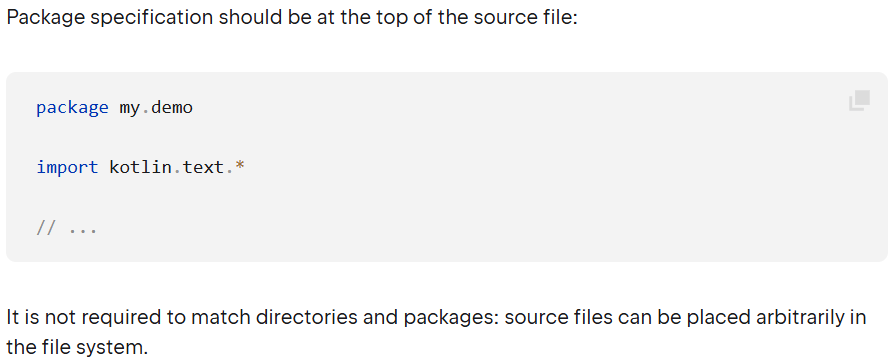
The new architecture and enriched data structure enables the K2 compiler to provide the following benefits:

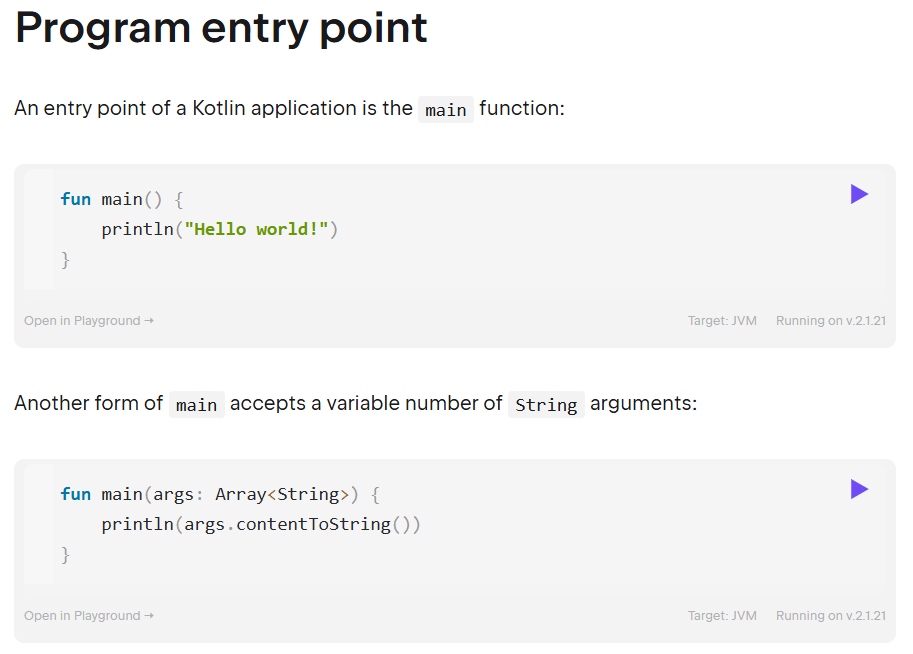
* Improved call resolution and type inference. The compiler behaves more consistently and understands your code better.
* Easier introduction of syntactic sugar for new language features. In the future, you'll be able to use more concise, readable code when new features are introduced.
* Faster compilation times. Compilation times can be [significantly faster](https://kotlinlang.org/docs/k2-compiler-migration-guide.html#performance-improvements).
* Enhanced IDE performance. Starting with 2025.1, IntelliJ IDEA uses K2 mode to analyze your Kotlin code, increasing stability and providing performance improvements. For more information, see [Support in IDEs](https://kotlinlang.org/docs/k2-compiler-migration-guide.html#support-in-ides).

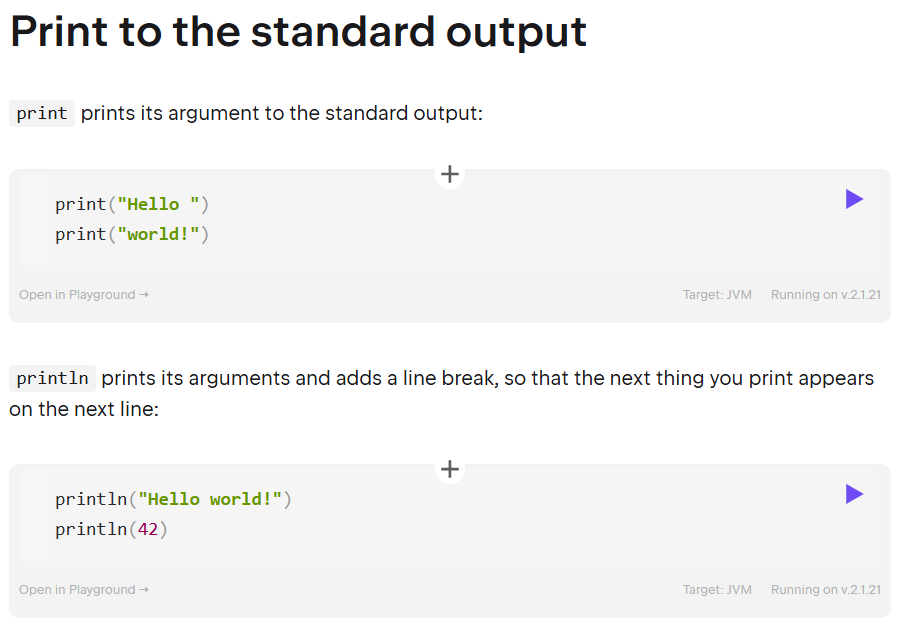
The new K2 compiler is enabled by default starting with 2.0.0. For more information on the new features provided in Kotlin 2.0.0, as well as the new K2 compiler, see [What's new in Kotlin 2.0.0](https://kotlinlang.org/docs/whatsnew20.html).

