What's new in Kotlin 2.3.0

Edit page 16 December 2025

Released: December 16, 2025

The Kotlin 2.3.0 release is out! Here are the main highlights:

- **Language**: more stable and default features, unused return value checker, explicit backing fields, and changes to context-sensitive resolution.
- **Kotlin/JVM**: support for Java 25.
- **Kotlin/Native**: improved interop through Swift export, faster build time for release tasks, C and Objective-C library import in Beta.
- **Kotlin/Wasm**: fully qualified names and new exception handling proposal enabled by default, as well as new compact storage for Latin-1 characters.
- **Kotlin/JS**: new experimental suspend function export, `LongArray` representation, unified companion object access, and more.
- **Gradle**: compatibility with Gradle 9.0 and a new API for registering generated sources.
- **Compose compiler**: stack traces for minified Android applications.
- **Standard library**: stable time tracking functionality and improved UUID generation and parsing.

IDE support

The Kotlin plugins that support 2.3.0 are bundled in the latest versions of IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is change the Kotlin version to 2.3.0 in your build scripts.

See Update to a new release for details.

Language

Kotlin 2.3.0 focuses on feature stabilization, introduces a new mechanism for detecting unused return values, and improves context-sensitive resolution.

Stable features

In previous Kotlin releases, several new language features were introduced as Experimental and Beta. The following features have now graduated to Stable in Kotlin 2.3.0:

- Support for nested type aliases
- Data-flow-based exhaustiveness checks for `when` expressions

Features enabled by default

In Kotlin 2.3.0, support for `return` statements in expression bodies with explicit return types is now enabled by default.

See the full list of Kotlin language features and proposals.

Experimental

Unused return value checker

Kotlin 2.3.0 introduces the unused return value checker to help prevent ignored results. It warns you whenever an expression returns a value other than `Unit` or `Nothing` and isn't passed to a function, checked in a condition, or otherwise used.

The checker helps catch bugs where a function call produces a meaningful result that's silently dropped, which can lead to unexpected behavior or hard-to-trace issues.

> The checker ignores values returned from increment operations such as `++` and `--`.

Consider the following example:

```
fun formatGreeting(name: String): String {
    if (name.isBlank()) return "Hello, anonymous user!"
    if (!name.contains(' ')) {
        // The checker reports a warning that this result is ignored
        "Hello, " + name.replaceFirstChar(Char::titlecase) + "!"
    }
    val (first, last) = name.split(' ')
    return "Hello, $first! Or should I call you Dr. $last?"
}
```

In this example, a string is created but never used, so the checker reports it as an ignored result.

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle   Maven

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xreturn-value-checker=check")
    }
}
```

With this option, the checker only reports ignored results from expressions that are marked, such as most functions in the Kotlin standard library.

To mark your functions, use the `@MustUseReturnValues` annotation to mark the scope on which you want the checker to report ignored return values.

For example, you can mark an entire file:

```kotlin
// Marks all functions and classes in this file so the checker reports unused return values
@file:MustUseReturnValues

package my.project

fun someFunction(): String
```

Alternatively, you can mark a specific class:

```kotlin
// Marks all functions in this class so the checker reports unused return values
@MustUseReturnValues
class Greeter {
    fun greet(name: String): String = "Hello, $name"
}

fun someFunction(): Int = ...
```

You can also mark your entire project by adding the following compiler option to your build file:

Gradle   Maven

```kotlin
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xreturn-value-checker=full")
    }
}
```

With this setting, Kotlin automatically treats your compiled files as if they are annotated with `@MustUseReturnValues`, and the checker reports all return values for your project's functions.

You can suppress warnings on specific functions by marking them with the `@IgnorableReturnValue` annotation. Annotate functions where ignoring the return value is common and expected, such as `MutableList.add`:

```kotlin
@IgnorableReturnValue
fun <T> MutableList<T>.addAndIgnoreResult(element: T): Boolean {
    return add(element)
}
```

You can suppress a warning without marking the function itself as ignorable. To do this, assign the result to a special unnamed variable with an underscore (`_`):

```kotlin
// Non-ignorable function
fun computeValue(): Int = 42

fun main() {
    // Reports a warning: result is ignored
    computeValue()

