

8 2023

SPRING  IC^{TH ANNIVERSARY}

#springio23

Kotlin for Fun for Spring Fans



Shinya Yanagihara

@yanashin18618



BARCELONA MAY 18-19 / WWW.SPRINGIO.NET



Kotlin for Fun for Spring Fans

for

柳原 伸弥 / Shinya Yanagihara

Developer Advocate
VMware Japan

May 19, 2023

About me



Shinya Yanagihara

- Company: VMware Japan
- Role: Developer Advocate
- Career:



- SNS:
 - Twitter icon: @yanashin18681
 - Mastodon icon: @yanashin@mastodon.social
 - Cloud icon: “Somebody, fly me to the sky”

- My motto:

“Work for the Developer Experience”

Qustion



Do you love
Spring ?

Qustion



Do you like
Kotlin ?

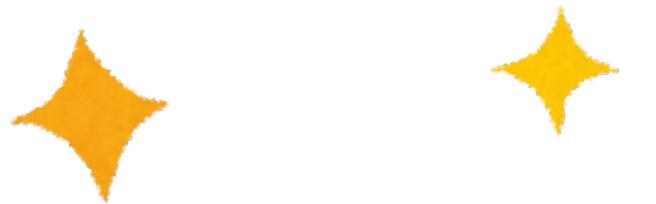
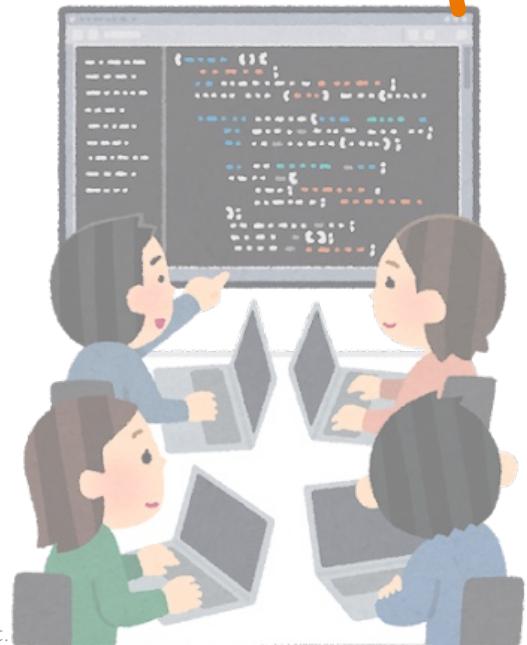
My Answer

I love
Kotlin and Spring.



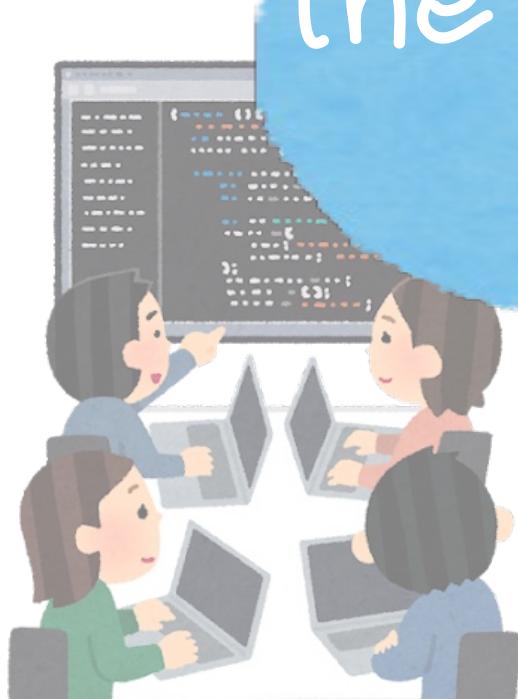
Why I love Kotlin

It's all about
the Developer Experience.

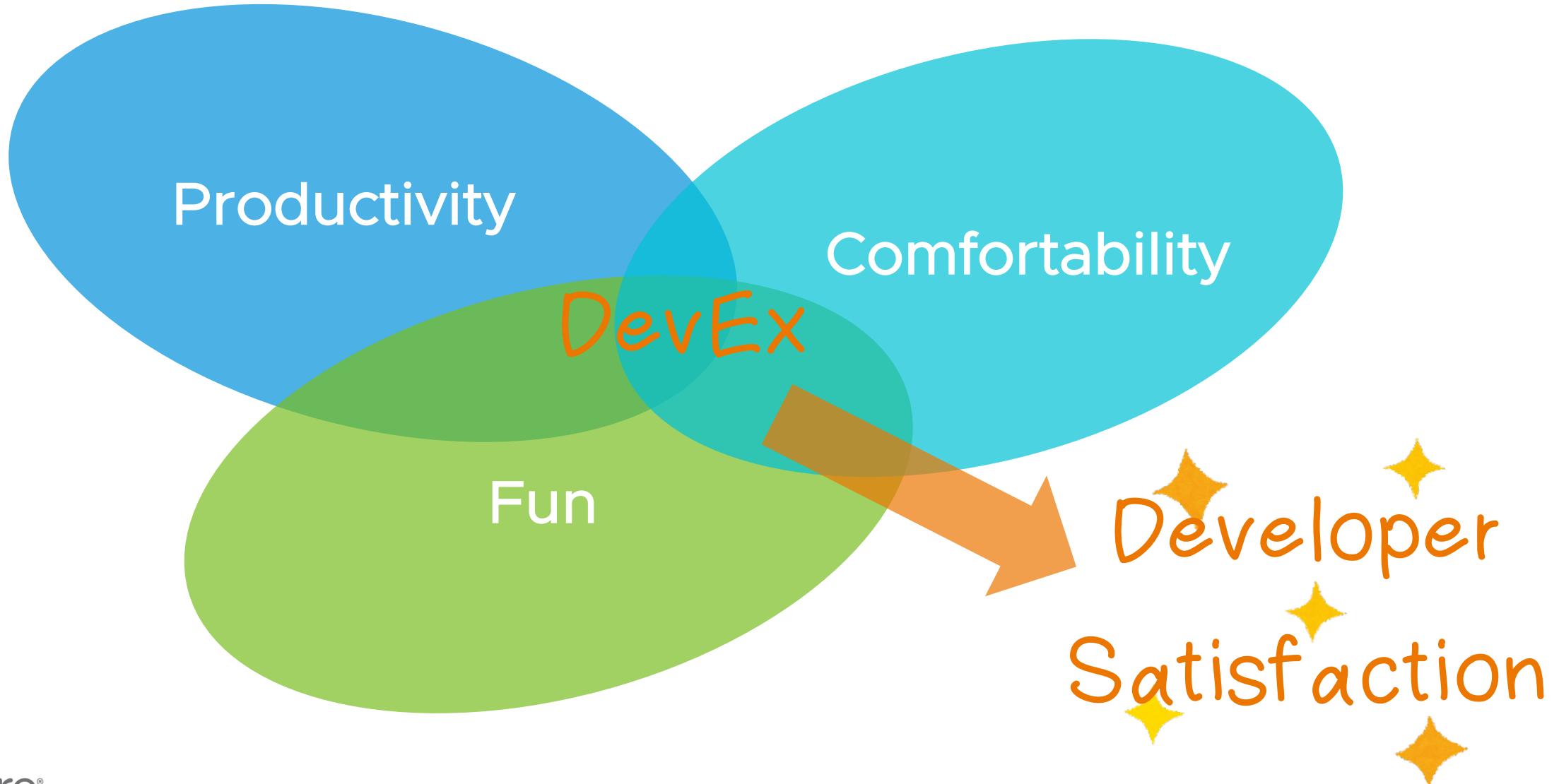


What is the Developer Experience

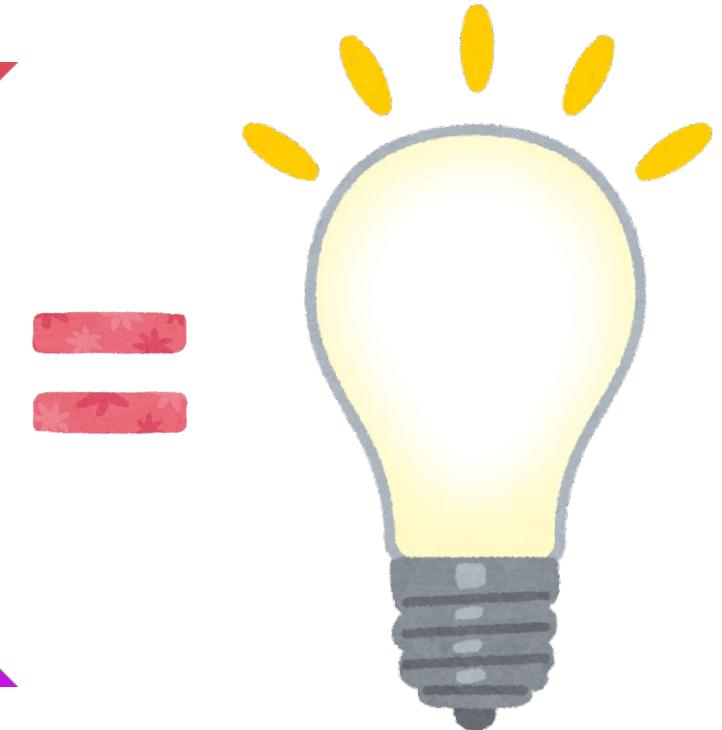
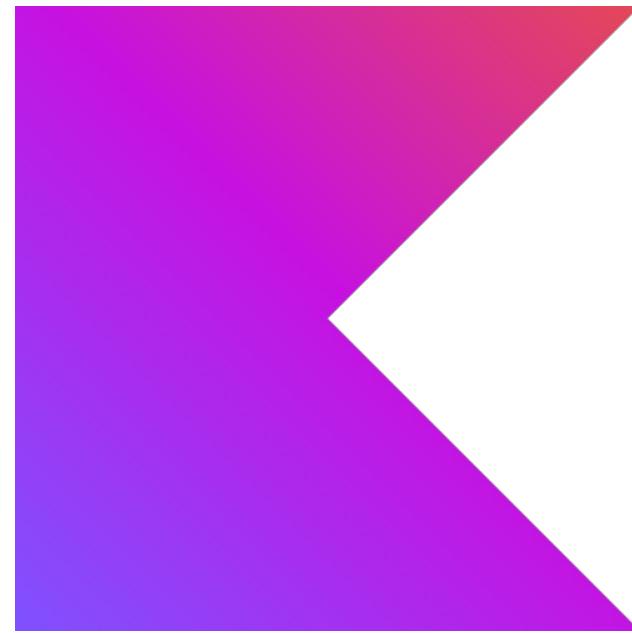
What is
the Developer Experience
for you?



