

‘Context parameters’ from the language design perspective

Anastasiia Nekrasova

Motivation for Language Design

Our motivation for designing language features is based on specific **use cases** and **unsolved problems**

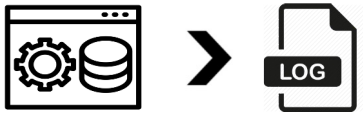


Use-case: loading...

```
class Logger(val name: String, val filePath: Path) {
    fun info(message: String) = ...
}

fun Order.process() {
    // ...
}
```

Here, we want to record which order we are processing



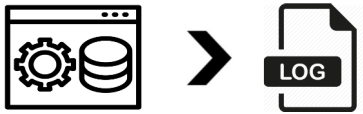
Use-case: loading...

```
class Logger(val name: String, val filePath: Path) {
     fun info(message: String) = ...
}

fun Order.process(logger: Logger) {
     logger.info("Order with id=$id is processed")
}

val orderLogger = Logger("Order logger", loggerFilePath)
order.process(orderLogger)
```

What if there is more than one such function?

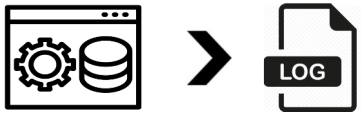


Use-case: loading...

```
class Logger(val name: String, val filePath: Path) {
     fun info(message: String) = ...
}


fun Order.process(logger: Logger) {
     logger.info("Order with id=$id is processed")
}

when (state) {
     State.READY -> order.process(orderLogger)
     State.PAID -> order.activate(orderLogger)
     State.ERROR -> order.cancel(orderLogger)
     State.UNDER_VERIFICATION -> order.wait(orderLogger)
}
```



Use-case: loading...

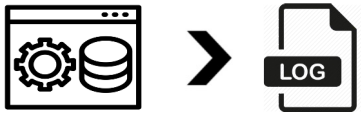
```
class Logger(val name: String, val filePath: Path) {
     fun info(message: String) = ...
}
```

Extension receiver is already present

```
fun Order.process(logger: Logger) {
     logger.info("Order with id=$id is processed")
}
```

Boilerplate code...

```
when (state) {
     State.READY -> order.process(orderLogger)
     State.PAID -> order.activate(timeStamp, wallet, orderLogger)
     State.ERROR -> order.cancel(orderStatus, errorMessage, orderLogger)
     State.UNDER_VERIFICATION -> order.wait(orderStatus, verificationReason, orderLogger)
}
```



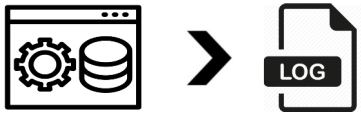
Use-case: loading...

```
val orderLogger = Logger(name="Order logger", filePath =...)
```

But how to set different file paths now?...

'orderLogger' is no longer repeated

```
when (state) {
     State.READY -> order.process()
     State.PAID -> order.activate(timeStamp, wallet)
     State.ERROR -> order.cancel(orderStatus, errorMessage)
     State.UNDER_VERIFICATION -> order.wait(orderStatus, verificationReason)
}
```

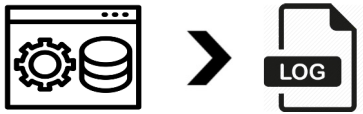


Use-case: Implicit services

```
class Logger(val name: String, val filePath: Path) {
     fun info(message: String) = ...
}

context(logger: Logger)
fun Order.process() {
     logger.info("Order with id=$id is processed")
}
```

Let's use a context parameter!





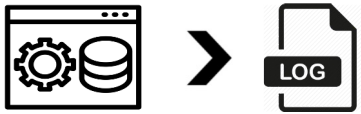
Use-case: Implicit services

```
context(logger: Logger)
fun Order.process() {
     logger.info("Order with id=$id is processed")
}

context(logger: Logger)
fun Order.activate(timeStamp: Timestamp, wallet: Wallet)
{...}

context(logger: Logger)
fun Order.cancel(status: Status, errorMessage: String) {...}

context(logger: Logger)
fun Order.wait(status: Status, reason: Reason) {...}
```

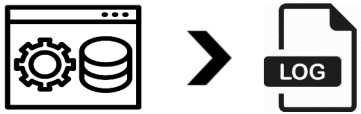


Use-case: Implicit services

```
context(logger: Logger)
fun Order.process() {
     logger.info("Order with id=$id is processed")
}

with(Logger("Order logger", loggerFilePath)) {
     when (state) {
          State.READY -> order.process()
          State.PAID -> order.activate(timeStamp, wallet)
          State.ERROR -> order.cancel(orderStatus, errorMessage)
          State.UNDER_VERIFICATION -> order.wait(orderStatus, verificationReason)
     }
}
```