    // Suppresses the warning only at this call site with a special unused variable
    val _ = computeValue()
}
```

For more information, see the feature's KEEP ↗.

We would appreciate your feedback in YouTrack ↗.

Experimental

Explicit backing fields

Kotlin 2.3.0 introduces explicit backing fields – a new syntax for explicitly declaring the underlying field that holds a property's value, in contrast to the existing implicit backing fields.

The new explicit syntax simplifies the common backing properties pattern where a property's internal type is different from its exposed API type. For example, you might use an `ArrayList` while exposing it as a read-only `List` or a `MutableList`. Previously, this required an additional private property.

With explicit backing fields, the implementation type of the `field` is directly defined within the property's scope. This eliminates the need for a separate private property and allows the compiler to automatically perform smart casting to the backing field's type within the same private scope.

Before:

```kotlin
private val _city = MutableStateFlow<String>("")
val city: StateFlow<String> get() = _city

fun updateCity(newCity: String) {
    _city.value = newCity
}
```

After:

```kotlin
val city: StateFlow<String>
    field = MutableStateFlow("")

fun updateCity(newCity: String) {
    // Smart casting works automatically
    city.value = newCity
}
```

This feature is Experimental. To opt in, add the following compiler option to your build file:

Gradle Maven

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xexplicit-backing-fields")
    }
}
```

For more information, see the feature's KEEP ↗.

We would appreciate your feedback in YouTrack ↗.

Experimental

Changes to context-sensitive resolution

Context-sensitive resolution is still Experimental, but we are continually improving the feature based on user feedback:

- The sealed and enclosing supertypes of the current type are now considered part of the contextual scope of the search. No other supertype scopes are considered. See the KT-77823 ↗ YouTrack issue for motivation and examples.
- When type operators and equalities are involved, the compiler now reports a warning if using context-sensitive resolution makes the resolution ambiguous. This can happen, for example, when a clashing declaration of a class is imported. See the KT-77821 ↗ YouTrack issue for motivation and examples.

See the full text of the current proposal in KEEP ↗.

Kotlin/JVM: Support for Java 25

Starting with Kotlin 2.3.0, the compiler can generate classes containing Java 25 bytecode.

Kotlin/Native

Kotlin 2.3.0 introduces improvements to the Swift export support and import of C and Objective-C libraries, as well as enhanced build times for release tasks.

Experimental

Improved interop through Swift export

Kotlin 2.3.0 further improves Kotlin's interoperability with Swift through Swift export, adding support for native enum classes and variadic function parameters.

Previously, Kotlin enums were exported as ordinary Swift classes. Now the mapping is direct, and you can use regular native Swift enums. For example:

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

val color = Color.RED
```

```swift
// Swift
public enum Color: Swift.CaseIterable,
Swift.LosslessStringConvertible, Swift.RawRepresentable {
    case RED, GREEN, BLUE

    var rgb: Int { get }
}
```

In addition, Kotlin's `vararg` functions are now directly mapped to Swift's variadic function parameters.

Such functions let you pass a variable number of arguments. This is useful when you don't know the number of arguments in advance or when you want to create or pass a collection without specifying its type. For example:

```
// Kotlin
fun log(vararg messages: String)
```

```
// Swift
public func log(messages: Swift.String...)
```

Generic types in variadic function parameters are not yet supported.

Beta

C and Objective-C library import is in Beta

Support for importing C and Objective-C libraries to Kotlin/Native projects is in Beta.

Full compatibility with different versions of Kotlin, dependencies, and Xcode is still not guaranteed, but the compiler now emits better diagnostics in the event of binary compatibility issues.

The import is not stable yet, and the `@ExperimentalForeignApi` opt-in annotation is still necessary when using C and Objective-C libraries in your project for certain things related to C and Objective-C interoperability, including:

- Some APIs in the `kotlinx.cinterop.*` package, required when working with native libraries or memory.
- All declarations in native libraries, except for platform libraries.

For compatibility and to prevent you from having to change your source code, the new stability status is not reflected in the annotation name.

For more information, see Stability of C and Objective-C library import.

Default explicit names in block types for Objective-C headers

Explicit parameter names in Kotlin's function types, introduced in Kotlin 2.2.20, are now the default for Objective-C headers exported from Kotlin/Native projects. These parameter names improve autocompletion suggestions in Xcode and help avoid Clang warnings.

Consider the following Kotlin code:

```
// Kotlin:
fun greetUser(block: (name: String) -> Unit) = block("John")
```

Kotlin forwards the parameter names from Kotlin function types to Objective-C block types, allowing Xcode to use them in suggestions:

```
// Objective-C:
greetUserBlock:^(NSString *name) {
    // ...
};
```

If you run into issues, you can disable explicit parameter names. To do that, add the following binary option to your `gradle.properties` file:

```
kotlin.native.binary.objcExportBlockExplicitParameterNames=false
```

Please report any problems in YouTrack ↗.

Faster build time for release tasks

Kotlin/Native has received several performance improvements in 2.3.0. They resulted in faster build times for release tasks like `linkRelease*`, for example `linkReleaseFrameworkIosArm64`.

According to our benchmarks, release builds can be up to 40% faster, depending on the project size. These improvements are most noticeable in Kotlin Multiplatform projects targeting iOS.

For more tips on improving project compilation times, see the documentation.

Changes to Apple target support

Kotlin 2.3.0 raises the minimum supported versions of Apple targets:

- For iOS and tvOS, from 12.0 to 14.0.
- For watchOS, from 5.0 to 7.0.

According to public data, usage of older versions is already very limited. This change simplifies overall Apple target maintenance for us and opens an opportunity to support Mac Catalyst ↗ in Kotlin/Native.

If you have to keep older versions in your project, add the following lines to your build file:

```
kotlin {

targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNative
Target>().configureEach {
        binaries.configureEach {
            freeCompilerArgs += "-Xoverride-konan-
properties=minVersion.ios=12.0"
            freeCompilerArgs += "-Xoverride-konan-
properties=minVersion.tvos=12.0"
        }
    }
}
```

Note that such a setup is not guaranteed to successfully compile and can break your app during building or at runtime.

This release also takes the next step in the deprecation cycle for Intel chip-based Apple targets.

Starting with Kotlin 2.3.0, the `macosX64`, `iosX64`, `tvosX64`, and `watchosX64` targets are demoted to support-tier 3. This means they are not guaranteed to be tested on CI, and source and binary compatibility between different compiler releases might not be provided. We plan to eventually remove support for the `x86_64` Apple targets in Kotlin 2.4.0.

For more information, see Kotlin/Native target support.

Kotlin/Wasm

Kotlin 2.3.0 enables by default fully qualified names for the Kotlin/Wasm targets, the new exception handling proposal for the `wasmWasi` target, and introduces compact storage for Latin-1 characters.

Fully qualified names enabled by default

On Kotlin/Wasm targets, fully qualified names (FQNs) were not enabled by default at runtime. You had to manually enable support for the `KClass.qualifiedName` property to use FQNs.

Only the class name (without the package) was accessible, which caused issues for code ported from the JVM to Wasm targets or for libraries that expected fully qualified names at runtime.

In Kotlin 2.3.0, the `KClass.qualifiedName` property is enabled by default on Kotlin/Wasm targets. This means that FQNs are available at runtime without any additional configuration.

Enabling FQNs by default improves code portability and makes runtime errors more informative by displaying the fully qualified name.

This change does not increase the size of the compiled Wasm binary, thanks to compiler optimizations that reduce metadata by using compact storage for Latin-1 string literals.

Compact storage for Latin-1 characters

Previously, Kotlin/Wasm stored string literal data as is, meaning every character was encoded in UTF-16. This was not optimal for text containing only, or predominantly, Latin-1 characters.

Starting with Kotlin 2.3.0, the Kotlin/Wasm compiler stores string literals containing only Latin-1 characters in UTF-8 format.

This optimization significantly reduces metadata, as experiments on JetBrains' KotlinConf application ↗ have shown. It results in:

- Up to 13% smaller Wasm binaries compared to builds without the optimization.
- Up to 8% smaller Wasm binaries even when fully qualified names are enabled, compared to earlier versions that did not store them.

This compact storage is important for web environments where download and startup times matter. Additionally, this optimization removes the size barrier that previously prevented storing fully qualified names for classes and enabling `KClass.qualifiedName` by default.

This change is enabled by default, and no further action is required.

New exception handling proposal enabled by default for `wasmWasi`

Previously, Kotlin/Wasm used the legacy exception handling proposal ↗ for all targets, including `wasmWasi`. However, most standalone WebAssembly virtual machines (VMs) are aligning with the new version of the exception handling proposal ↗.

Starting with Kotlin 2.3.0, the new WebAssembly exception handling proposal is enabled by default for the `wasmWasi` target, ensuring better compatibility with modern WebAssembly runtimes.

For the `wasmWasi` target, the change is safe to introduce early because applications targeting it usually run in a less diverse runtime environment (often running on a single specific VM), which is typically controlled by the user, reducing the risk of compatibility issues.

The new exception handling proposal remains off by default for the `wasmJs` target. You can enable it manually by using the `-Xwasm-use-new-exception-proposal` compiler option.

Kotlin/JS

Kotlin 2.3.0 brings experimental support for exporting suspend functions to JavaScript and the `BigInt64Array` type to represent Kotlin's `LongArray` type.

With this release, you can now access companion objects inside interfaces in a unified way, use the `@JsStatic` annotation in interfaces with companion objects, the `@JsQualifier` annotation in individual functions and classes, and default exports through a new annotation, `@JsExport.Default`.

Experimental

New export of suspend function with `JsExport`

Previously, the `@JsExport` annotation did not allow exporting suspend functions (or classes and interfaces containing such functions) to JavaScript. You had to manually wrap each suspend function, which was cumbersome and prone to errors.

Starting with Kotlin 2.3.0, suspend functions can be exported directly to JavaScript using the `@JsExport` annotation.

Enabling suspend function exports reduces boilerplate and improves interoperability between Kotlin/JS and JavaScript/TypeScript (JS/TS). Kotlin's async functions can now be called directly from JS/TS without extra code.

To enable this feature, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xenable-suspend-function-exporting")
    }
}
```