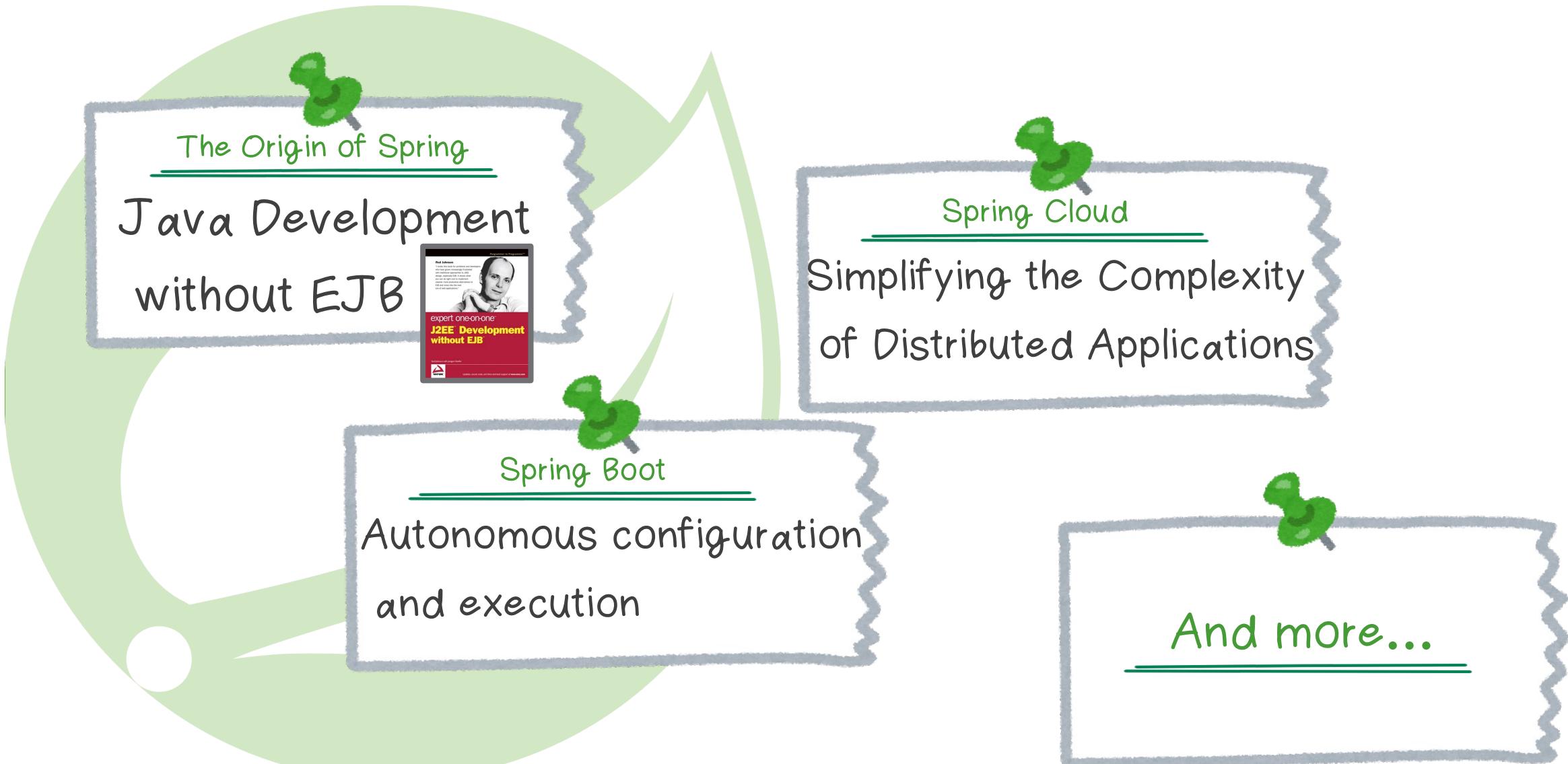
Developer Experience



Developer Satisfaction with Spring and Kotlin



Developer Experience for Spring

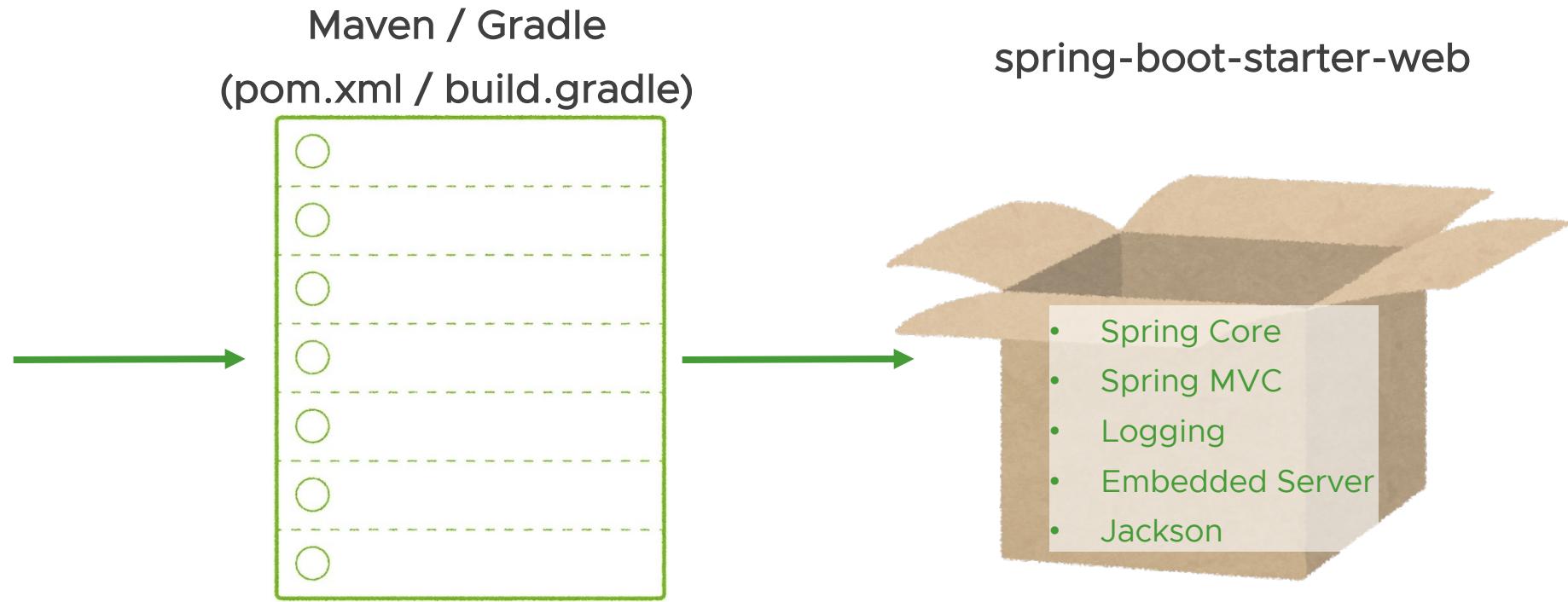


Developer Experience for Spring



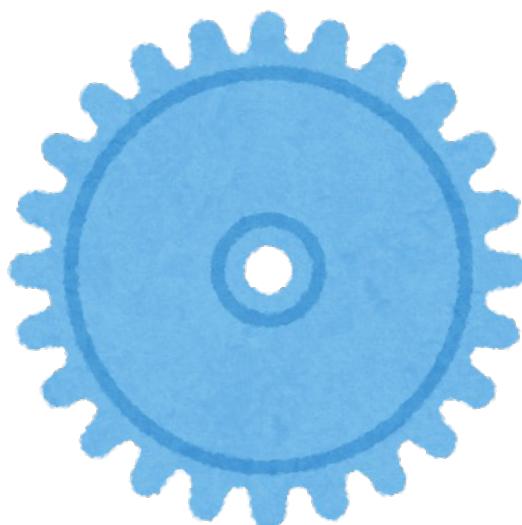
Lots of value
in Spring

Spring Boot Starter



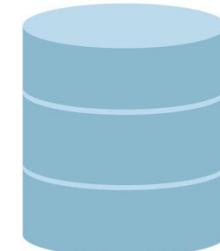
Automatic retrieval of a set of libraries required for development

Auto-Configuration



Automatically imported Configuration Classes

- DataSourceAutoConfiguration
 - JdbcTemplateAutoConfiguration
 - DispatcherServletAutoConfiguration
 - WebClientAutoConfiguration
 - ReactorAutoConfiguration
 - WebMvcAutoConfiguration
 - SecurityAutoConfiguration
 - ...



3rd party library

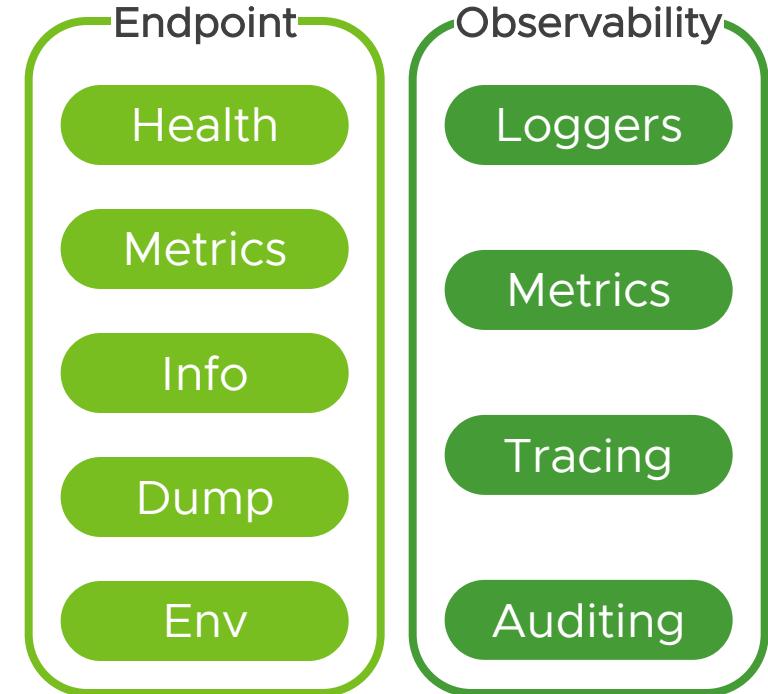
1

Automatically configure the application based on the dependencies that are present on the classpath

