Kotlin 1.4 Online Event

Kotlin 1.4 Language Features

Svetlana Isakova

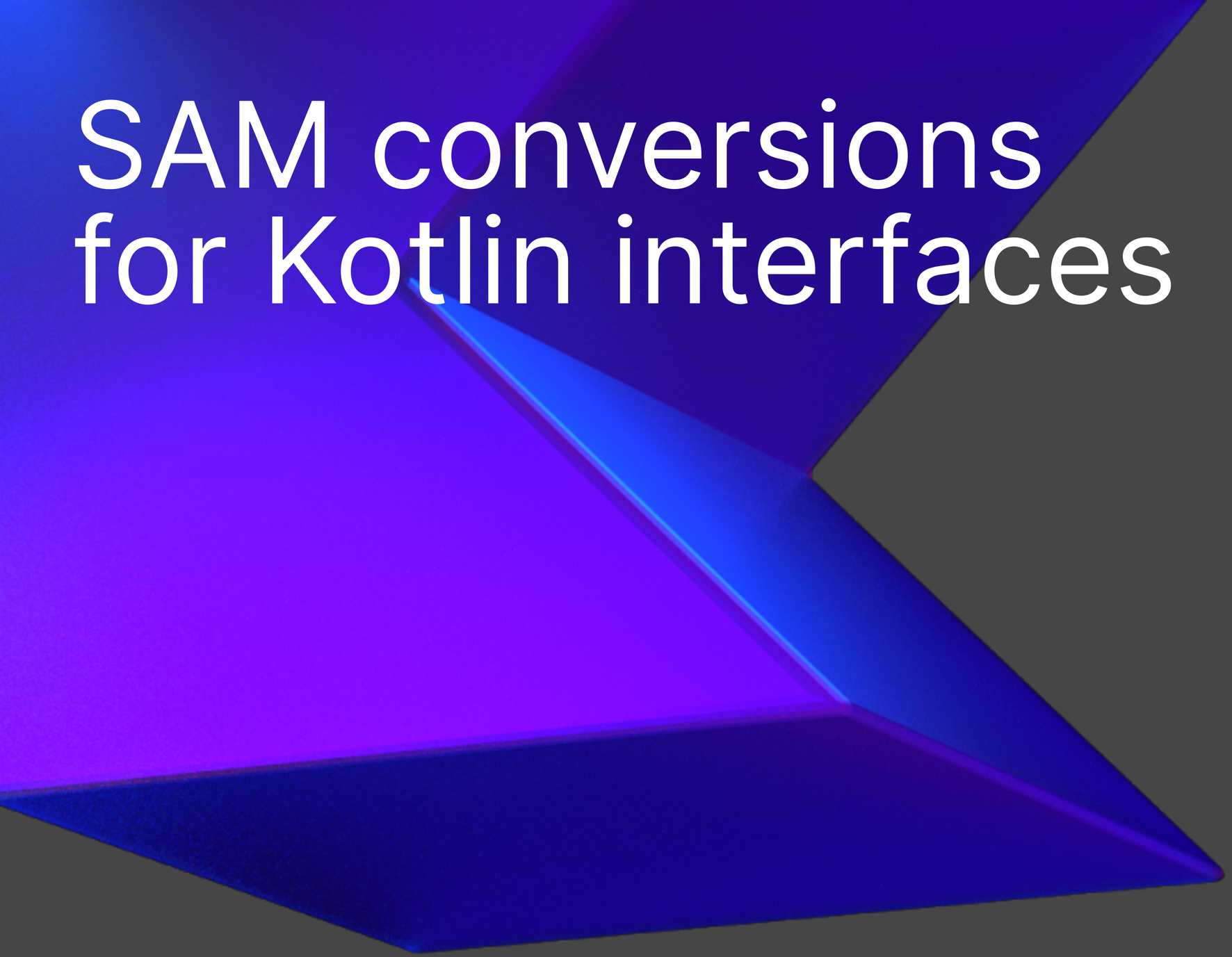
@sveta_isakova

October 12, 2020

Kotlin 1.4 Language Features

- SAM conversions for Kotlin classes
- Explicit API mode
- Trailing comma
- Break and continue inside `when` expressions
- Mixing named and positional arguments
- New type inference
- Unified exception type for null checks

SAM conversions for Kotlin interfaces



SAM = Single abstract method

```
interface Action {
    fun run()
}
```

SAM conversion

Ability to pass a lambda or callable reference when a SAM interface is expected

SAM conversion for a Java interface

Java:

```java
public interface Action {
    void run();
}

public static void runAction(Action action) {
    action.run();
}
```

Kotlin:

```kotlin
runAction {
    println("I'm Kotlin 1.3")
}
```

SAM conversion for a Kotlin interface

~~Java:~~ Kotlin:

```
interface Action {
    fun run()
}

fun runAction(a: Action) = a.run()
```

Kotlin:

```
runAction {
    println("I'm Kotlin 1.3")
}
```

Doesn’t work. Please use function types!