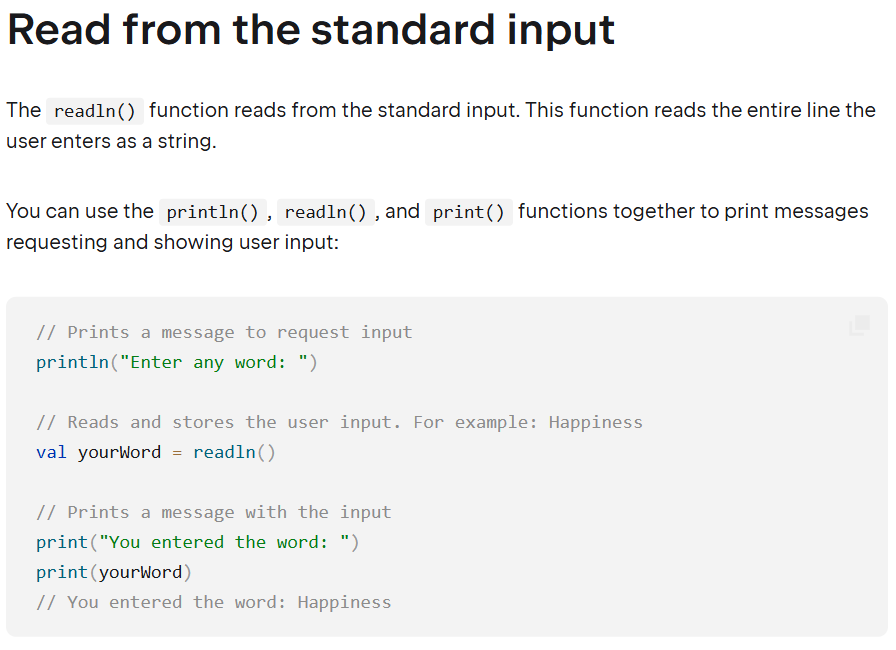
**Basic syntax﻿:**

**Package definition and imports﻿:-**

****

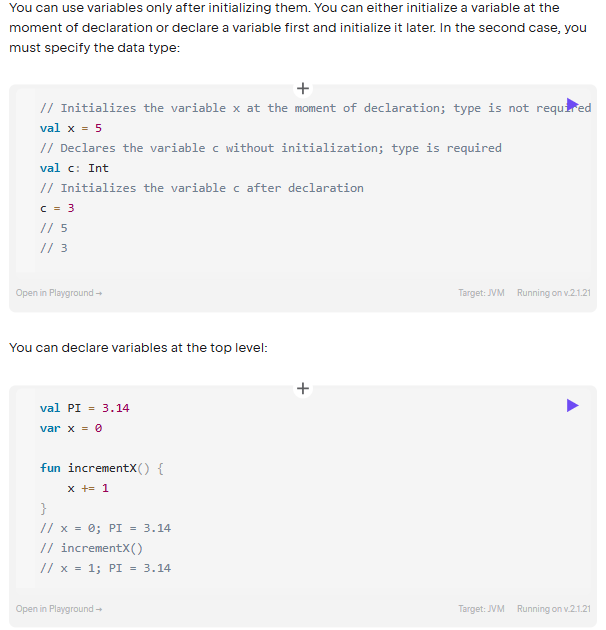




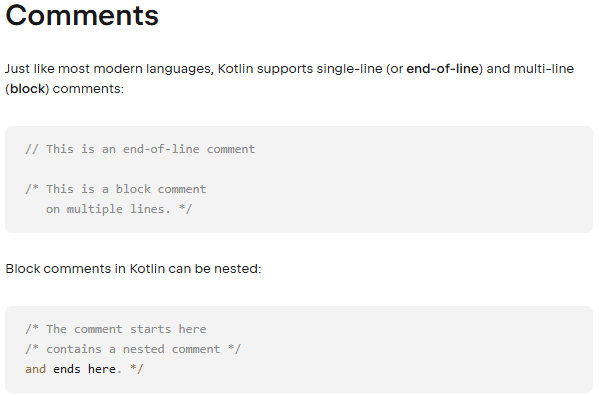




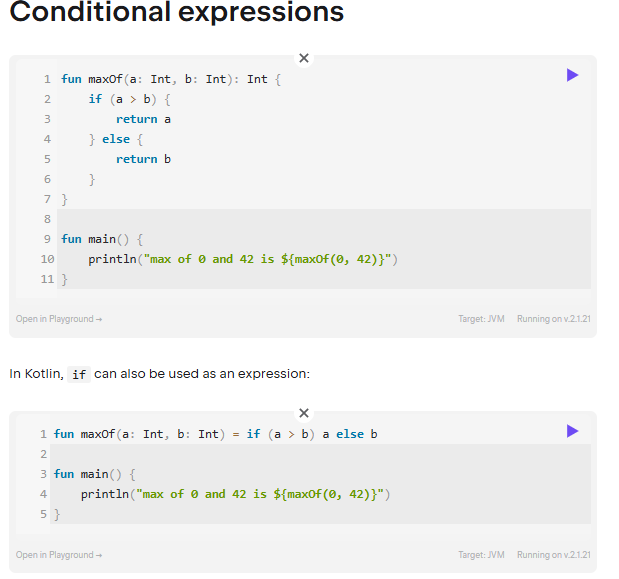