Spring Boot Actuator

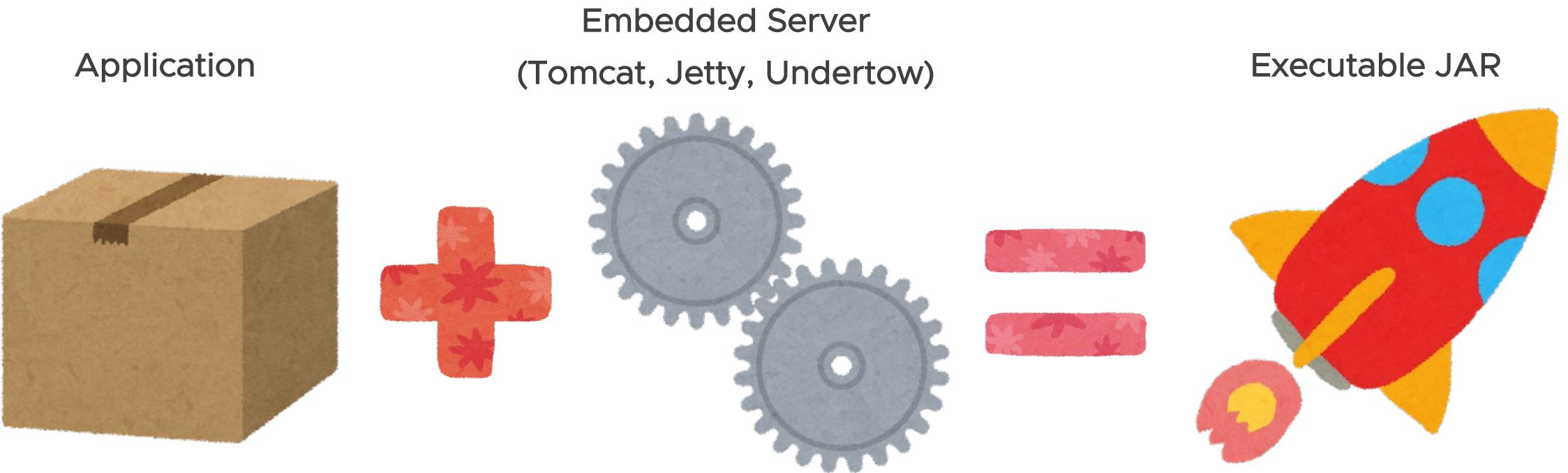


Expose operational information
about the running application



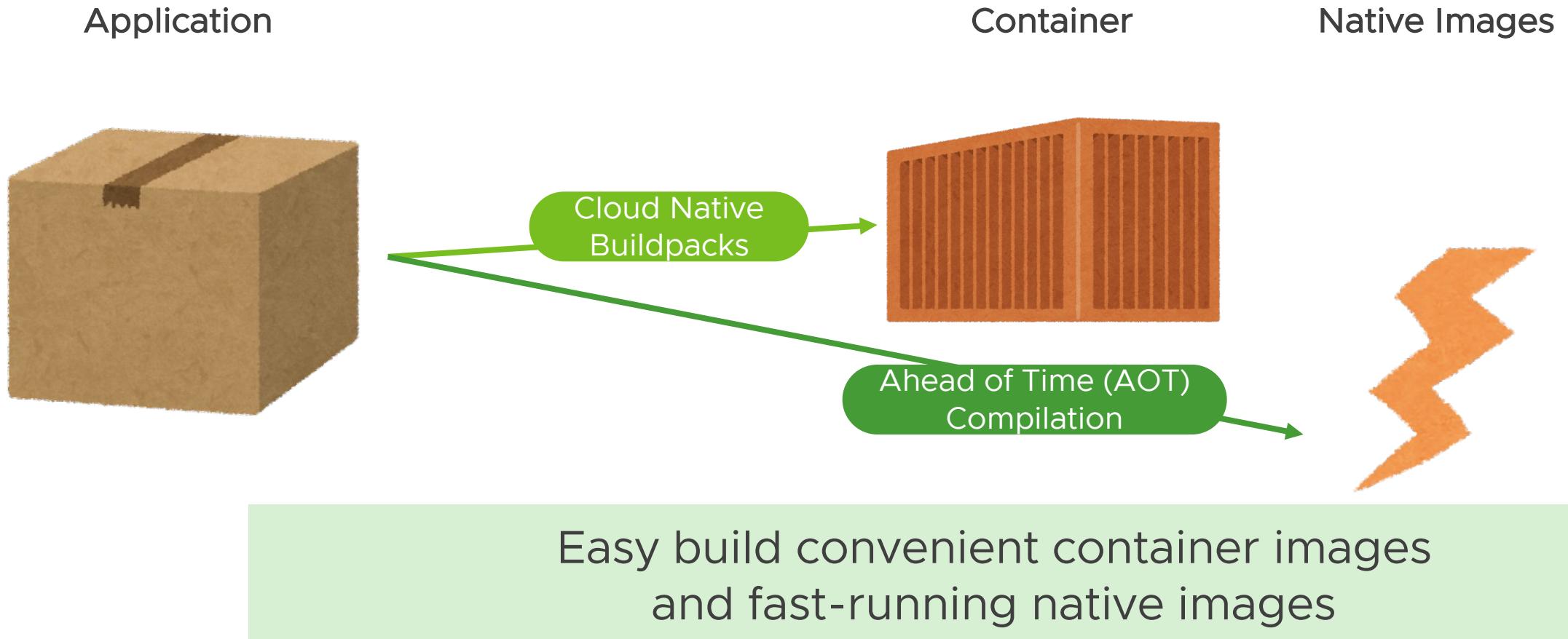
Bring production-ready features to your application
without implementing yourself.

Executable JAR & Embedded Server



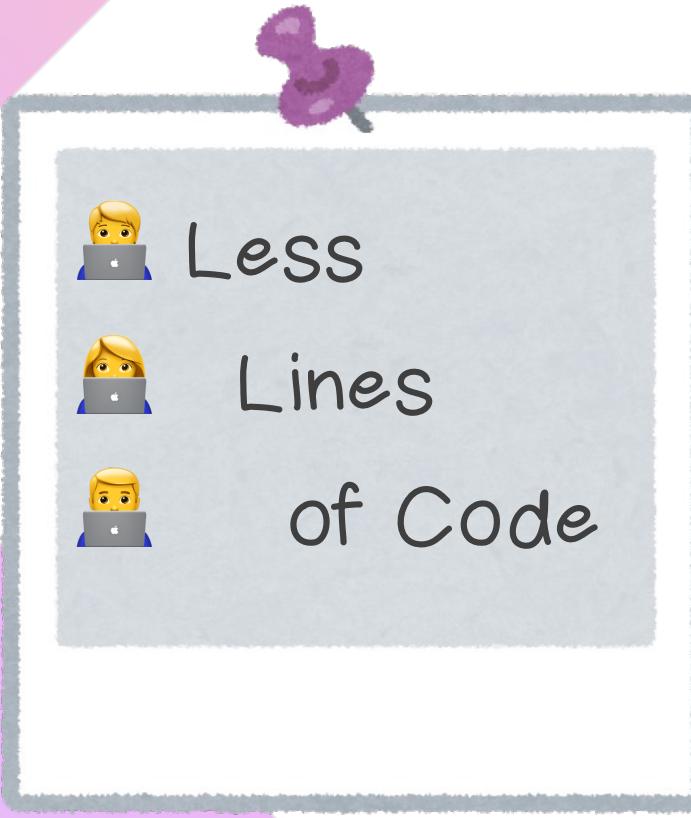
Autonomously running application binaries

Containers & Native Images



Developer Experience for Kotlin

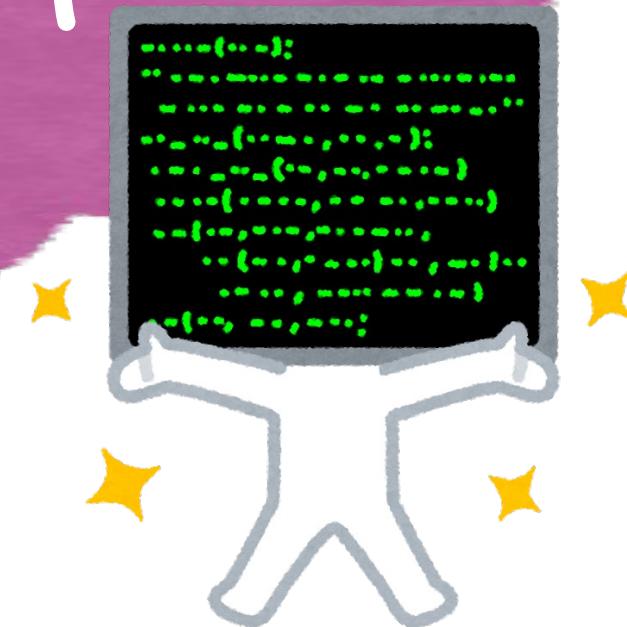
- ✓ Simpler
- ✓ Easier
- ✓ Conciser



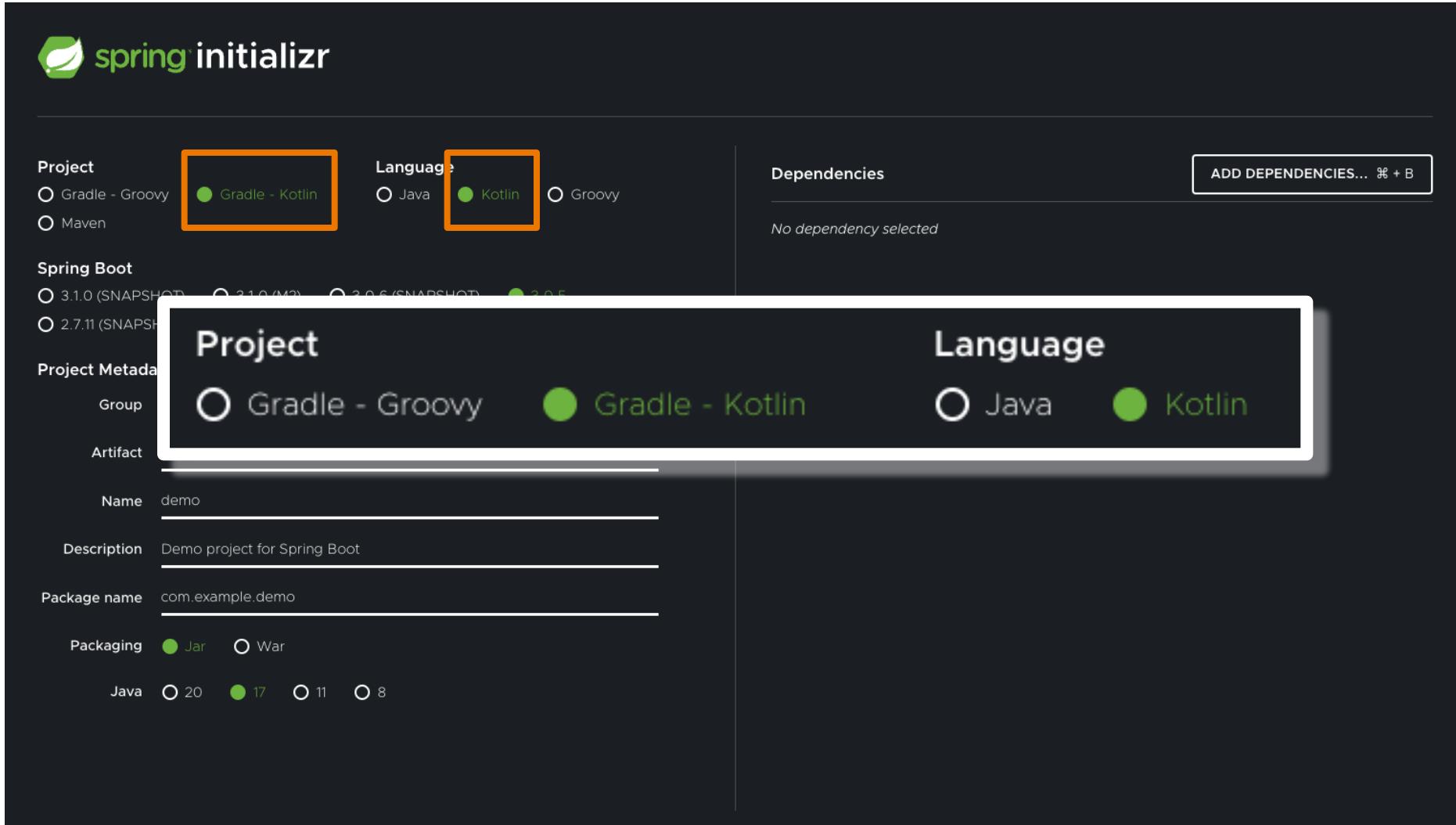
Java
Compatibility
Better Java

That's Today's topic.