KT-7770 SAM for Kotlin classes

SAM conversion only works
for **Java interfaces**

That's **by design**: everyone
should use functional types

But that is **inconvenient**!

211 👍
50+comments

Functional interfaces

```
fun interface Action {
    fun run()
}
```

Functional interfaces

```
fun interface Action {
    fun run()
}

fun runAction(a: Action) = a.run()

runAction {
    println("Hello, Kotlin 1.4!")
}
```

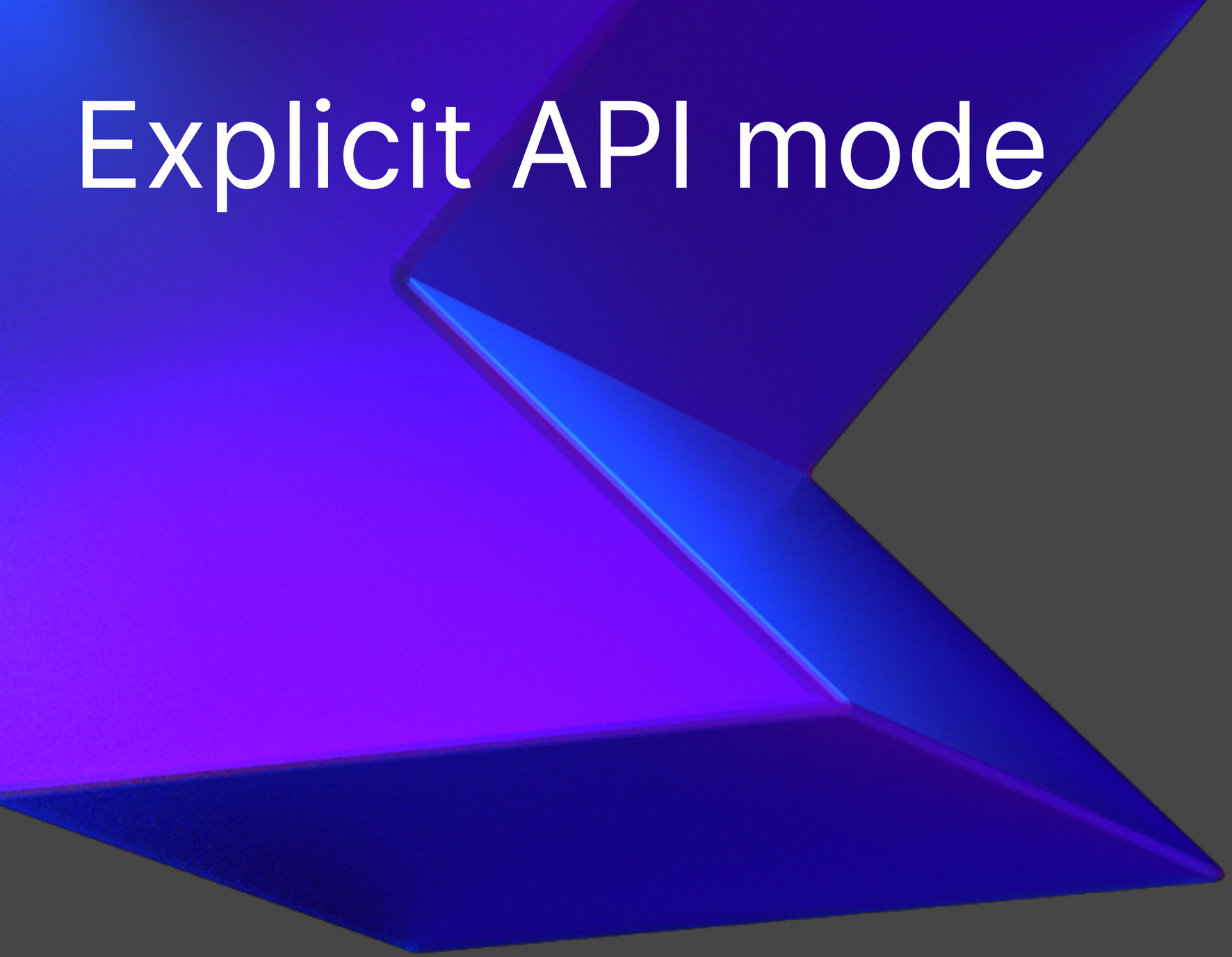
Why functional interfaces?