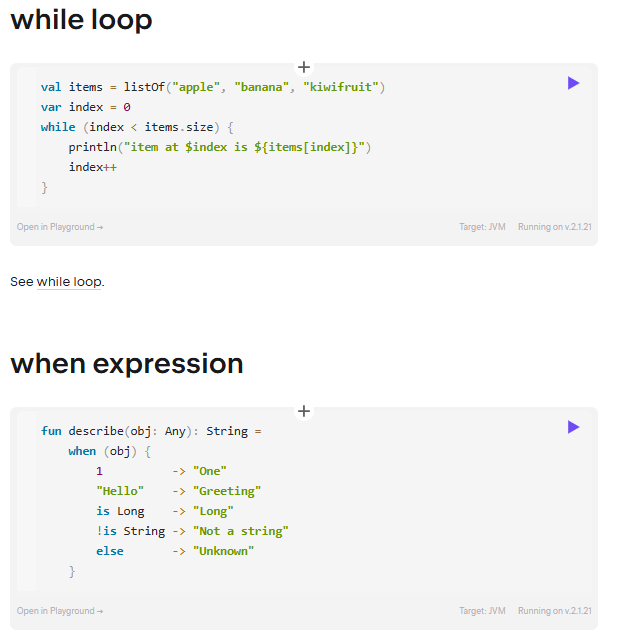




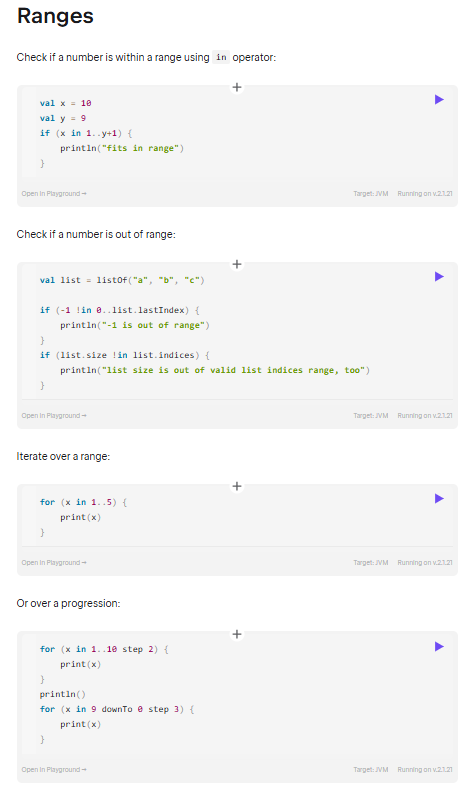


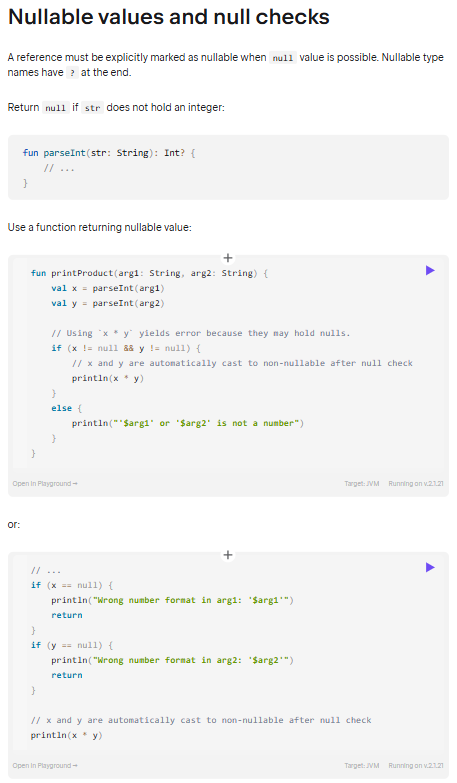


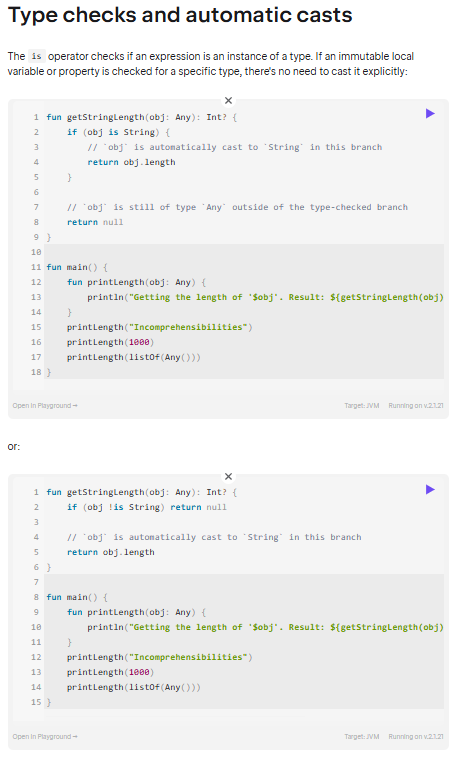




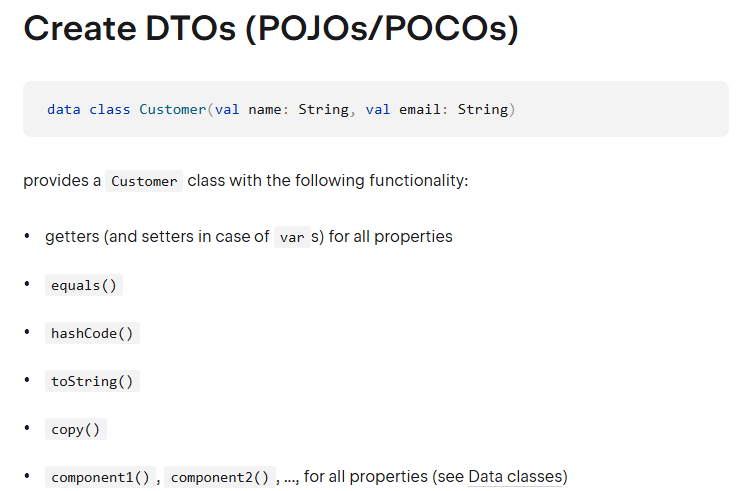




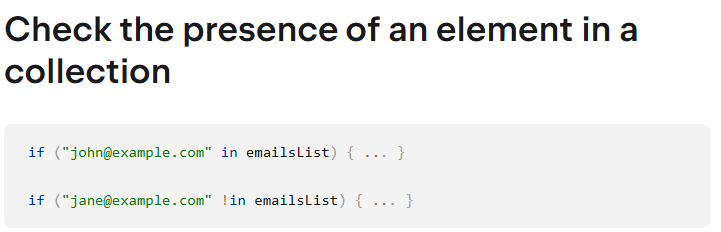