Here, we include ‘Logger’ into the context

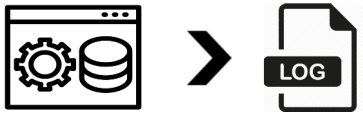


Use-case: Implicit services

```
context(logger: Logger)
fun Order.process() {
     logger.info("Order with id=$id is processed")
}

with(Logger("Order logger", testLoggerFilePath)) {
     runAllTests()
}
```

Now we can call the function 'process' in the tests



Context parameters use-cases

Set of services included implicitly

Pass your services using context parameters



Context parameters use-cases

Set of services included implicitly

???



Use-case: loading...

We need a property that converts types from `Jetpack Compose`: `DpOffset -> Offset`

```
val DpOffset.asPx: Offset get() = Offset(...)
```



Jetpack Compose
time!



Use-case: loading...

We need a property that converts types from `Jetpack Compose`: `DpOffset -> Offset`

properties of `DpOffset`

```
val DpOffset.asPx: Offset get() = Offset(x, y)
```

How to convert *x* and *y* from `Dp` to `Float` type?

`Dp` → device-independent pixels

`Float` → actual device pixels



Jetpack Compose
time!



Use-case: Scopes

We need a property that converts types from `Jetpack Compose`: `DpOffset -> Offset`

It cannot work without **`Density`**

```
val DpOffset.asPx: Offset get() = Offset(x.toPx(), y.toPx())
```

How to include **`Density`**:

- Declare our property in the **`Density`** interface → are you working at the Jetpack Compose team?

Jetpack Compose library

```
interface Density : FontScaling {
    fun Dp.toPx(): Float = ...
    // ...
}
```



`Jetpack Compose time!`



Use-case: Scopes

We need a property that converts types from `Jetpack Compose`: `DpOffset -> Offset`

It's possible to omit the context parameter name

```
context (_: Density) val DpOffset.asPx: Offset get() = Offset(x.toPx(), y.toPx())
```

Our property works in the `Density` scope and has access to `Density` functions



Jetpack Compose
time!



Context parameters use-cases

Set of services included implicitly

Scopes

Declarations work only within a specific scope



Use-case: loading...

Our DSL library

```
class Client(var name: String? = null, var birthday: LocalDate? = null)

fun buildClient(init: Client.() -> Unit): Client {
   var client = Client()
   client.init()
   return client
}
```



User code

```
buildClient {
      name = "Bob"
      birthday = LocalDate.of(2000, 3, 10)
}
```

We need more DSL here!!!

Use-case: loading...

Our DSL library

```
class Client(var name: String? = null, var birthday: LocalDate? = null)

fun buildClient(init: Client.() -> Unit): Client {
   var client = Client()
   client.init()
   return client
}

infix fun Int.March(year: Int) = LocalDate.of(year, Month.MARCH, this)
```



User code

```
buildClient {
      name = "Bob"
      birthday = LocalDate.of(2000, 3, 10)
}
```

We need more DSL here!!!



Use-case: loading...

Our DSL library

```
class Client(var name: String? = null, var birthday: LocalDate? = null)

fun buildClient(init: Client.() -> Unit): Client {
   var client = Client()
   client.init()
   return client
}

infix fun Int.March(year: Int) = LocalDate.of(year, Month.MARCH, this)
```



User code

```
buildClient {
      name = "Bob"
      birthday = 10 March 2000
}

val date = 10 March 2000
```

All our infix functions will be available throughout the client's scope....



Use-case: DSL

Our DSL library

```
object ClientBuilderContext

context(_: ClientBuilderContext)
infix fun Int.March(year: Int) = LocalDate.of(year, Month.MARCH, this)
```



User code

```
buildClient {
      name = "Bob"
      birthday = 10 March 2000
}

val date = 10 March 2000
```

Use-case: DSL

Our DSL library

```
object ClientBuilderContext

context(_: ClientBuilderContext)
infix fun Int.March(year: Int) = LocalDate.of(year, Month.MARCH, this)

fun buildClient(init: context(ClientBuilderContext) Client.() -> Unit): Client =
   with(ClientBuilderContext) {
        var client = Client()
        init(client)
        return client
   }
```



User code

```
buildClient {
      name = "Bob"
      birthday = 10 March 2000
}

val date = 10 March 2000
```