Let's look at the differences between Kotlin and Java and Kotlin's DevEx while building an application.



Spring Initializr: “Everything starts here.”



build.gradle / build.gradle.kts

```
1 import org.jetbrains.kotlin.gradle.tasks.KotlinCompile  
2  
3 plugins {  
4     id 'org.springframework.boot' version '3.0.5'  
5     id 'io.spring.dependency-management' version '1.1.0'  
6     id 'org.jetbrains.kotlin.jvm' version '1.7.22'  
7     id 'org.jetbrains.kotlin.plugin.spring' version '1.7.22'  
8 }  
9  
10 group = 'com.example'  
11 version = '0.0.1-SNAPSHOT'  
12 sourceCompatibility = '17'  
13  
14 repositories {  
15     mavenCentral()  
16 }  
17  
18 dependencies {  
19     implementation 'org.springframework.boot:spring-boot-starter'  
20     implementation 'org.jetbrains.kotlin:kotlin-reflect'  
21     testImplementation 'org.springframework.boot:spring-boot-starter-test'  
22 }  
23  
24 tasks.withType(KotlinCompile) {  
25     kotlinOptions {  
26         freeCompilerArgs = ['-Xjsr305=strict']  
27         jvmTarget = '17'  
28     }  
29 }  
30  
31 tasks.named('test') {  
32     useJUnitPlatform()  
33 }  
34
```



©2023 VMware, Inc.

build.gradle

```
1 import org.jetbrains.kotlin.gradle.tasks.KotlinCompile  
2  
3 plugins {  
4     id("org.springframework.boot") version "3.0.5"  
5     id("io.spring.dependency-management") version "1.1.0"  
6     kotlin("jvm") version "1.7.22"  
7     kotlin("plugin.spring") version "1.7.22"  
8 }  
9  
10 group = "com.example"  
11 version = "0.0.1-SNAPSHOT"  
12 java.sourceCompatibility = JavaVersion.VERSION_17  
13  
14 repositories {  
15     mavenCentral()  
16 }  
17  
18 dependencies {  
19     implementation("org.springframework.boot:spring-boot-starter")  
20     implementation("org.jetbrains.kotlin:kotlin-reflect")  
21     testImplementation("org.springframework.boot:spring-boot-starter-test")  
22 }  
23  
24 tasks.withType<KotlinCompile> {  
25     kotlinOptions {  
26         freeCompilerArgs = listOf("-Xjsr305=strict")  
27         jvmTarget = "17"  
28     }  
29 }  
30  
31 tasks.withType<Test> {  
32     useJUnitPlatform()  
33 }  
34
```

build.gradle.kts

It enables you
to use Kotlin

Kotlin JVM plugin

Kotlin - Spring compiler plugin

It helps
Spring implementations
with Kotlin

Kotlin Basics: Classes and Members as “final” by default

non-inheritable

```
class Parent {  
    fun name() = "Base Class"  
}
```



```
class Child: Parent() {  
    override fun name() =  
        "Derived Class"  
}
```

inheritable

```
open class Parent {  
    open fun name() =  
        "Base Class"  
}
```

```
class Child: Parent() {  
    override fun name() =  
        "Derived Class"  
}
```



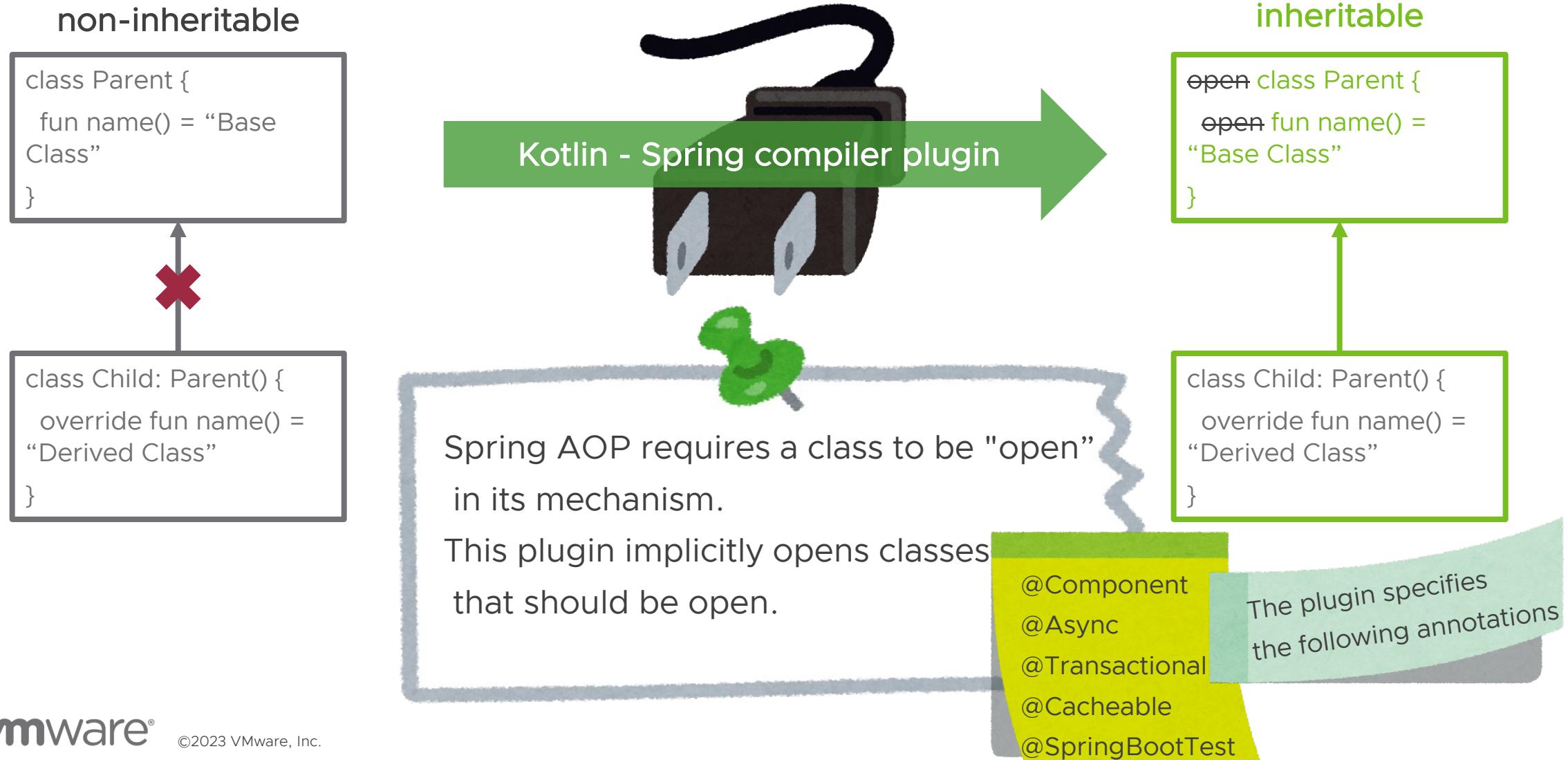
Due to concerns about the "fragile base class" problem, Kotlin classes and their functions are **final** by default.

To allow a class to be extended, it must be marked **open**.

To allow class functions and fields to be overridden, they must also be marked **open**.

“final” by default

Kotlin - Spring compiler plugin



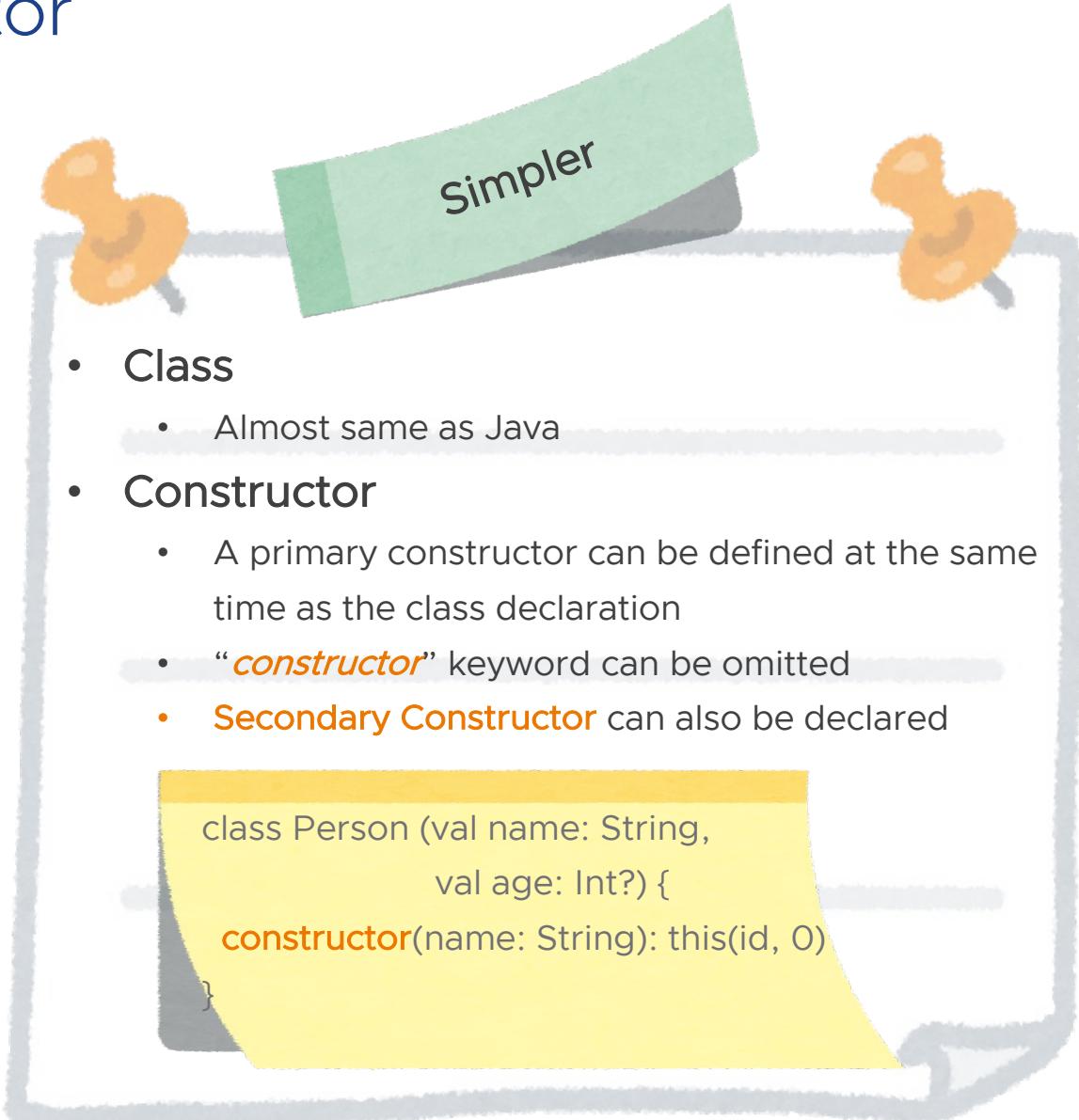
Kotlin Basics: Class and Constructor

Java

```
class Person {  
    final String name;  
    final Integer age;  
  
    public Person(String name, Integer age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

Kotlin