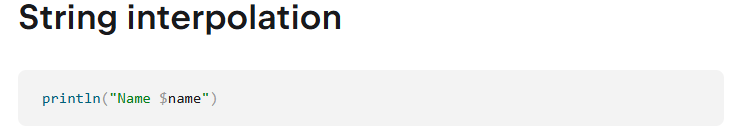


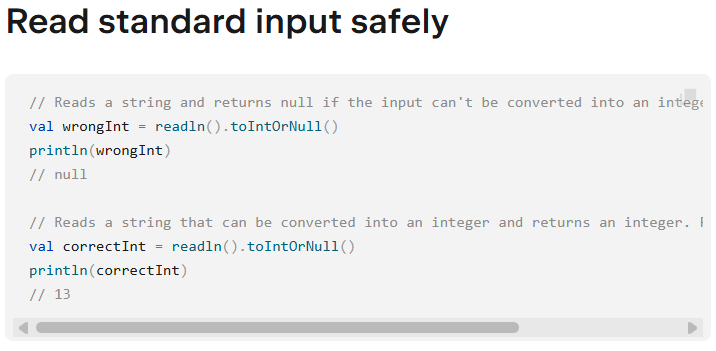
**Idioms﻿:** A collection of random and frequently used idioms in Kotlin. If you have a favourite idiom, contribute it by sending a pull request.