Fun interfaces must have exactly one abstract method

```
fun interface Action {
    fun run()
    fun runWithDelay()
}
```

No longer a SAM interface!

Explicit API mode



Kotlin style guide

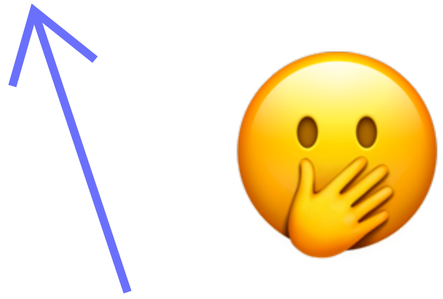
Ability to pass a lambda or callable reference when a SAM interface is expected

kotlinlang.org

Specify member visibility

to avoid accidentally exposing declarations as public APIs

```
fun privateFun() { … }
```



It’s PUBLIC!

Specify member visibility

to avoid accidentally exposing declarations as public APIs

```
private fun privateFun() { … }

public fun publicFun() { … }
```

Explicitly specify function return types and property types

to avoid accidentally changing the return type when the implementation changes

```
fun getAnswer(finished: Boolean) =
    if (finished) "42" else "unknown"
```

: String

Explicitly specify function return types and property types

to avoid accidentally changing the return type when the implementation changes

```
fun getAnswer(finished: Boolean) =
    if (finished) 42 else "unknown"
```

: Any

Explicit API mode

```
fun getAnswer(finished: Boolean) =
    if (finished) 42 else "unknown"
```

The visibility must be specified in explicit API mode
The return type must be specified in explicit API mode

Explicit API mode

```
public fun getAnswer(finished: Boolean): Any =
    if (finished) 42 else "unknown"
```

Strict mode: reports errors

build.gradle.kts:

```
kotlin {
    explicitApi()
}
```

build.gradle:

```
kotlin {
    explicitApi = 'strict'
}
```

Warning mode: reports warnings

build.gradle.kts:

```
kotlin {
    explicitApiWarning()
}
```

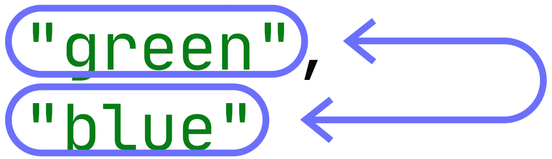
build.gradle:

```
kotlin {
    explicitApi = 'warning'
}
```

Trailing comma

Argument list

```
val colors = listOf(
    "red",
    "green",
    "blue"
)
```



Argument list

```
val colors = listOf(
    "red",
    "blue"
    "green",....
)
```

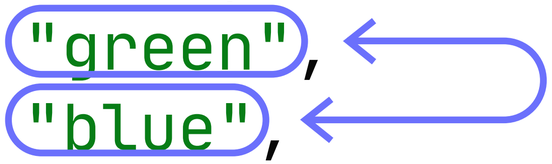
Compiler error: Expecting ','

Argument list

```
val colors = listOf(
    "red",
    "green",
    "blue",
)
```

Argument list

```
val colors = listOf(
    "red",
    "green",
    "blue",
)
```