```
class Person constructor(  
    val name: String,  
    val age: Int? = null  
) {...}  
  
class Person (val name: String, val age: Int?) {...}
```



First Spring source code by Kotlin

Java

```
1 package com.example.demo;  
2  
3 import org.springframework.boot.SpringApplication;  
4 import org.springframework.boot.autoconfigure.SpringBootApplication;  
5  
6 @SpringBootApplication  
7 public class DemoApplication {  
8  
9     public static void main(String[] args) {  
10         SpringApplication.run(DemoApplication.class, args);  
11     }  
12 }  
13 }
```

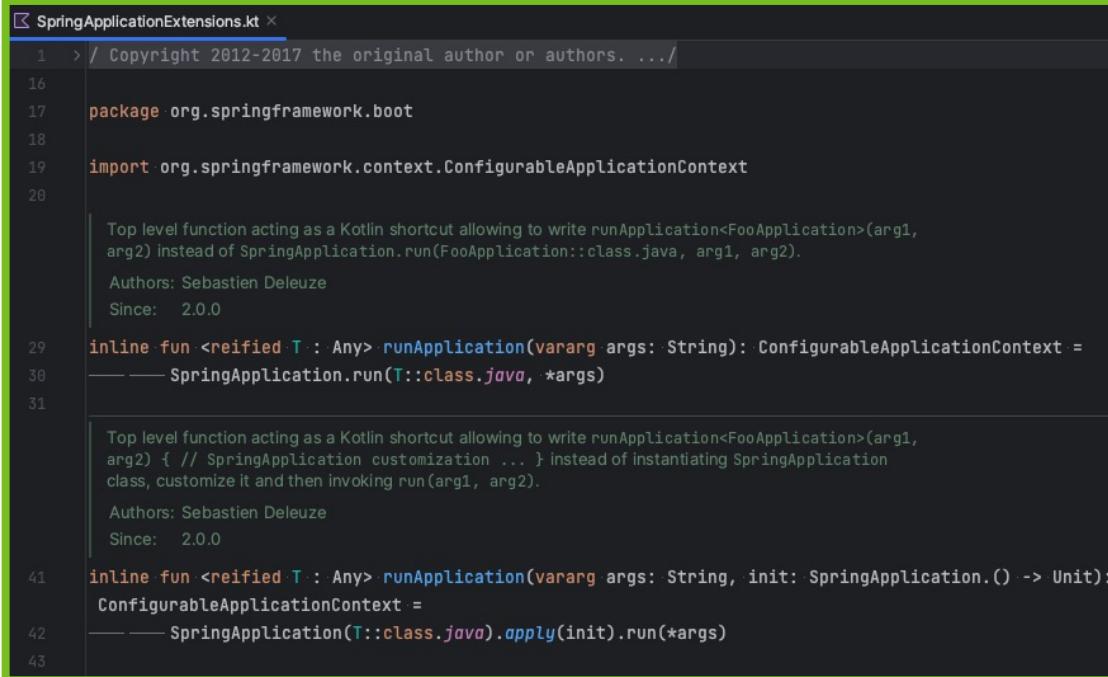
Kotlin

```
1 package com.example.demo  
2  
3 import org.springframework.boot.autoconfigure.SpringBootApplication  
4 import org.springframework.boot.runApplication  
5  
6 @SpringBootApplication  
7 class DemoApplication  
8  
9 fun main(args: Array<String>) {  
10     runApplication<DemoApplication>(*args)  
11 }  
12 }
```

Top-level function

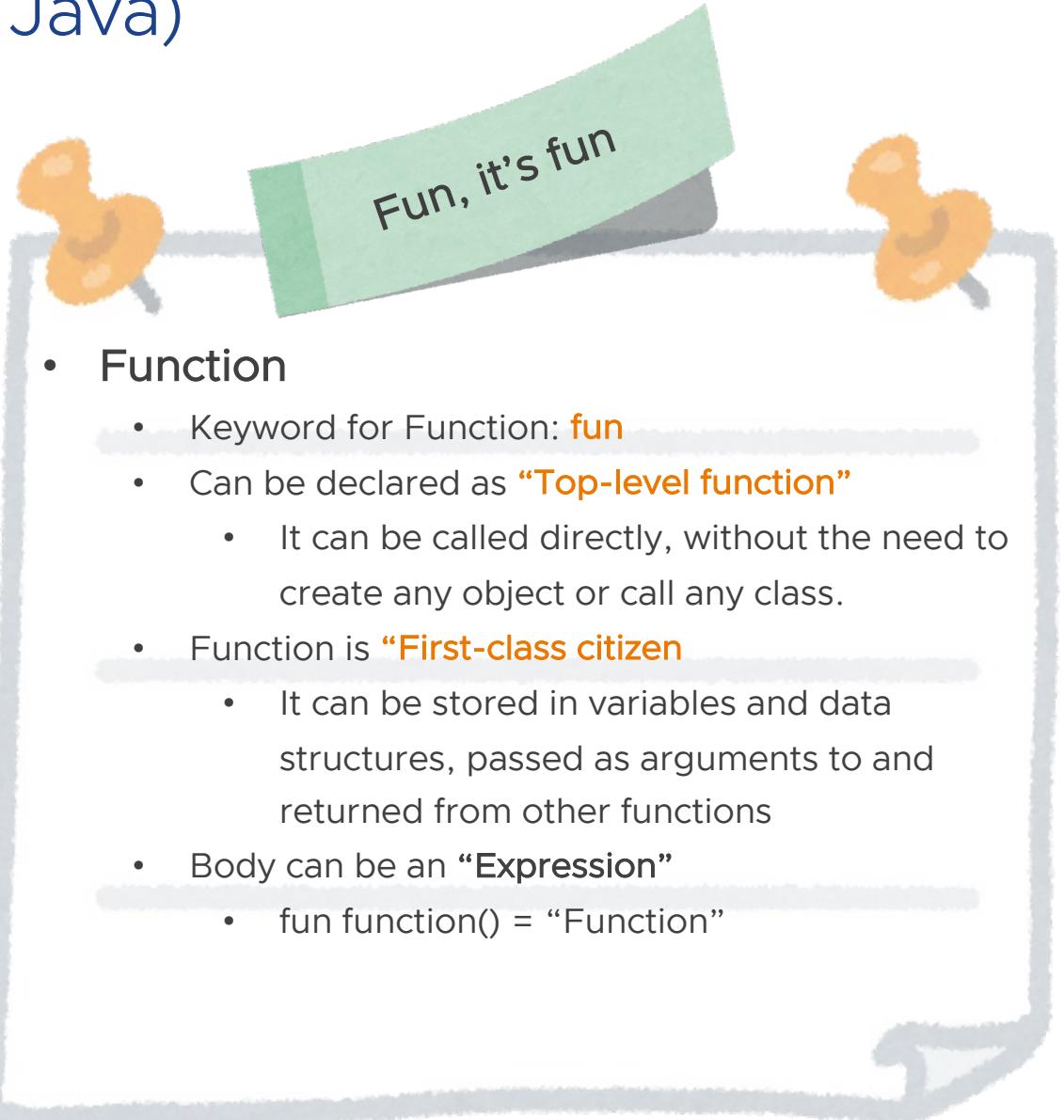
Function is first-class citizen

Kotlin Basics: Function (Method in Java)



```
1 > / Copyright 2012-2017 the original author or authors. ...
16
17 package org.springframework.boot
18
19 import org.springframework.context.ConfigurableApplicationContext
20
21
22 Top level function acting as a Kotlin shortcut allowing to write runApplication<FooApplication>(arg1, arg2) instead of SpringApplication.run(FooApplication::class.java, arg1, arg2).
23 Authors: Sébastien Deléuze
24 Since: 2.0.0
25
26 inline fun <reified T : Any> runApplication(vararg args: String): ConfigurableApplicationContext =
27     SpringApplication.run(T::class.java, *args)
28
29
30 Top level function acting as a Kotlin shortcut allowing to write runApplication<FooApplication>(arg1, arg2) { // SpringApplication customization ... } instead of instantiating SpringApplication class, customize it and then invoking run(arg1, arg2).
31 Authors: Sébastien Deléuze
32 Since: 2.0.0
33
34 inline fun <reified T : Any> runApplication(vararg args: String, init: SpringApplication.() -> Unit): ConfigurableApplicationContext =
35     SpringApplication(T::class.java).apply(init).run(*args)
36
37
```

‘runApplication’ Function
in “SpringApplicationExtensions.kt”



What is First-class function?

First-class function

These operations include:

- ✓ Can be assigned to variables and constants
- ✓ Does not have to have a name
- ✓ Can be used as function parameters
- ✓ Can be used as a return value of a function

```
fun add(a: Int, b: Int) = a + b  
fun subtract(a: Int, b: Int) = a - b  
val calc = mutableListOf(::add, ::subtract)
```

```
println(calc[0](1, 2))  
// 1 + 2 = 3  
println(calc[1](1, 2))  
// 1 - 2 = -1
```

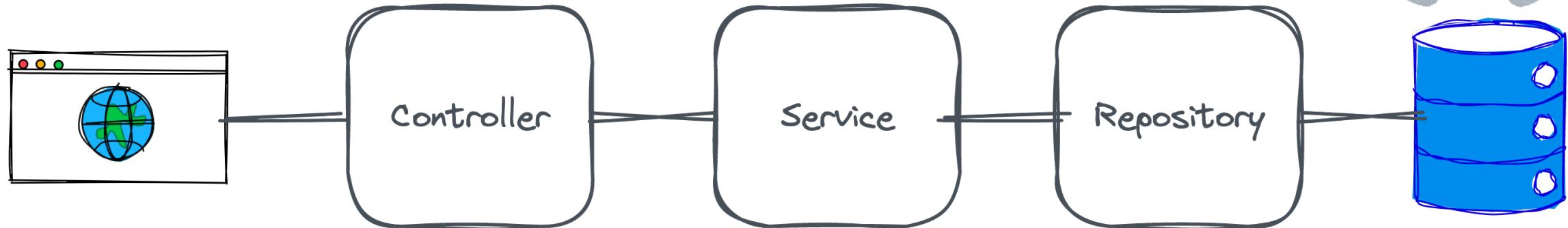
fun add(a: Int, b: Int) = a + b
Normal function definition

val calc: (Int, Int) -> Int = ::add

- ::add : Function Object
- (Int, Int) : Parameter Type
- -> Int : Return Type

Let's code sample app by Kotlin

Let's look at the fun of Kotlin while building the following application.



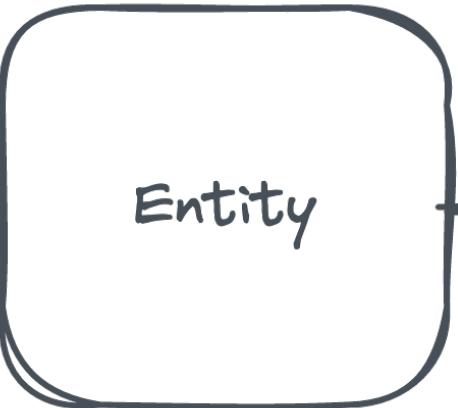
1. Controller to control requests

2. Services to control processing

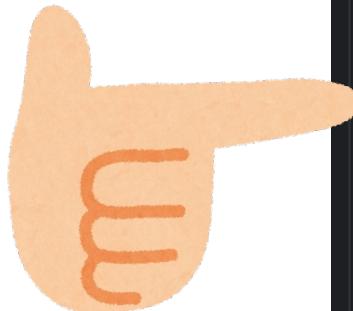
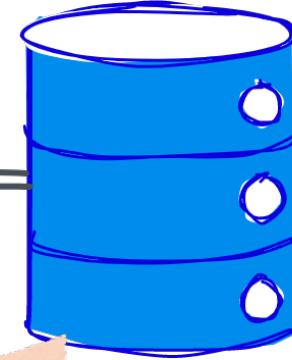
3. Repository for persistent data (using JPA because I have something to mention)



Entity for Data model



EMPNO	ENAME	SAL
7369	SMITH	800
7499	ALLEN	1600
7521	WARD	1250
7566	JONES	2975
7654	MARTIN	1250
7698	BLAKE	2850
7782	CLARK	2450
7788	SCOTT	3000
7839	KING	5000
7844	TURNER	1500
7876	ADAMS	1100
7900	JAMES	950
7902	FORD	3000
7934	MILLER	1300