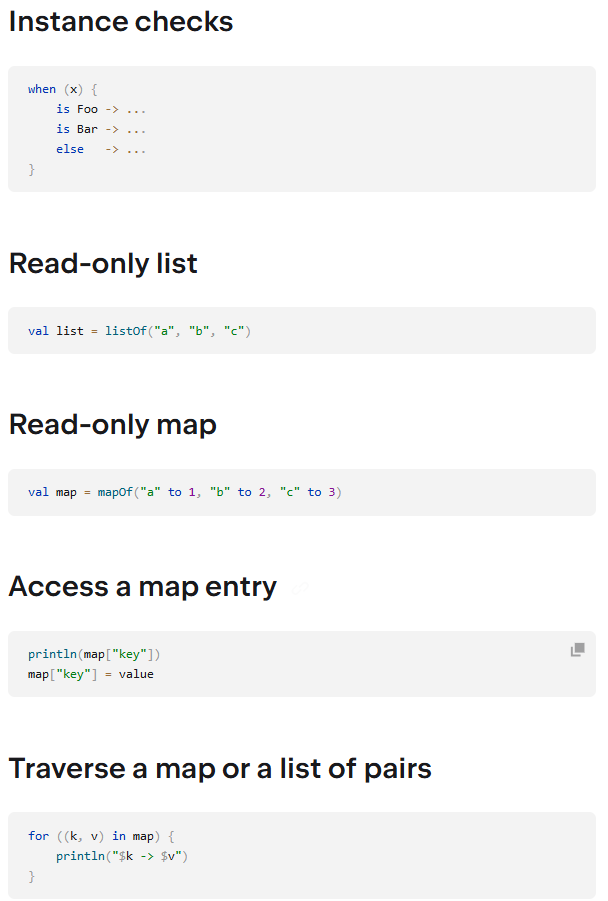


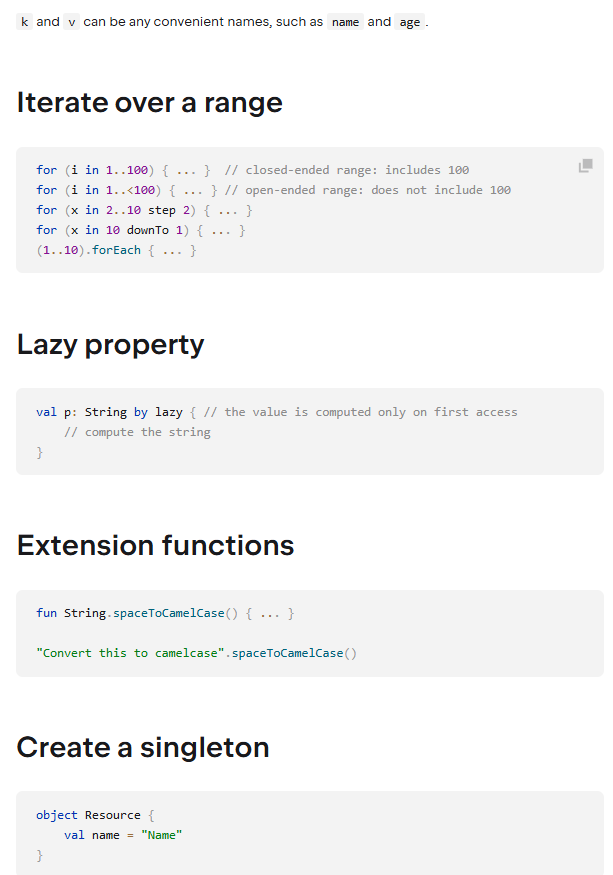








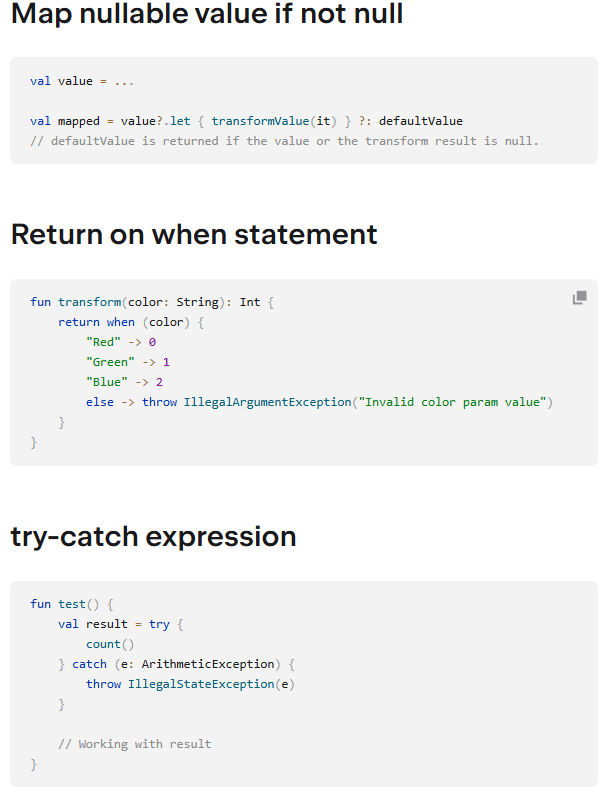
















<https://kotlinlang.org/docs/coding-conventions.html>

<https://developer.android.com/kotlin/add-kotlin>

**Android KTX**Android KTX is a set of Kotlin extensions that are included with Android [Jetpack](https://developer.android.com/jetpack) and other Android libraries. KTX extensions provide concise, idiomatic Kotlin to Jetpack, Android platform, and other APIs. To do so, these extensions leverage several Kotlin language features, including the following:

* Extension functions
* Extension properties
* Lambdas
* Named parameters
* Parameter default values
* Coroutines

As an example, when working with [SharedPreferences](https://developer.android.com/reference/android/content/SharedPreferences), you must [create an editor](https://developer.android.com/reference/android/content/SharedPreferences#edit()) before you can make modifications to the preferences data. You must also apply or commit those changes when you are finished editing, as shown in the following example:

sharedPreferences

.edit() // create an Editor

.putBoolean("key", value)

.apply() // write to disk asynchronously

Kotlin lambdas are a perfect fit for this use case. They allow you to take a more concise approach by passing a block of code to execute after the editor is created, letting the code execute, and then letting the SharedPreferences API apply the changes atomically.

Here's an example of one of the Android KTX Core functions, [SharedPreferences.edit](https://developer.android.com/reference/kotlin/androidx/core/content/package-summary#edit), which adds an edit function to SharedPreferences. This function takes an optional boolean flag as its first argument that indicates whether to commit or apply the changes. It also receives an action to perform on the SharedPreferences editor in the form of a lambda.

// SharedPreferences.edit extension function signature from Android KTX - Core

// inline fun SharedPreferences.edit(

// commit: Boolean = false,

// action: SharedPreferences.Editor.() -> Unit)