Argument list

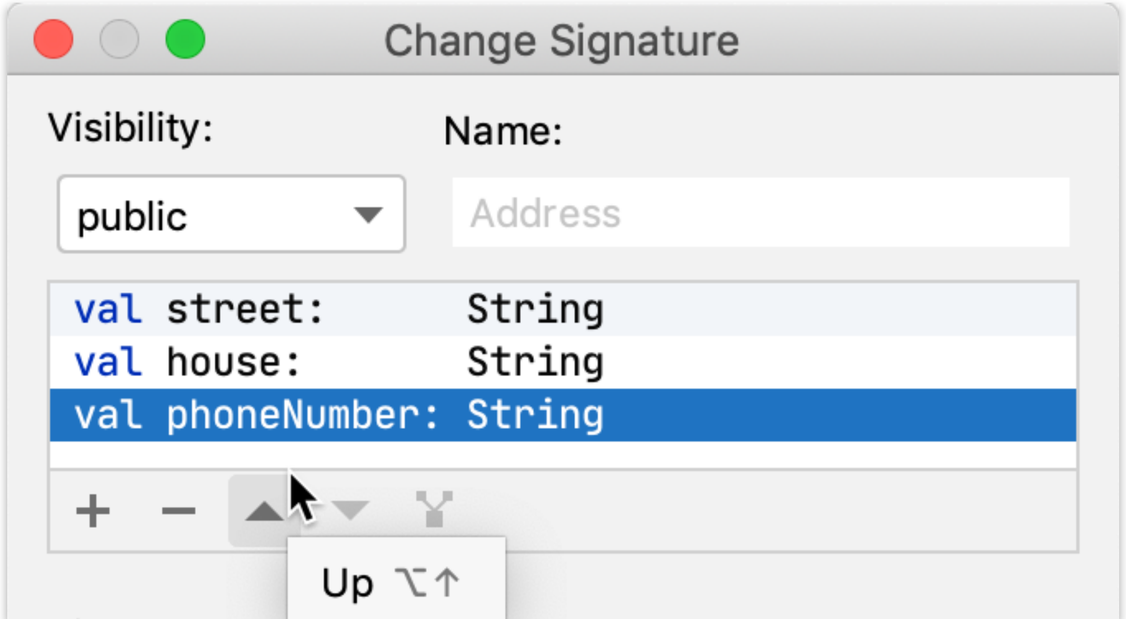
```
val colors = listOf(
    "red",
    "blue",
    "green",
)
```

Function declaration

```
fun displayRectangle(
    color: Color,
    width: Int,
    height: Int,
) {
  ...
}
```

Class declaration

```
data class Contact(
    val address: String,
    val phoneNumber: String,
    val email: String,
)
```



Break & continue in when expressions

Kotlin 1.3 forbids break and continue in when expressions

```
fun foo(list: List<Int>) {
    for (i in list) {
        when (i) {
            42 → continue
            else → println(i)
        }
    }
}
```

'break' and 'continue' are not allowed in 'when' statements.
Consider using labels to continue/break from the outer loop.

Workaround: using labels

```
fun foo(list: List<Int>) {
    l@ for (i in list) {
        when (i) {
            42 -> continue@l
            else -> println(i)
        }
    }
}
```

Kotlin 1.4 allows break and continue in when expressions

```
fun foo(list: List<Int>) {
    for (i in list) {
        when (i) {
            42 -> continue
            else -> println(i)
        }
    }
}
```

Kotlin 1.4 allows break and continue in when expressions

```
fun foo(list: List<Int>) {
    for (i in list) {
        when (i) {
            42 -> break
            else -> println(i)
        }
    }
}
```

Mixing named and positional arguments

Kotlin 1.3: using named args

```
drawRectangle(
    width = 10, height = 20, color = Color.BLUE
)
```