Once enabled, classes and functions marked with the `@JsExport` annotation can include suspend functions without additional wrappers.

They can be consumed as regular JavaScript async functions and can be overridden as an async function, as well:

```
@JsExport
open class Foo {
    suspend fun foo() = "Foo"
}
```

```
class Bar extends Foo {
    override async foo(): Promise<string> {
        return "Bar"
    }
}
```

This feature is Experimental. We would appreciate your feedback in our issue tracker, YouTrack ↗.

Experimental

Usage of the `BigInt64Array` type to represent Kotlin's `LongArray` type

Previously, Kotlin/JS represented its `LongArray` as a JavaScript `Array<bigint>`. This approach worked but was not ideal for interoperability with JavaScript APIs that expect typed arrays.

Starting with this release, Kotlin/JS now uses JavaScript's built-in `BigInt64Array` type to represent Kotlin's `LongArray` values when compiling to JavaScript.

Using `BigInt64Array` simplifies interop with JavaScript APIs that use typed arrays. It also allows APIs that accept or return `LongArray` to be exported more naturally from Kotlin to JavaScript.

To enable this feature, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    js {
        // ...
        compilerOptions {
            freeCompilerArgs.add("-Xes-long-as-bigint")
        }
    }
}
```

This feature is Experimental. We would appreciate your feedback in our issue tracker, YouTrack ↗.

Unified companion object access across JS module systems

Previously, when you exported a Kotlin interface with a companion object to JavaScript/TypeScript using the `@JsExport` annotation, consuming the interface in TypeScript worked differently for ES modules compared to other module systems.

As a result, you had to adjust the consumption of the output on the TypeScript side, depending on the module system.

Consider this Kotlin code:

```kotlin
@JsExport
interface Foo {
    companion object {
        fun bar() = "OK"
    }
}
```

You had to call it differently depending on the module system:

```
// Worked for CommonJS, AMD, UMD, and no modules
Foo.bar()

// Worked for ES modules
Foo.getInstance().bar()
```

In this release, Kotlin unifies companion-object exports across all JavaScript module systems.

Now, for every module system (ES modules, CommonJS, AMD, UMD, no modules), companion objects inside interfaces are always accessed the same way (just like companions in classes):

```
// Works for all module systems
Foo.Companion.bar()
```

This improvement also fixes collection interoperability. Before, collection factory functions had to be accessed differently depending on the module system:

```
// Worked for CommonJS, AMD, UMD, and no modules
KtList.fromJsArray([1, 2, 3])

// Worked for ES modules
KtList.getInstance().fromJsArray([1, 2, 3])
```

Now, accessing collection factory functions is similar across all the module systems:

```
// Works for all module systems
KtList.fromJsArray([1, 2, 3])
```

This change reduces inconsistent behavior between module systems and avoids bugs and interoperability issues.

This feature is enabled by default.

Support for `@JsStatic` annotations in interfaces with companion objects

Previously, the `@JsStatic` annotation was not allowed inside exported interfaces with companion objects.

For example, the following code would produce an error because only members of class companion objects can be annotated with `@JsStatic` :

```
@JsExport
interface Foo {
    companion object {
        @JsStatic // Error
        fun bar() = "OK"
    }
}
```

In this case, you had to drop the `@JsStatic` annotation and access the companion from JavaScript (JS) in the following way:

```
// For all module systems
Foo.Companion.bar()
```

Now, the `@JsStatic` annotation is supported in interfaces with companion objects. You can use this annotation on such companions and call the function directly from JS, just as you would for classes:

```
// For all module systems
Foo.bar()
```

This change simplifies API consumption in JS, allows static factory methods on interfaces, and removes inconsistencies between classes and interfaces.

This feature is enabled by default.

`@JsQualifier` annotation allowed in individual functions and classes

Previously, you could only apply the `@JsQualifier` annotation at the file level, requiring all external JavaScript (JS) declarations to be placed in separate files.

Starting from Kotlin 2.3.0, you can apply the `@JsQualifier` annotation directly to individual functions and classes, just like the `@JsModule` and `@JsNonModule` annotations.

For example, you can now write the following external function code next to regular Kotlin declarations in the same file:

```
@JsQualifier("jsPackage")
private external fun jsFun()
```

This change simplifies Kotlin/JS interop, keeps your project structure cleaner, and aligns Kotlin/JS with how other platforms handle external declarations.

This feature is enabled by default.

Support for JavaScript default exports

Previously, Kotlin/JS could not generate JavaScript's default exports from Kotlin code. Instead, Kotlin/JS only generated named exports, for example:

```
export { SomeDeclaration };
```

If you required a default export, you had to use workarounds inside the compiler, such as placing the `@JsName` annotation with `default` and a space as an argument:

```
@JsExport
@JsName("default ")
class SomeDeclaration
```

Kotlin/JS now supports default exports directly through a new annotation:

```
@JsExport.Default
```

When you apply this annotation to a Kotlin declaration (class, object, function, or property), the generated JavaScript automatically includes an `export default` statement for ES modules:

```
export default HelloWorker;
```

For module systems different from ES modules, the new `@JsExport.Default` annotation works similarly to the regular `@JsExport` annotation.

This change enables Kotlin code to conform to JavaScript conventions and is particularly important for platforms like Cloudflare Workers, or frameworks like `React.lazy`.

This feature is enabled by default. You only need to use the `@JsExport.Default` annotation.

Gradle

Kotlin 2.3.0 is fully compatible with Gradle 7.6.3 through 9.0.0. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

In addition, the minimum supported Android Gradle plugin version is now 8.2.2, and the maximum supported version is 8.13.0.

Kotlin 2.3.0 also introduces a new API for registering generated sources in your Gradle projects.

Experimental

New API for registering generated sources in Gradle projects

Kotlin 2.3.0 introduces a new Experimental API in the `KotlinSourceSet` ↗ interface, which you can use to register generated sources in your Gradle projects.

This new API is a quality-of-life improvement that helps IDEs distinguish between generated code and regular source files. The API allows IDEs to highlight the generated code differently in the UI and to trigger generation tasks when the project is imported. We are currently working on adding this support in IntelliJ IDEA. The API is also especially useful for third-party plugins or tools that generate code, such as KSP (Kotlin Symbol Processing).

For more information, see Register generated sources.

Standard library

Kotlin 2.3.0 stabilizes the new time tracking functionality, `kotlin.time.Clock` and `kotlin.time.Instant`, and adds several improvements to the Experimental UUID API.

Experimental

Improved UUID generation and parsing

Kotlin 2.3.0 introduces several improvements for the UUID API, including:

- Support for returning `null` when parsing invalid UUIDs
- New functions to generate v4 and v7 UUIDs
- Support for generating v7 UUIDs for specific timestamps

UUID support in the standard library is Experimental but planned to be stabilized in the future ↗. To opt in, use the `@OptIn(ExperimentalUuidApi::class)` annotation or add the following compiler option to your build file:

Gradle   Maven