```
@Entity  
class Book(var name: String?, var author: String?) {  
    @Id  
    @GeneratedValue  
    var id: Long? = null  
  
    constructor() : this(name = "", author = "")  
}
```

Java's **Record** cannot be used in JPA as well as in Kotlin.

Kotlin has Syntax like Java's Record.

Data Class

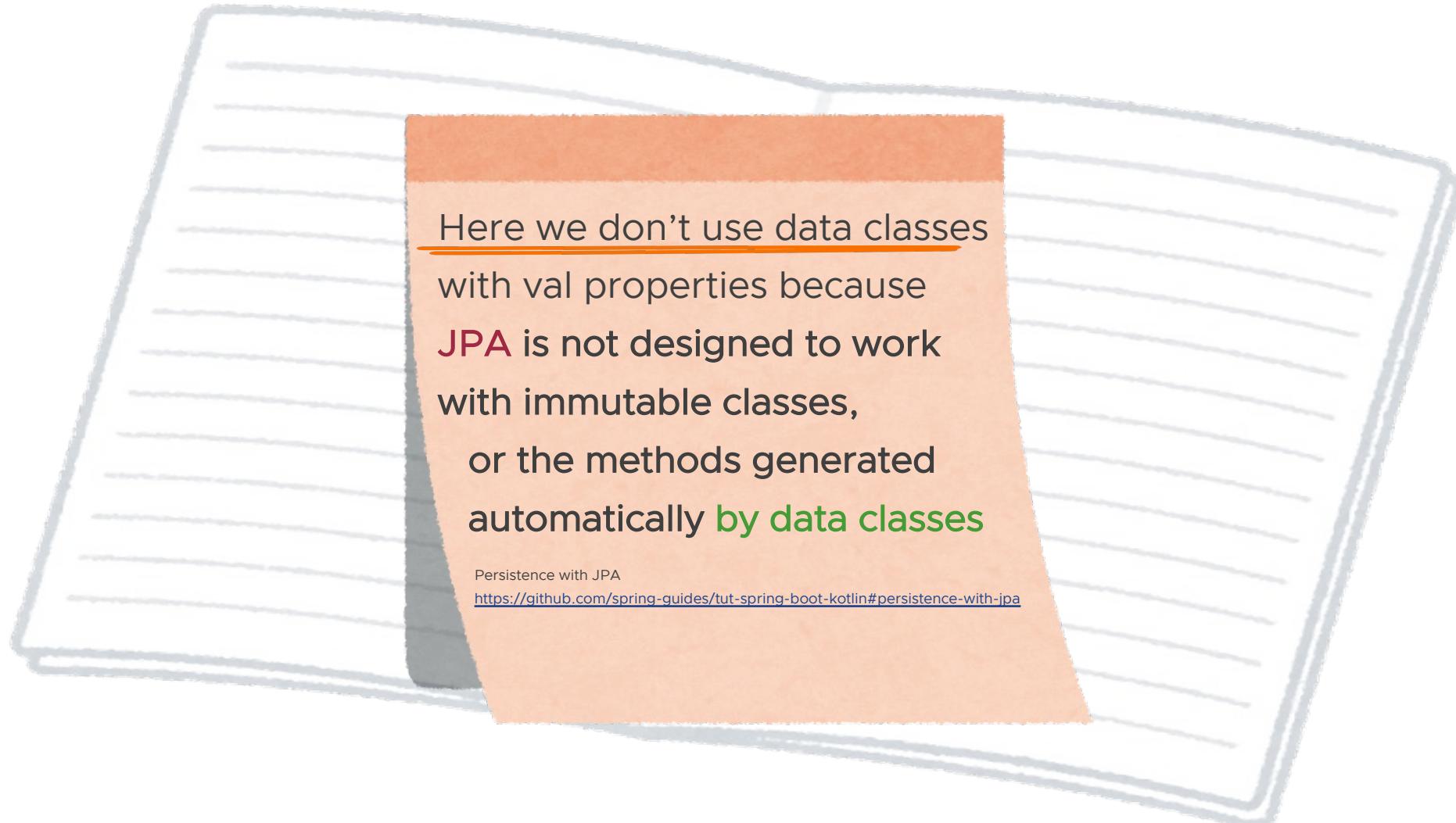
Why not use Java Record for Entity?

Records Can't Be Entities

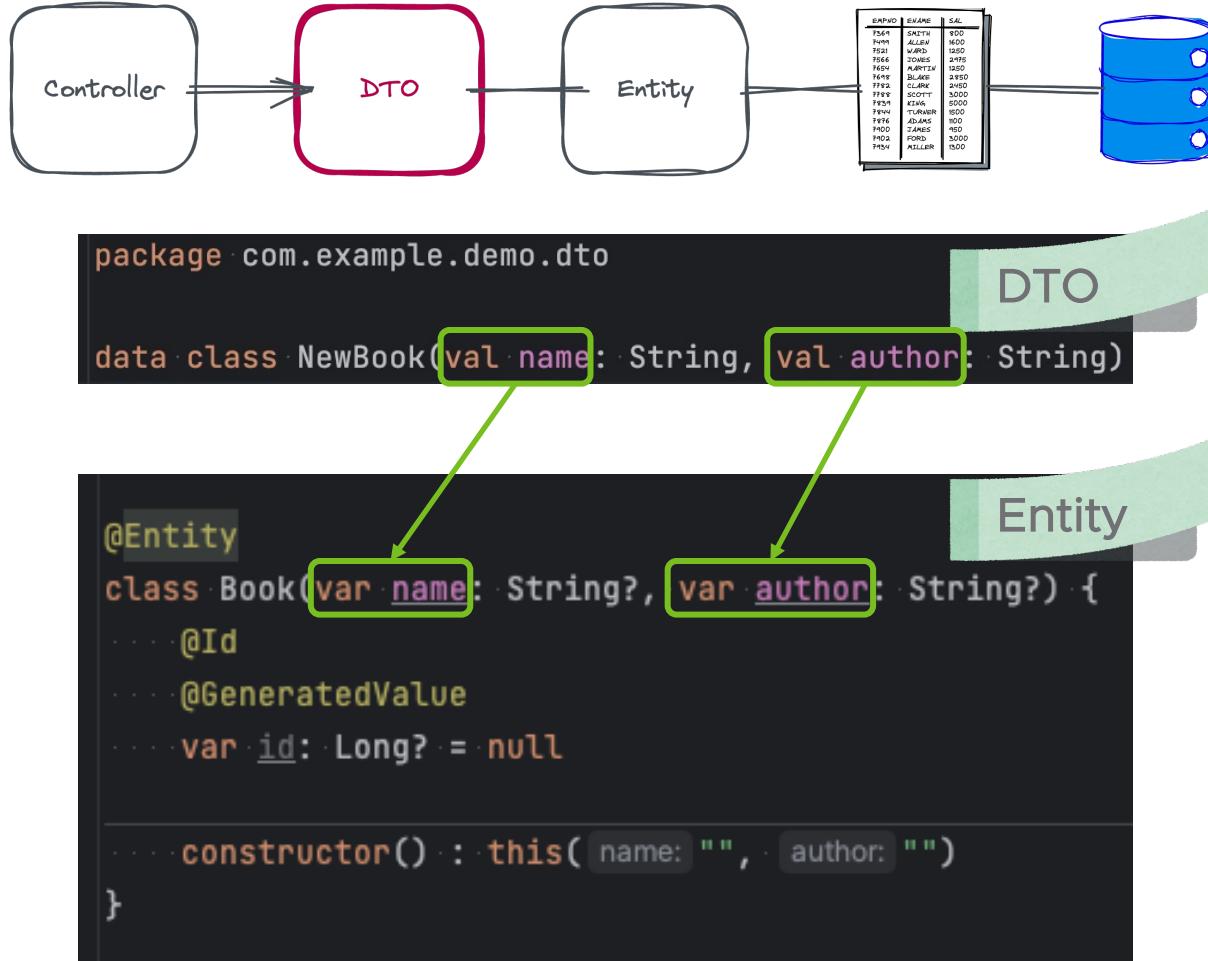
Entities are handled by the JPA provider. JPA providers are responsible for creating the database tables, mapping the entities to the tables, and persisting the entities to the database. In popular JPA providers like Hibernate, entities are created and managed using proxies.

Proxies are classes that are generated at runtime and extend the entity class. **These proxies rely on the entity class to have a no-args constructor and setters.** Since records don't have these, they can't be used as entities.

Why not use Kotlin Data Class for Entity?



Kotlin Basics: Data Class (like Record in Java)

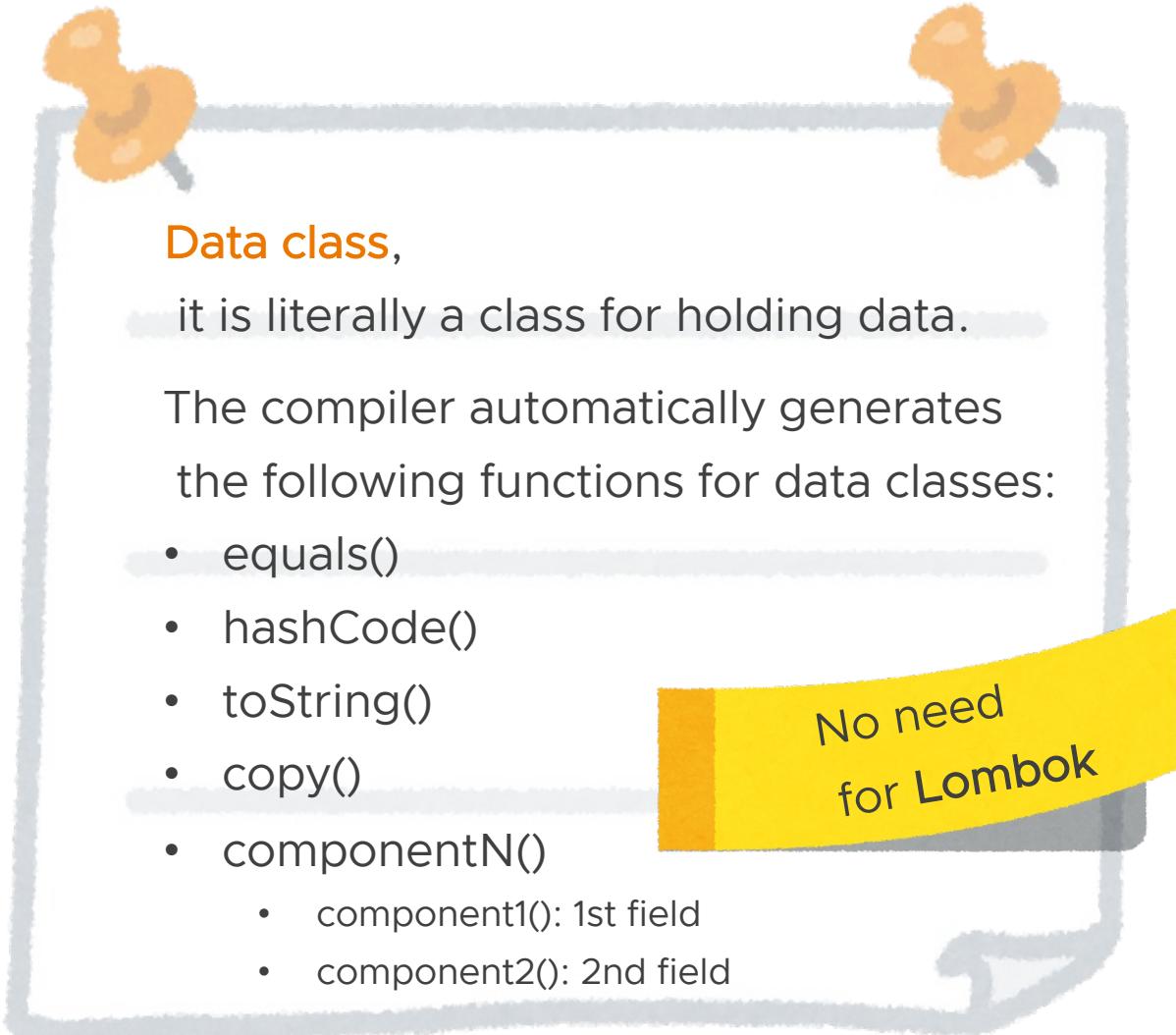