helpful helpful redundant

Kotlin 1.3: mixing not allowed

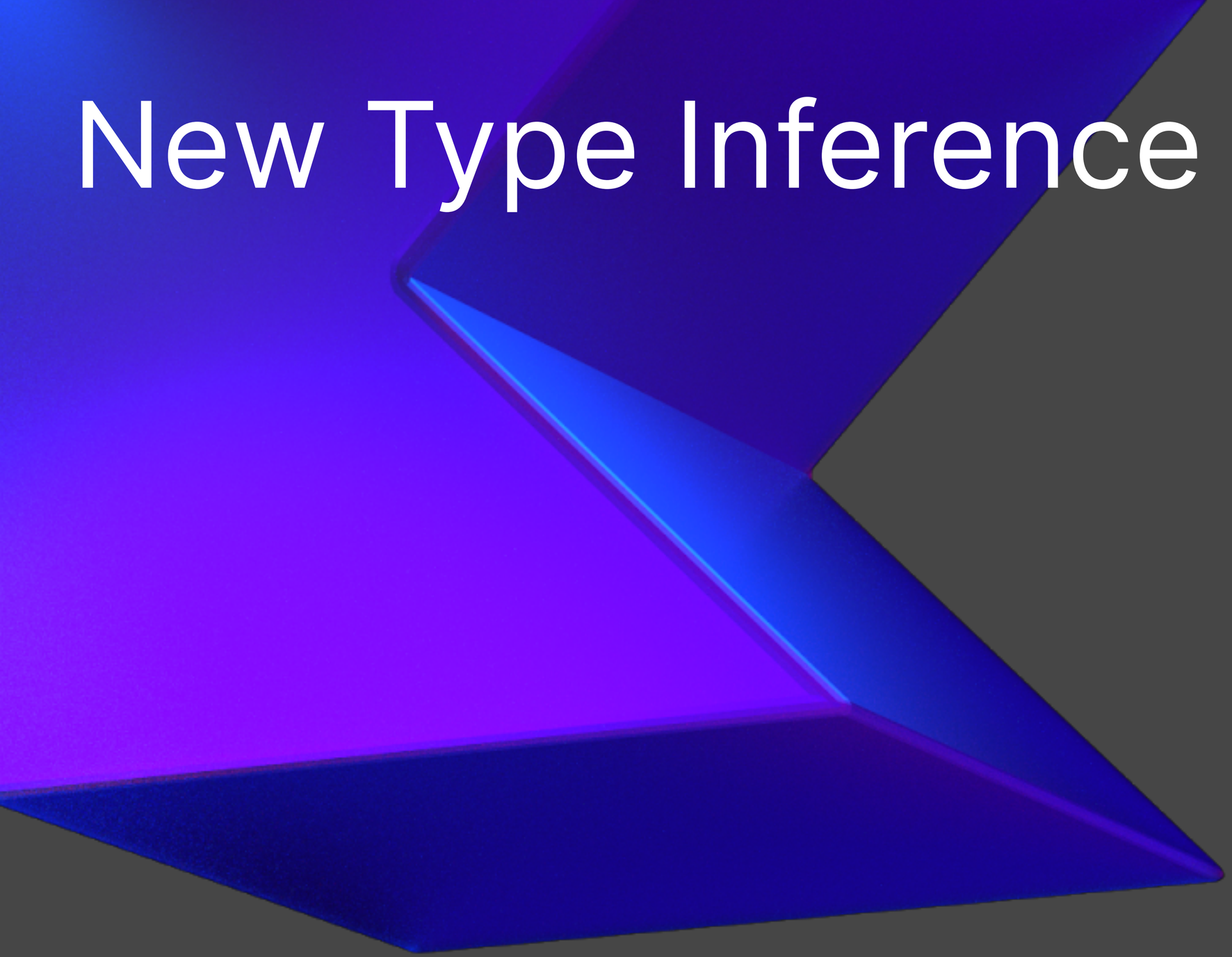
```
drawRectangle(
    width = 10, height = 20, Color.BLUE
)
```

Mixing named and positioned arguments is not allowed.

Kotlin 1.4

```
drawRectangle(
    width = 10, height = 20, color = Color.BLUE
)
```

New Type Inference



New Type Inference

- Infers types automatically in more use-cases
- Supports smart casts in more complicated scenarios
- Supports more cases for using callable references
- And much more

Kotlin 1.3: lambda parameter type

```
val rulesMap: Map<String, (String?) → Boolean> =
    mapOf(
        "weak" to { it ≠ null },
        "medium" to { !it.isNullOrBlank() },
        "strong" to { it ≠ null &&
            "^[a-zA-Z0-9]+$".toRegex().matches(it)
        }
    )
```

Kotlin 1.3: lambda parameter type

```
val rulesMap: Map<String, (String?) → Boolean> =
    mapOf(
        "weak" to { it ≠ null },
        "medium" to { !it.isNullOrBlank() },
        "strong" to { it ≠ null &&
            "^[a-zA-Z0-9]+$".toRegex().matches(it)
        }
    )
```

Kotlin 1.3: lambda parameter type

```
val rulesMap: Map<String, (String?) → Boolean> =
    mapOf(
        "weak" to { it ≠ null },
        "medium" to { !it.isNullOrBlank() },
        "strong" to { it ≠ null &&
            "^[a-zA-Z0-9]+$".toRegex().matches(it)
        }
    )
```

Kotlin 1.3: lambda parameter type

```
val rulesMap: Map<String, (String?) → Boolean> =
    mapOf(
        "weak" to { it ≠ null },
        "medium" to { !it.isNullOrBlank() },
        "strong" to { it ≠ null &&
            "^[a-zA-Z0-9]+$".toRegex().matches(it)
        }
    )
```

: String?