The required context is missing



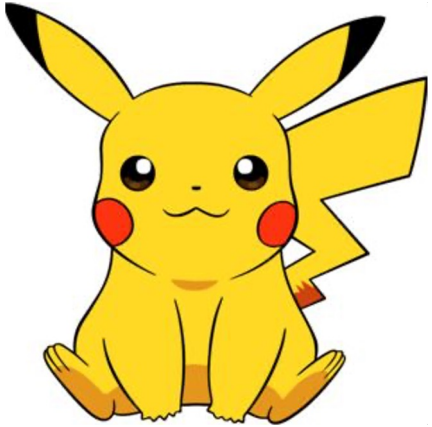
Context parameters use-cases

Set of services included implicitly

Scopes

DSL

Smarter, better, stronger



Context parameters use-cases

Set of services included implicitly

Scopes

DSL

Externally implemented

interfaces or type classes



KEEP-367: https://github.com/Kotlin/KEEP/blob/context-parameters/proposals/context-parameters.md



Context parameters use-cases

Set of services included implicitly

Scopes

DSL

Externally implemented

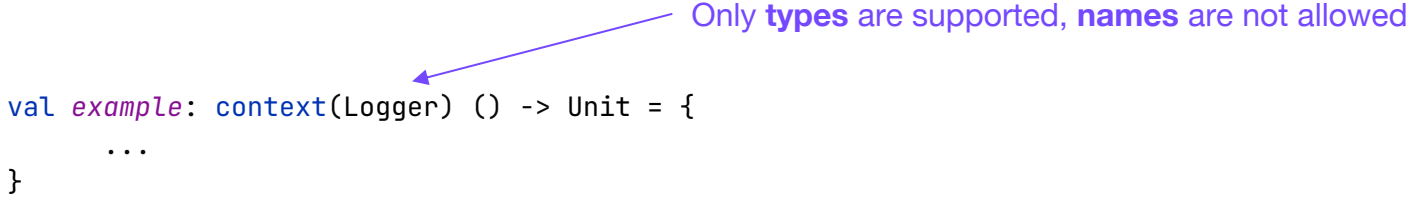
interfaces or type classes



KEEP-367: https://github.com/Kotlin/KEEP/blob/context-parameters/proposals/context-parameters.md



Under the Hood: Function types




```
val example: context(Logger) () -> Unit = {
      ...
}
```

Under the Hood: Function types

```
class Logger {
    fun log(string: String) {}
}
```

Only **types** are supported, **names** are not allowed

```
val example: context(Logger) () -> Unit = {
        log("Hello")
        implicit<Logger>().log("Hello again")
}
```

The context parameter behaves as if declared with ‘_’ as its name



Under the Hood: Function types

```
class Logger {
   fun log(string: String) {}
}
```

Only **types** are supported, **names** are not allowed

```
val example: context(Logger) () -> Unit = {
      log("Hello")
      implicit<Logger>().log("Hello again")
}
```

You can use the function 'implicit' to access the context parameter

The context parameter behaves as if declared with '_' as its name



Limitations: Classes

No context parameters in **constructors** and in **classes**

```
context(scope: AnalysisScope) class TypeScope {
    ...
}
```

This is no longer allowed

```
class TypeScope(val scope: AnalysisScope) {
    companion object {
        context(scope: AnalysisScope)
        operator fun invoke() = TypeScope(scope)
    }
}
```

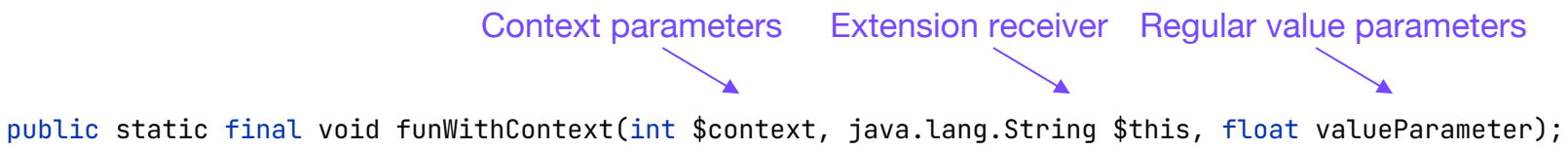
Use the operator function ‘invoke’ instead