```
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-opt-
in=kotlin.uuid.ExperimentalUuidApi")
    }
}
```

We would appreciate your feedback in YouTrack ↗ or in the related Slack channel ↗.

Support for returning `null` when parsing invalid UUIDs

Kotlin 2.3.0 introduces new functions to create a `Uuid` instance from a string, which return `null` instead of throwing an exception if the string isn't a valid UUID.

These functions include the following:

- `Uuid.parseOrNull()` – parses UUIDs in either hex-and-dash or hexadecimal format.
- `Uuid.parseHexDashOrNull()` – parses UUIDs only in hex-and-dash format, and returns `null` otherwise.
- `Uuid.parseHexOrNull()` – parses UUIDs only in plain hexadecimal format, and returns `null` otherwise.

Here's an example:

```kotlin
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

@OptIn(ExperimentalUuidApi::class)
fun main() {
    val valid = Uuid.parseOrNull("550e8400-e29b-41d4-a716-446655440000
    println(valid)
    // 550e8400-e29b-41d4-a716-446655440000

    val invalid = Uuid.parseOrNull("not-a-uuid")
    println(invalid)
    // null
    val hexDashValid = Uuid.parseHexDashOrNull("550e8400-e29b-41d4-a71
446655440000")
    println(hexDashValid)
    // 550e8400-e29b-41d4-a716-446655440000


    val hexDashInvalid = Uuid.parseHexDashOrNull("550e8400e29b41d4a716
    println(hexDashInvalid)
    // null
}
```

Open in Playground → Target: JVM Running on v.2.3.0

New functions to generate v4 and v7 UUIDs

Kotlin 2.3.0 introduces two new functions for generating UUIDs: `Uuid.generateV4()` and `Uuid.generateV7()`.

Use the `Uuid.generateV4()` function to generate version 4 UUIDs, or the `Uuid.generateV7()` function to generate version 7 UUIDs.

The `Uuid.random()` function remains unchanged and still generates version 4 UUIDs, just like `Uuid.generateV4()`.

Here's an example:

```
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

@OptIn(ExperimentalUuidApi::class)
fun main() {
    // Generates a v4 UUID
    val v4 = Uuid.generateV4()
    println(v4)

    // Generates a v7 UUID
    val v7 = Uuid.generateV7()
    println(v7)

    // Generates a v4 UUID
```

```kotlin
    val random = Uuid.random()
    println(random)
}
```

Open in Playground → Target: JVM Running on v.2.3.0

Support for generating v7 UUIDs for specific timestamps

Kotlin 2.3.0 introduces the new `Uuid.generateV7NonMonotonicAt()` function, which you can use to generate a version 7 UUID for a specific moment in time.

> Unlike `Uuid.generateV7()`, `Uuid.generateV7NonMonotonicAt()` doesn't guarantee monotonic ordering, so multiple UUIDs created for the same timestamp might not be sequential.

Use this function when you need identifiers tied to a known timestamp, such as when recreating event IDs or generating database entries that reflect when something originally occurred.

For example, to create a version 7 UUID for a particular instant, use the following code:

```kotlin
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid
import kotlin.time.ExperimentalTime
import kotlin.time.Instant

@OptIn(ExperimentalUuidApi::class, ExperimentalTime::class)
fun main() {
    val timestamp = Instant.fromEpochMilliseconds(1577836800000) // 20
01T00:00:00Z

    // Generates a v7 UUID for the specified timestamp (without monoto
guarantees)
    val v7AtTimestamp = Uuid.generateV7NonMonotonicAt(timestamp)
    println(v7AtTimestamp)
}
```

Open in Playground → Target: JVM Running on v.2.3.0

Compose compiler: Stack traces for minified Android applications

Starting from Kotlin 2.3.0, the compiler outputs ProGuard mappings for Compose stack traces when applications are minified by R8. This expands the experimental stack traces feature that was previously only available in debuggable variants.

The release variant of stack traces contains group keys that can be used to identify composable functions in minified applications without the overhead of recording source information at runtime. The group key stack traces require your application to be built with the Compose runtime 1.10 or newer.

To enable group key stack traces, add the following line before initializing any `@Composable` content:

```
Composer.setDiagnosticStackTraceMode(ComposeStackTraceMode.GroupKeys
)
```

With these stack traces enabled, the Compose runtime will append its own stack trace after a crash is captured during composition, measure, or draw passes, even when the app is minified:

```
java.lang.IllegalStateException: <message>
        at <original trace>
    Suppressed: androidx.compose.runtime.DiagnosticComposeException:
Composition stack when thrown:
        at $$compose.m$123(SourceFile:1)
        at $$compose.m$234(SourceFile:1)
        ...
```

Stack traces produced by Jetpack Compose 1.10 in this mode only contain group keys that still have to be deobfuscated. This is addressed in the Kotlin 2.3.0 release with the Compose Compiler Gradle plugin, which now appends group key entries to the ProGuard mapping files produced by R8. If you see new warnings in cases when the compiler fails to create mappings for some functions, please report them to the Google IssueTracker ↗.

> The Compose Compiler Gradle plugin only creates deobfuscation mappings for the group key stack traces when R8 is enabled for the build due to dependencies on R8 mapping files.

By default, the mapping file Gradle tasks run regardless of whether you enable the traces. If they cause problems in your build, you can disable the feature entirely. Add the following property in the `composeCompiler {}` block of your Gradle configuration:

```
composeCompiler {
    includeComposeMappingFile.set(false)
}
```

> There is a known issue with some code not showing up in stack traces for project files supplied by the Android Gradle plugin: KT-83099 ↗.

Please report any problems encountered to the Google IssueTracker ↗.

Breaking changes and deprecations

This section highlights important breaking changes and deprecations. For a complete overview, see our Compatibility guide.

- Starting with Kotlin 2.3.0, the compiler no longer supports `-language-version=1.8`. There's also no support for `-language-version=1.9` on non-JVM platforms.
- Language feature sets older than 2.0 (excluding 1.9 for the JVM platform) aren't supported, but the language itself remains fully backward-compatible with Kotlin 1.0.

If you use both the `kotlin-dsl` **and** the `kotlin("jvm")` plugin in your Gradle project, you may see a Gradle warning about an unsupported Kotlin plugin version. See our compatibility guide for guidance on migration steps.

- In Kotlin Multiplatform, support for the Android target is now available through Google's `com.android.kotlin.multiplatform.library` plugin ↗. Migrate your projects with Android targets to the new plugin and rename your `androidTarget` blocks to `android`.

- If you continue using the Kotlin Multiplatform Gradle plugin for Android targets with Android Gradle plugin (AGP) 9.0.0 or later, you see a configuration error when using the `androidTarget` block, along with diagnostic messages that provide guidance on how to migrate. For more information, see Migrate to Google's plugin for Android targets ↗.

- AGP 9.0.0 includes built-in support for Kotlin ↗. Starting with Kotlin 2.3.0, you see a configuration error if you use this version of AGP with the `kotlin-android` plugin, because the plugin is no longer necessary. New diagnostic messages are available to help you migrate. If you use older AGP versions, you see a deprecation warning.

- Support for the Ant build system is no longer available.

Documentation updates

Kotlin Multiplatform documentation has moved to kotlinlang.org. Now you can switch between Kotlin and KMP docs in one place. We've also refreshed the table of contents for the language guide and introduced new navigation.

Other notable changes since the last Kotlin release:

- KMP overview ↗ – explore the Kotlin Multiplatform ecosystem on a single page.

- Kotlin Multiplatform quickstart ↗ – learn how to set up an environment with the KMP IDE plugin.

* What's new in Compose Multiplatform 1.9.3 ↗ – learn about the highlights from the latest release.
* Get started with Kotlin/JS – create a web application for the browser using Kotlin/JavaScript.
* Classes – learn the basics and best practices of using classes in Kotlin.
* Extensions – learn how you can extend classes and interfaces in Kotlin.
* Coroutines basics – explore key coroutine concepts and learn how to create your first coroutines.
* Cancellation and timeouts – learn how coroutine cancellation works and how to make coroutines respond to cancellation.
* Kotlin/Native libraries – see how to produce `klib` library artifacts.
* Kotlin Notebook overview – create interactive notebook documents with the Kotlin Notebook plugin.
* Add Kotlin to a Java project – configure a Java project to use both Kotlin and Java.
* Test Java code with Kotlin – test a mixed Java-Kotlin project with JUnit.
* New case studies page ↗ – discover how different companies apply Kotlin.

How to update to Kotlin 2.3.0

The Kotlin plugin is distributed as a bundled plugin in IntelliJ IDEA and Android Studio.

To update to the new Kotlin version, change the Kotlin version to 2.3.0 in your build scripts.

Was this page helpful? Yes No