```
package com.example.demo.dto

data class NewBook(val name: String, val author: String)

@Entity
class Book(var name: String?, var author: String?) {
    @Id
    @GeneratedValue
    var id: Long? = null

    constructor() : this(name = "", author = "")
}
```



Kotlin Basics: Read-Only / Mutable Variables

"var" keyword is different from variable declarations for Java type inference



```
package com.example.demo.dto  
  
data class NewBook(val name: String, val author: String)
```

val: Read-Only

```
@Entity  
class Book(var name: String?, var author: String?) {  
    @Id  
    @GeneratedValue  
    var id: Long? = null  
  
    constructor() : this(name = "", author = "")  
}
```

var: Mutable

val: Read-Only

Read-only means that once a variable is initialized, it cannot be reallocated.

"final" keyword in Java

var: Mutable

Mutable means that a variable can be reassigned to a different value after it is initially assigned.

"val" is preferred for variable declarations.

Kotlin Basics: Null Safety and Nullability

Kotlin's type system is aimed at eliminating the danger of null references, also known as "The Billion Dollar Mistake".



```
package com.example.demo.dto

data class NewBook(val name: String, val author: String)
```

```
@Entity
class Book(var name: String?, var author: String?) {
    @Id
    @GeneratedValue
    var id: Long? = null

    constructor() : this(name = "", author = "")
}
```

Java and Kotlin, they are "type safe" because type checking is performed at compile time.

Null Safety

Kotlin

Kotlin's type system

basically **does not allow null references**.

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok
```

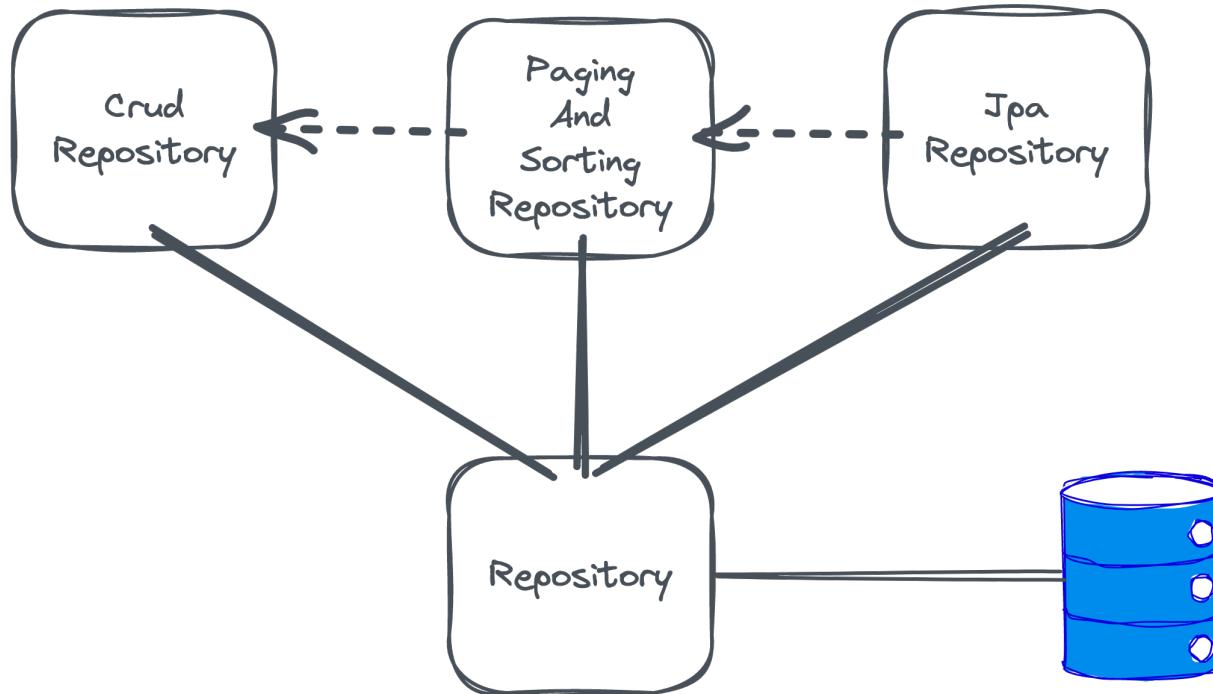
Nullable types in Kotlin
are marked with
a question mark ?

The only possible causes of an “NullPointerException” in Kotlin

NullPointerException
that everyone hates
and wants to avoid.

- An explicit call to `throw NullPointerException()`
- Usage of the `!!` Operator Null Pointer Exception Lovers' Operator
- Data inconsistency with regard to initialization, such as when:
 - An uninitialized `this` available in a constructor is passed and used somewhere
 - A superclass constructor calls an open member whose implementation in the derived class uses an uninitialized state.
- Java interoperation
 - Attempts to access a member of a `null` reference

Repository implementing useful interfaces



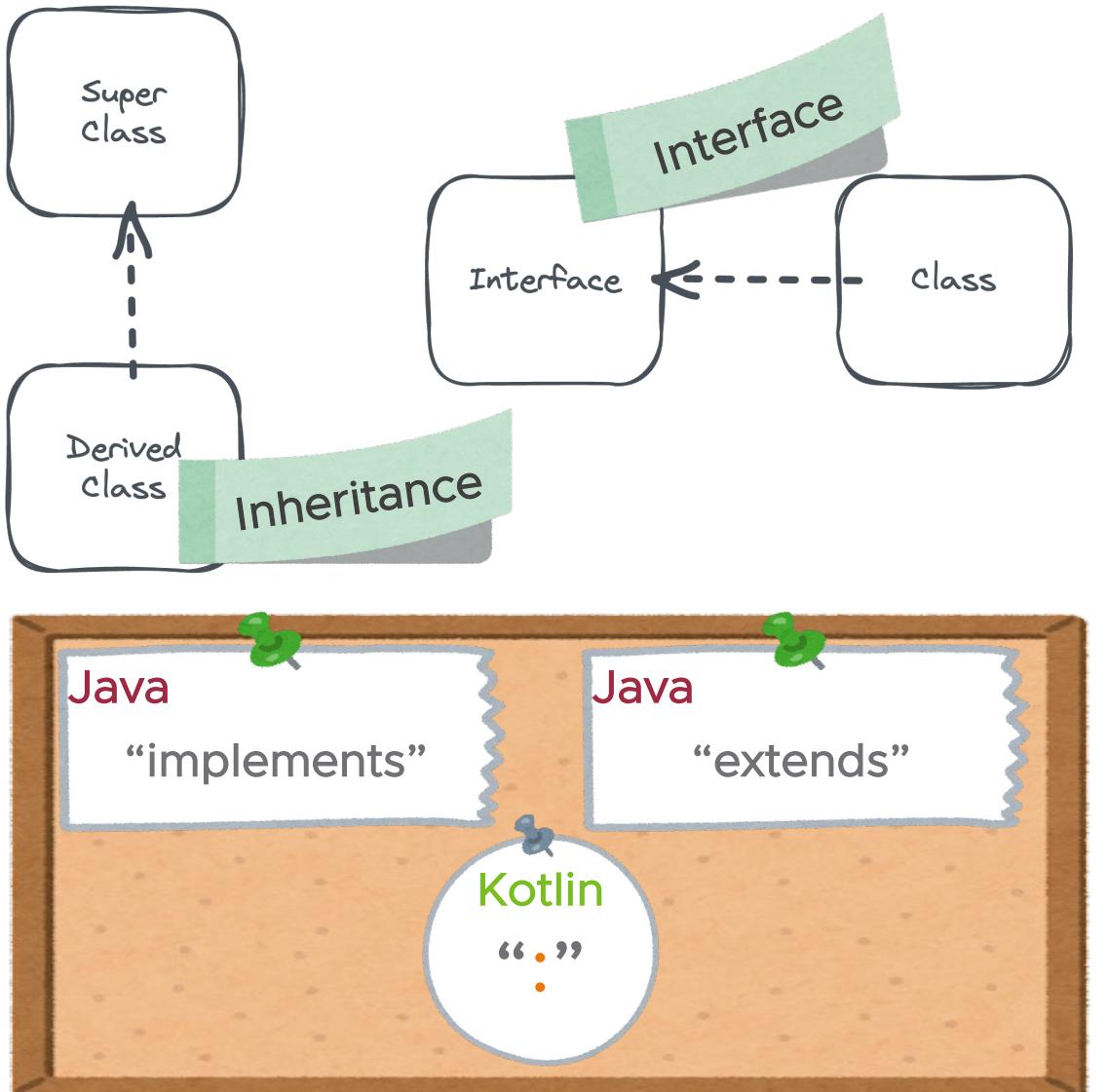
Java

```
public interface JavaBookRepository extends JpaRepository<Book, Long> {  
}
```