Kotlin 1.4: lambda parameter type

```
val rulesMap: Map<String, (String?) → Boolean> =
    mapOf(
        "weak" to { it ≠ null },
        "medium" to { !it.isNullOrBlank() },
        "strong" to { it ≠ null &&
            "^[a-zA-Z0-9]+$".toRegex().matches(it)
        }
    )
```

Kotlin 1.3: lambda’s last expression

```
                :String?
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str
}
```

Kotlin 1.4: lambda’s last expression

```
                :String?
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str
}
```

References to functions with default argument values

```
fun foo(i: Int = 0): String = "$i!"

apply(::foo)
```

:(Int)→String

References to functions with default argument values

```
fun foo(i: Int = 0): String = "$i!"

apply(::foo)
```

:()→String

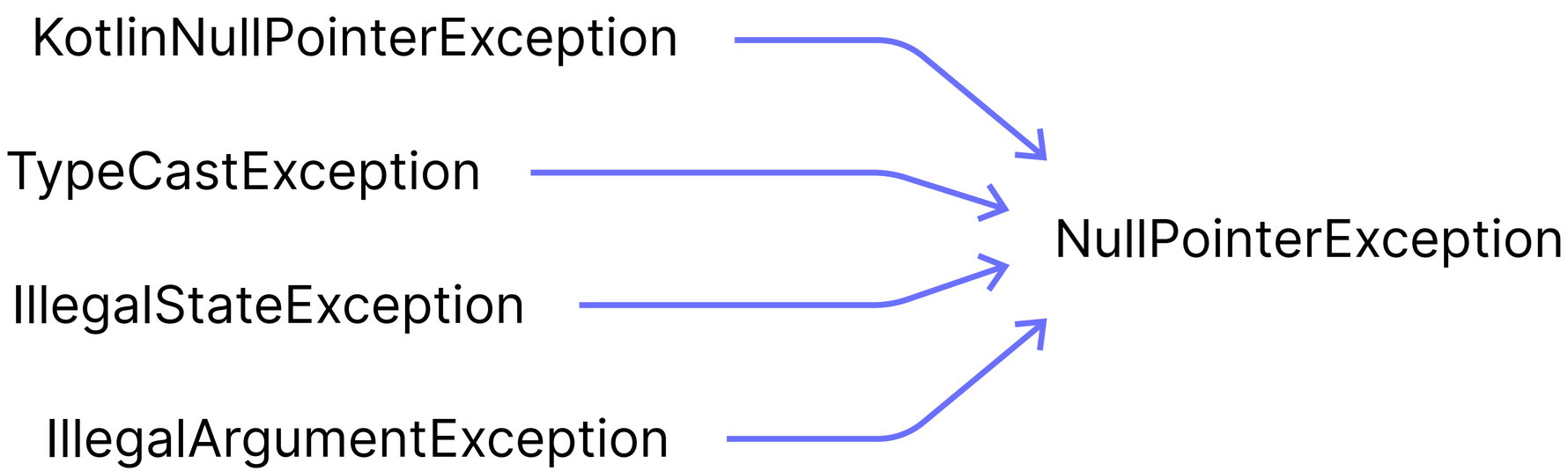
References to functions with default argument values

```
fun foo(i: Int = 0): String = "$i!"


apply(::foo)  // 0!

fun apply(f: () → String): String = f()
```