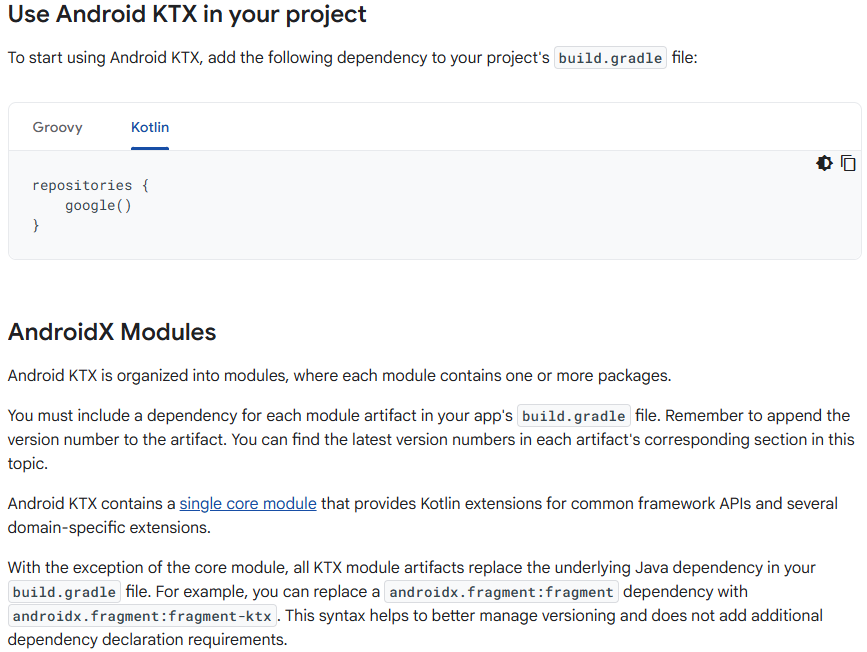
// Commit a new value asynchronously

sharedPreferences.edit { putBoolean("key", value) }

// Commit a new value synchronously

sharedPreferences.edit(commit = true) { putBoolean("key", value) }

The caller can choose whether to commit or apply the changes. The action lambda is itself an anonymous extension function on SharedPreferences.Editor which returns Unit, as indicated by its signature. This is why inside the block, you are able to perform the work directly on the SharedPreferences.Editor.



**Core KTX**

The Core KTX module provides extensions for common libraries that are part of the Android framework. These libraries do not have Java-based dependencies that you need to add to build.gradle.

To include this module, add the following to your app's build.gradle file:

dependencies {

implementation("androidx.core:core-ktx:1.16.0")

}

Here's a list of the packages that are contained in the Core KTX module:

* [androidx.core.animation](https://developer.android.com/reference/kotlin/androidx/core/animation/package-summary)
* [androidx.core.content](https://developer.android.com/reference/kotlin/androidx/core/content/package-summary)
* [androidx.core.content.res](https://developer.android.com/reference/kotlin/androidx/core/content/res/package-summary)
* [androidx.core.database](https://developer.android.com/reference/kotlin/androidx/core/database/package-summary)
* [androidx.core.database.sqlite](https://developer.android.com/reference/kotlin/androidx/core/database/sqlite/package-summary)
* [androidx.core.graphics](https://developer.android.com/reference/kotlin/androidx/core/graphics/package-summary)
* [androidx.core.graphics.drawable](https://developer.android.com/reference/kotlin/androidx/core/graphics/drawable/package-summary)
* [androidx.core.location](https://developer.android.com/reference/kotlin/androidx/core/location/package-summary)
* [androidx.core.net](https://developer.android.com/reference/kotlin/androidx/core/net/package-summary)
* [androidx.core.os](https://developer.android.com/reference/kotlin/androidx/core/os/package-summary)
* [androidx.core.text](https://developer.android.com/reference/kotlin/androidx/core/text/package-summary)
* [androidx.core.transition](https://developer.android.com/reference/kotlin/androidx/core/transition/package-summary)
* [androidx.core.util](https://developer.android.com/reference/kotlin/androidx/core/util/package-summary)
* [androidx.core.view](https://developer.android.com/reference/kotlin/androidx/core/view/package-summary)
* [androidx.core.widget](https://developer.android.com/reference/kotlin/androidx/core/widget/package-summary)

Collection KTX:

The Collection extensions contain utility functions for working with Android's memory-efficient collection libraries, including ArrayMap, LongSparseArray, LruCache, and others.

To use this module, add the following to your app's build.gradle file:

dependencies {

implementation("androidx.collection:collection-ktx:1.5.0")

}

**Fragment KTX:**

The [Fragment KTX module](https://developer.android.com/reference/kotlin/androidx/fragment/app/package-summary#extension-functions-summary) provides a number of extensions to simplify the fragment API.

To include this module, add the following to your app's build.gradle file:

dependencies {

implementation("androidx.fragment:fragment-ktx:1.8.8")

}

With the Fragment KTX module, you can simplify fragment transactions with lambdas, for example:

fragmentManager().commit {

addToBackStack("...")

setCustomAnimations(

R.anim.enter\_anim,

R.anim.exit\_anim)

add(fragment, "...")

}

You can also bind to a ViewModel in one line by using the viewModels and activityViewModels property delegates:

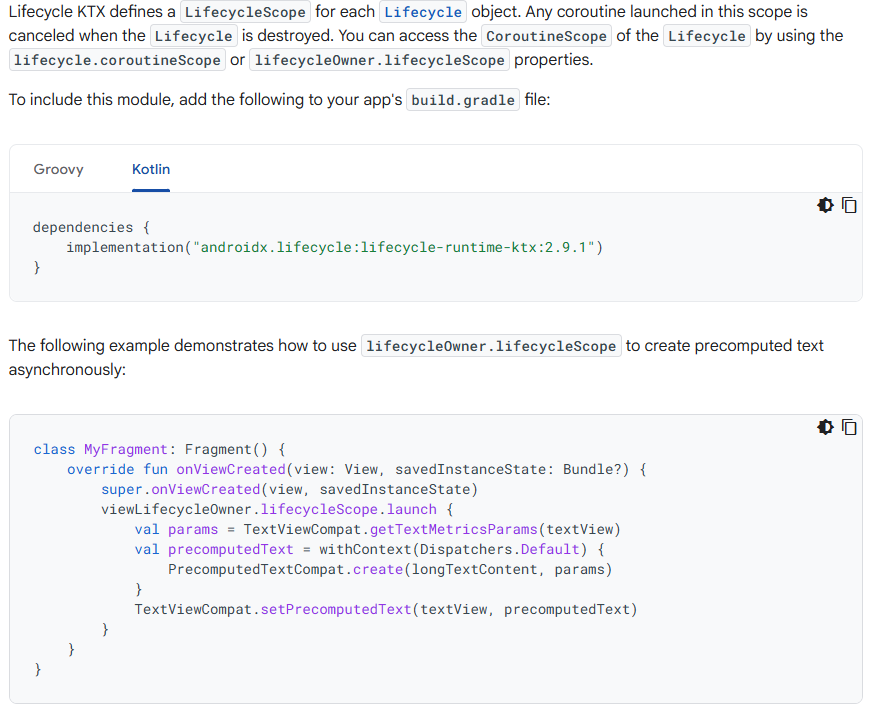
// Get a reference to the ViewModel scoped to this Fragment