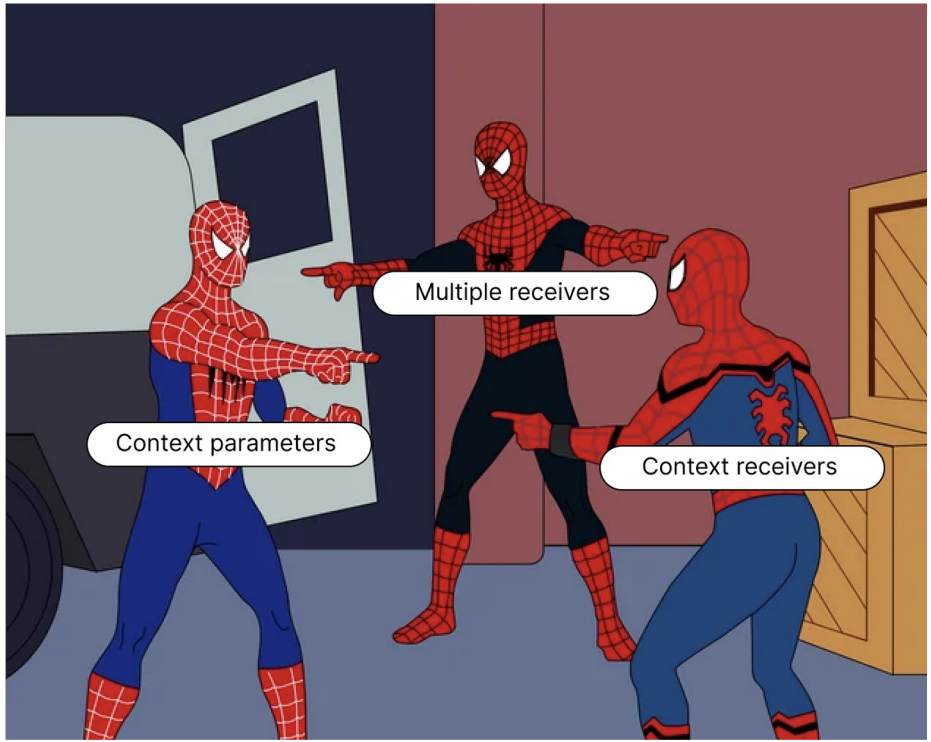
Under the Hood: JVM

A function with context parameters can be seen as a function with extended parameters:

```
context(contextParameter: Int)
fun String.funWithContext(valueParameter: Float) {}
```



‘Context parameters’ or ‘Context receivers’?



- ‘Context parameters’ - KEEP-367
- ‘Context receivers’ - KEEP-259
- ‘Multiple receivers’ - KT-10468

Why did we drop context receivers?

```
interface File {
     fun write()
}

interface Database {
     fun write()
}
```

Context parameters

```
context(file: File, db: Database)
fun save() {
          file.write()
}
```

Context receivers

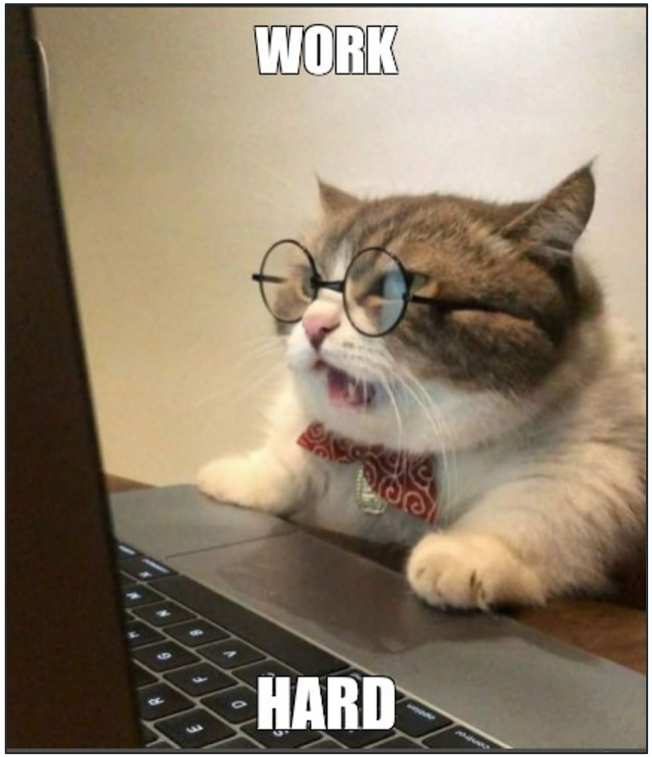
Scope pollution :(

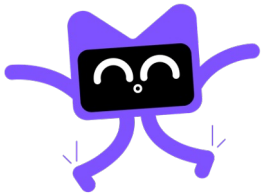
```
context(File, Database)
fun save() {
          write()
}
```



Future work plans

The prototype of ‘Context parameters’ will be available after Kotlin 2.2





anastasia.nekrasova@jetbrains.com

Thank you, and don’t forget to vote