Unified exception type for null checks

Unified exception type

!!, as Type, platform-typed expression
null checks, parameter null checks



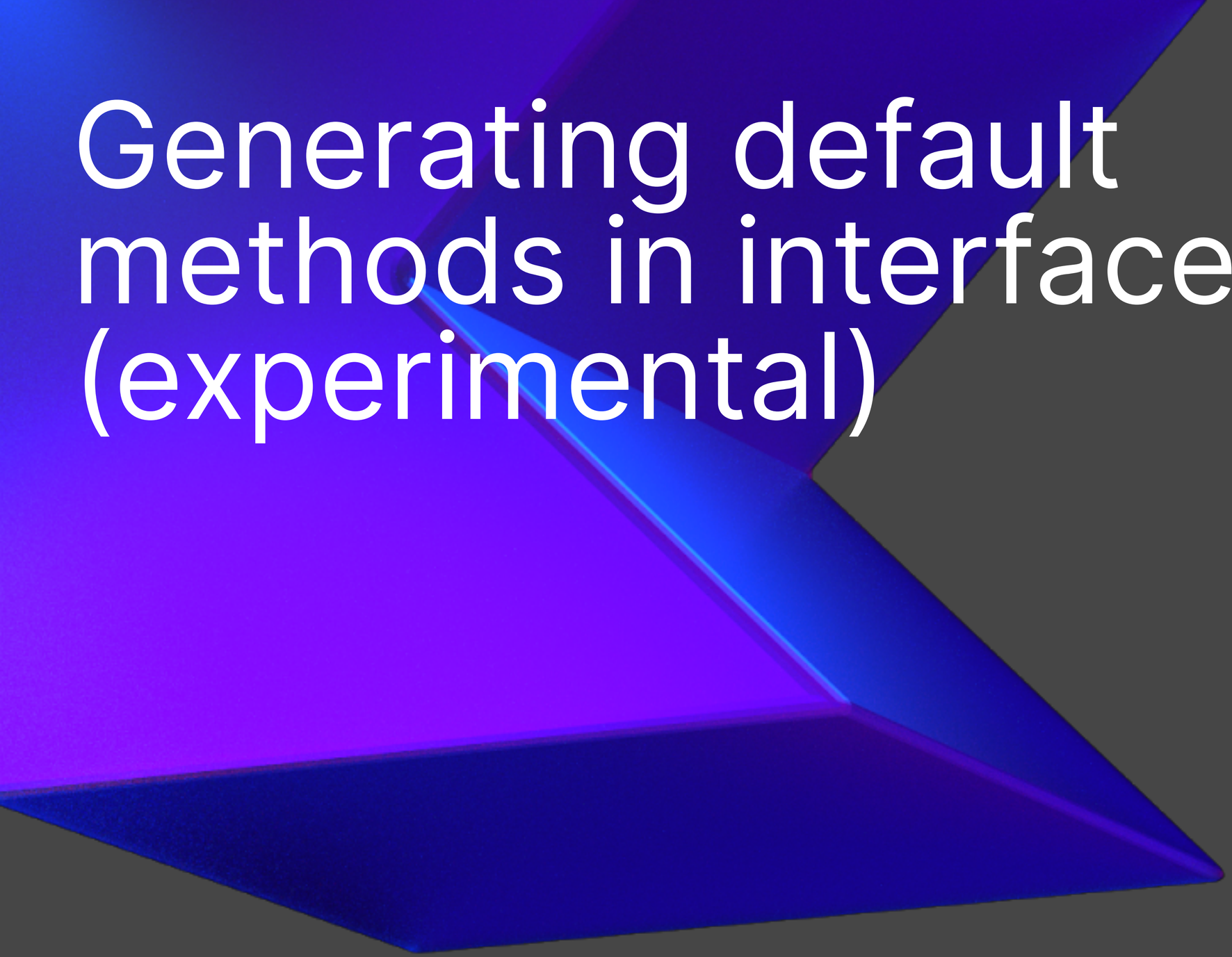
The message remains the same

- Kotlin 1.3:
  IllegalStateException: User.name must not be null
- Kotlin 1.4:
  NullPointerException: User.name must not be null

This makes future optimizations possible

- Optimizations by the Kotlin compiler
- Optimizations by various kinds
  of bytecode processing tools,
  such as the Android R8 optimizer

Generating default methods in interfaces (experimental)



Default methods in interfaces

```
interface Alien {
    fun speak() = "Wubba lubba dub dub"
}

class BirdPerson : Alien
```

Works with the Java 6 target!

Under the hood: DefaultImpls

```
public interface Alien {
    String speak();

    public static final class DefaultImpls {
        public static String speak(Alien obj) {
            return "Wubba lubba dub dub";
        }
    }
}
```

Under the hood: DefaultImpls

```
public final class BirdPerson implements Alien {
    public String speak() {
        return Alien.DefaultImpls.speak(this);
    }
}
```

Default methods in Kotlin 1.2+

`-Xjvm-default=enable`

```
interface Alien {
    @JvmDefault
    fun speak() = "Wubba lubba dub dub"
}
```

No `DefaultImpls` is generated!