val viewModel by viewModels<MyViewModel>()

// Get a reference to the ViewModel scoped to its Activity

val viewModel by activityViewModels<MyViewModel>()

Lifecycle KTX:



<https://developer.android.com/kotlin/ktx/extensions-list>

**Kotlin coroutines on Android:**

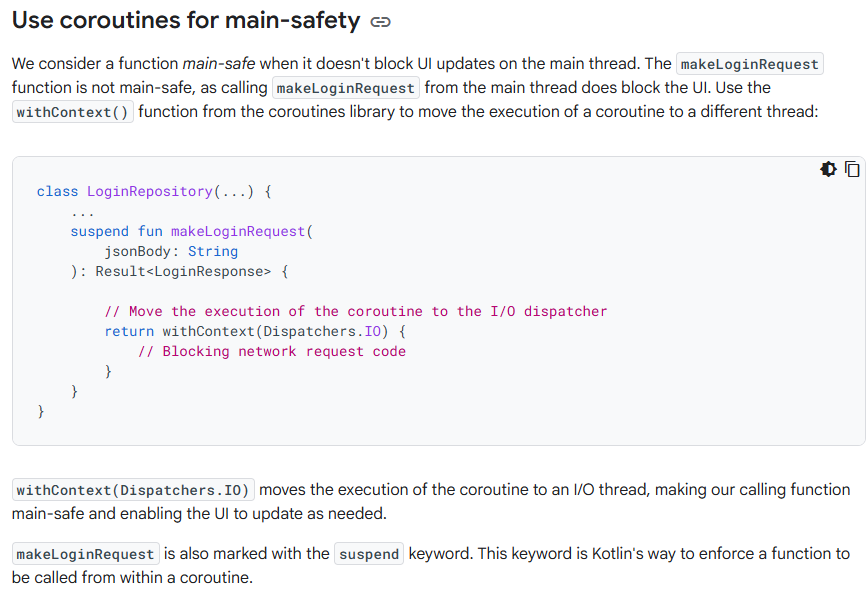
A *coroutine* is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously. [Coroutines](https://kotlinlang.org/docs/coroutines-guide.html) were added to Kotlin in version 1.3 and are based on established concepts from other languages.

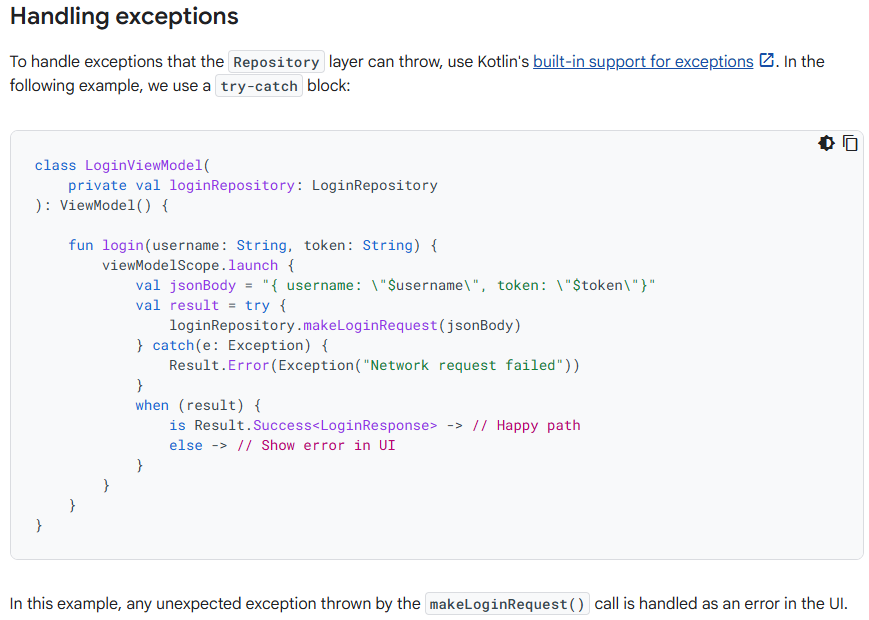
On Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive. Over 50% of professional developers who use coroutines have reported seeing increased productivity. This topic describes how you can use Kotlin coroutines to address these problems, enabling you to write cleaner and more concise app code.