Generating default methods in Kotlin 1.2+

- Use one of the special modes for the Java 8 target:
  - Generating only default methods
  - Generating default methods and `DefaultImpls` classes for compatibility
- Annotate with `@JvmDefault` each interface method that has an implementation

Generating default methods in Kotlin 1.4

- Use one of the **new** modes for the Java 8 target:
  - Generating only default methods
  - Generating default methods and `DefaultImpls` classes for compatibility
- ~~Annotate with `@JvmDefault` each interface method that has an implementation~~

Generating default methods in Kotlin 1.4

- Use one of the **new** modes for the Java 8 target:
  - Generating only default methods
  - Generating default methods and `DefaultImpls` classes for compatibility

Default methods in Kotlin 1.4

`-Xjvm-default=all`

```
interface Alien {
    fun speak() = "Wubba lubba dub dub"
}
```

No `DefaultImpls` is generated!

No `@JvmDefault` annotation is needed!

New modes

- Currently experimental
- Will be used by default in the future major versions:
  - `Xjvm-default=all-compatibility`
  - `Xjvm-default=all`

New modes

- Currently experimental
- Will be used by default in the future major versions:
  - `Xjvm-default=all-compatibility`
  - `Xjvm-default=all`

More resources

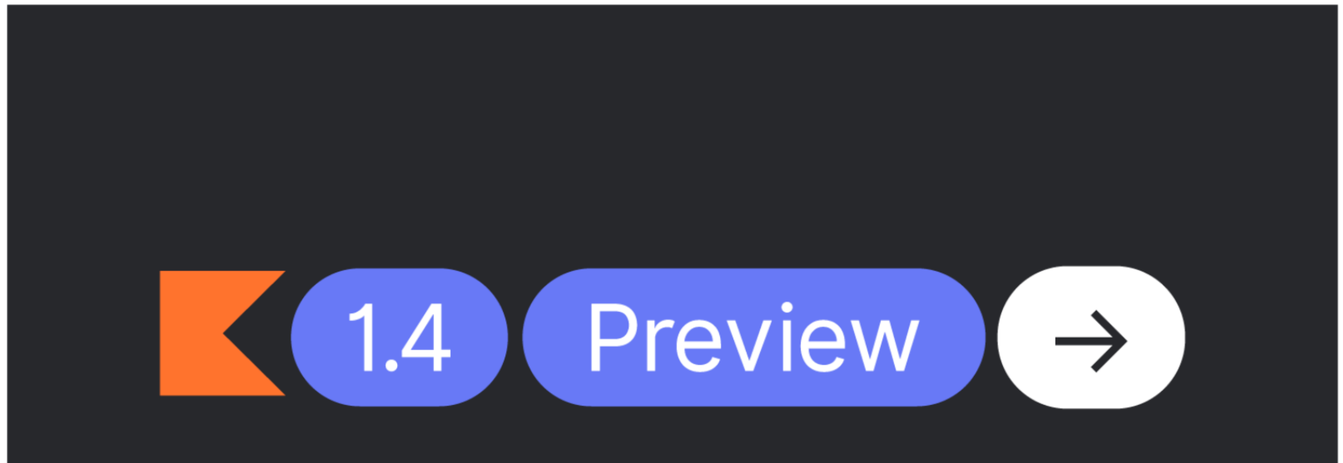
Read more at the Kotlin blog

Kotlin Releases Server Web Mobile Community

Kotlin 1.4-M3: Generating Default Methods in Interfaces

Posted on July 16, 2020 by Svetlana Isakova

In Kotlin 1.4, we're adding new experimental ways for generating default methods in interfaces in the bytecode for the Java 8 target. Later, we're going to be deprecating the `@JvmDefault` annotation in favor of generating all the method bodies in interfaces directly when the code is compiled in a special mode. Read more details of how it currently works and what will change, below.



What’s New in 1.4 documentation page

What's New in Kotlin 1.4.0

Edit Page

In Kotlin 1.4.0, we ship a number of improvements in all of its components, with the focus on quality and performance. Below you will find the list of the most important changes in Kotlin 1.4.0.

Language features and improvements

- SAM conversions for Kotlin interfaces
- Explicit API mode for library authors
- Mixing named and positional arguments
- Trailing comma
- Callable reference improvements
- `break` and `continue` inside `when` included in loops

New tools in the IDE

- New flexible Project Wizard
- Coroutine Debugger

Thanks!
Have a nice Kotlin

@sveta_isakova