**Features**

Coroutines is our recommended solution for asynchronous programming on Android. Noteworthy features include the following:

* **Lightweight**: You can run many coroutines on a single thread due to support for [*suspension*](https://kotlinlang.org/docs/reference/coroutines/basics.html), which doesn't block the thread where the coroutine is running. Suspending saves memory over blocking while supporting many concurrent operations.
* **Fewer memory leaks**: Use [*structured concurrency*](https://kotlinlang.org/docs/reference/coroutines/basics.html#structured-concurrency) to run operations within a scope.
* **Built-in cancellation support**: [Cancellation](https://kotlinlang.org/docs/reference/coroutines/cancellation-and-timeouts.html) is propagated automatically through the running coroutine hierarchy.
* **Jetpack integration**: Many Jetpack libraries include [extensions](https://developer.android.com/kotlin/ktx) that provide full coroutines support. Some libraries also provide their own [coroutine scope](https://developer.android.com/topic/libraries/architecture/coroutines) that you can use for structured concurrency.





[Kotlin coroutines](https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html) enable you to write clean, simplified asynchronous code that keeps your app responsive while managing long-running tasks such as network calls or disk operations.

This topic provides a detailed look at coroutines on Android. If you're unfamiliar with coroutines, be sure to read [Kotlin coroutines on Android](https://developer.android.com/kotlin/coroutines) before reading this topic.

**Manage long-running tasks**

Coroutines build upon regular functions by adding two operations to handle long-running tasks. In addition to invoke (or call) and return, coroutines add suspend and resume:

* suspend pauses the execution of the current coroutine, saving all local variables.
* resume continues execution of a suspended coroutine from the place where it was suspended.

You can call suspend functions only from other suspend functions or by using a coroutine builder such as launch to start a new coroutine.

The following example shows a simple coroutine implementation for a hypothetical long-running task:

suspend fun fetchDocs() { // Dispatchers.Main

val result = get("https://developer.android.com") // Dispatchers.IO for `get`

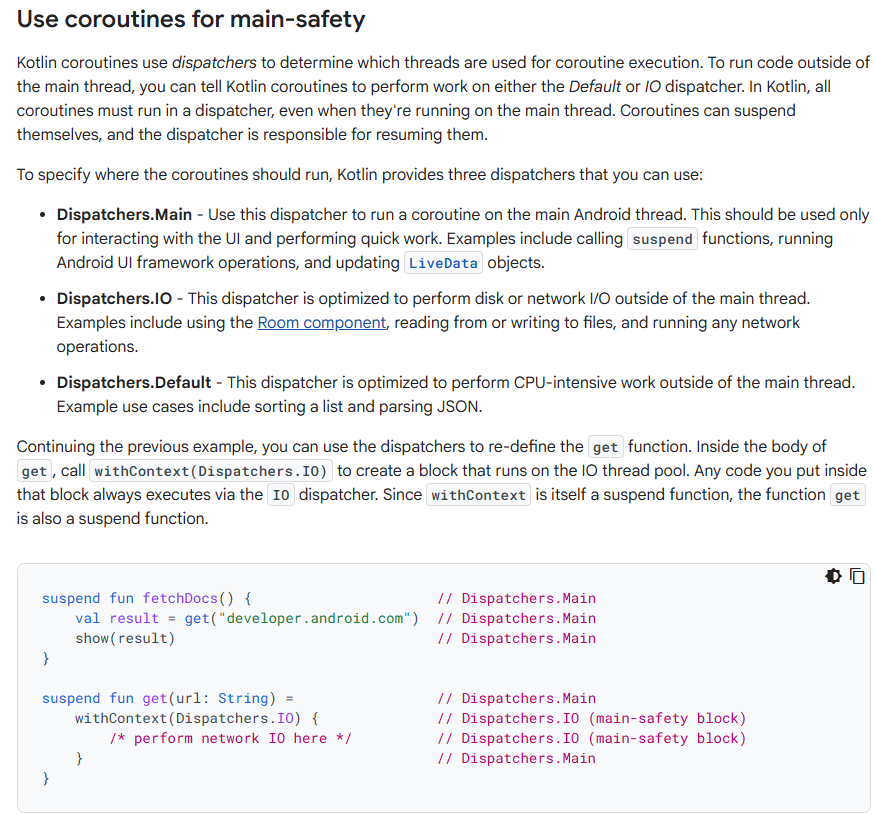
show(result) // Dispatchers.Main

}

suspend fun get(url: String) = withContext(Dispatchers.IO) { /\* ... \*/ }

In this example, get() still runs on the main thread, but it suspends the coroutine before it starts the network request. When the network request completes, get resumes the suspended coroutine instead of using a callback to notify the main thread.

Kotlin uses a *stack frame* to manage which function is running along with any local variables. When suspending a coroutine, the current stack frame is copied and saved for later. When resuming, the stack frame is copied back from where it was saved, and the function starts running again. Even though the code might look like an ordinary sequential blocking request, the coroutine ensures that the network request avoids blocking the main thread.



**Start a coroutine**

You can start coroutines in one of two ways:

* [launch](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch.
* [async](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) starts a new coroutine and allows you to return a result with a suspend function called await.

Typically, you should launch a new coroutine from a regular function, as a regular function cannot call await. Use async only when inside another coroutine or when inside a suspend function and performing parallel decomposition.

**Warning:** **launch** and **async** handle exceptions differently. Since **async** expects an eventual call to **await**, it holds exceptions and rethrows them as part of the **await** call. This means if you use **async** to start a new coroutine from a regular function, you might silently drop an exception. These dropped exceptions won't appear in your crash metrics or be noted in logcat. For more information, see [Cancellation and Exceptions in Coroutines](https://medium.com/androiddevelopers/cancellation-in-coroutines-aa6b90163629).