Kotlin

```
interface BookRepository : JpaRepository<Book, Long>
```

Kotlin Basics: Interfaces and Inheritance



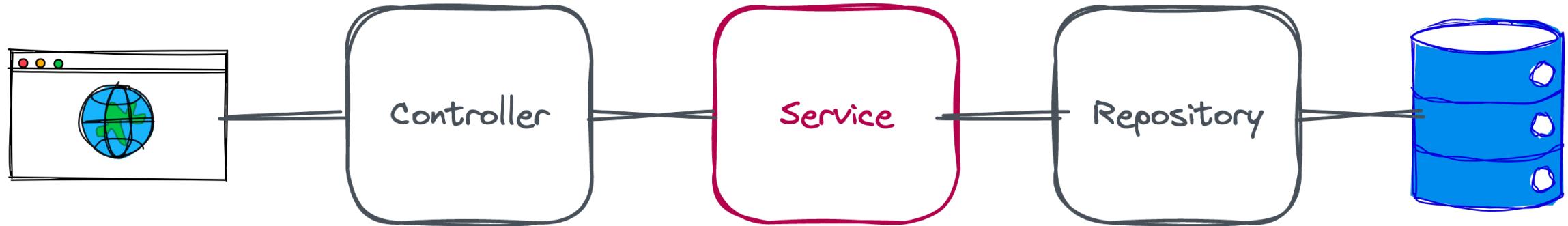
Interfaces and Inheritance,
that's all taken care of by colon ":".

```
abstract class MySuperClass  
interface MyInterface  
  
class MyDerivedClass : MySuperClass(), MyInterface
```

When defining "inheritance",
initialize by specifying the primary constructor
of the super class.

No Need to use the keywords
"extends" and "implements"
interchangeably like Java.

Let's code service layer



If you have made it this far,
you are almost done.
Now let's look at the service
layer.



The almost familiar Spring Service Layer

```
@Service  
class BookService(val repository: BookRepository) {  
    fun getBooks(): MutableList<Book> = repository.findAll()  
  
    fun search(searchBook: SearchBook): MutableList<Book> {  
        val probe = Book()  
  
        if (StringUtils.hasText(searchBook.value)) {  
            probe.apply { this: Book  
                name = searchBook.value  
                author = searchBook.value  
            }  
        }  
  
        val example: Example<Book> = Example.of(  
            probe,  
            ExampleMatcher.matchingAny()  
                .withIgnoreCase()  
                .withStringMatcher(ExampleMatcher.StringMatcher.CONTAINING)  
        )  
        return repository.findAll(example)  
    }  
  
    fun delete(id: Long) {  
        repository.findById(id).Optional<Book>  
    }  
}
```

✓ Service Annotation

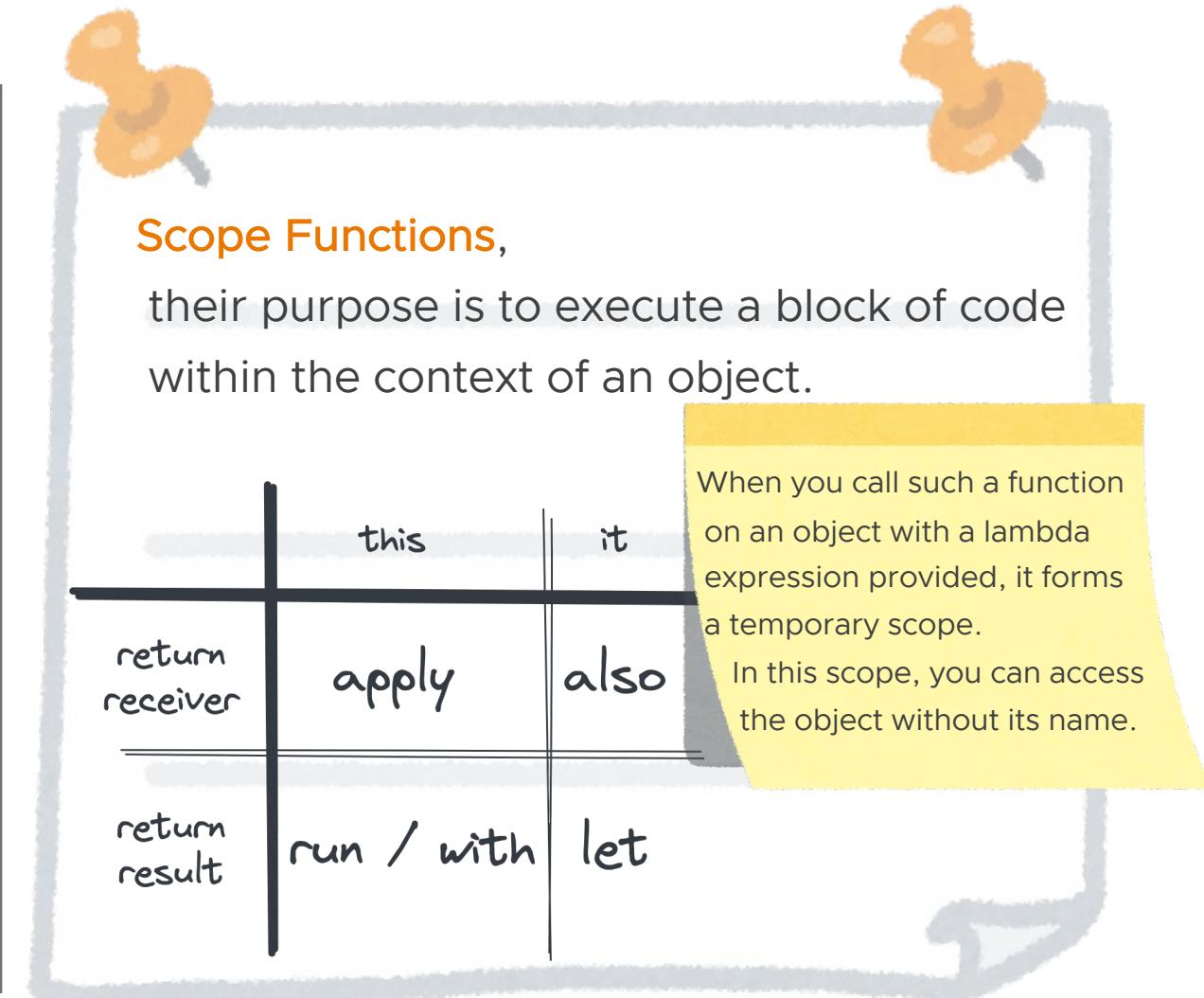
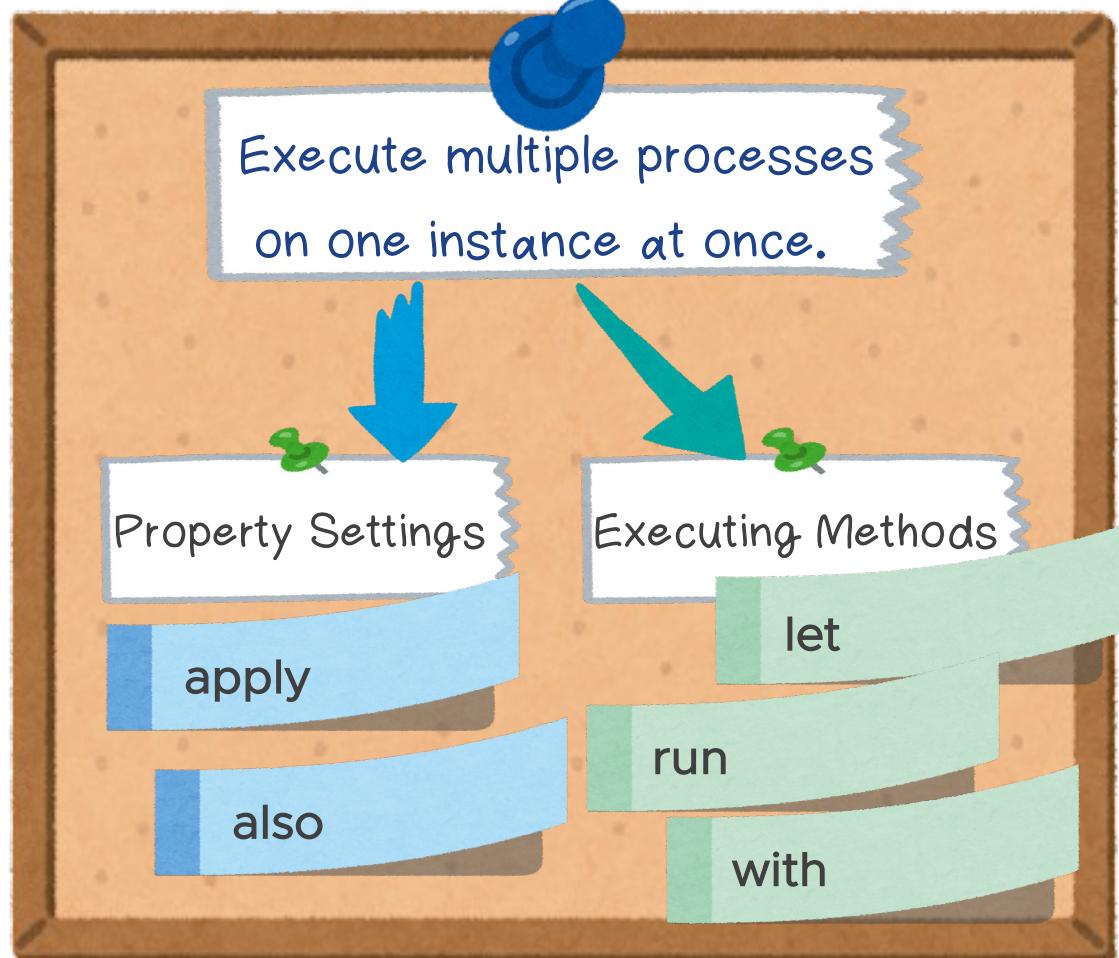
✓ Primary Constructor

✓ Function as Expression

✓ Scope Functions

✓ Spring Data JPA
Query by Example

Kotlin Basics: Scope Functions



The backlogged tasks that still remains

```
fun delete(id: Long) {
    repository.findById(id) <Optional<Book!>
        .map { book: Book ->
            repository.delete(book)
            true ^map
        } <Optional<Book!>
        .orElseThrow(RuntimeException("No
        })
    }
}

fun register(newBook: NewBook) =
    repository.saveAndFlush(Book(newBook.name, newBook.author))

fun update(): Book {
    TODO(reason: "Need to implement a process to update")
}
```

`@kotlin.internal.InlineOnly`

`public inline fun TODO(): Nothing = throw NotImplementedError()`

Always throws `NotImplementedError` stating that operation is not implemented.

Params: `reason` - a string explaining why the implementation is missing.

`@kotlin.internal.InlineOnly`

`public inline fun TODO(reason: String): Nothing = throw NotImplementedError("An operation is not implemented: $reason")`

Nothing

Nothing, Unit, Any

Any

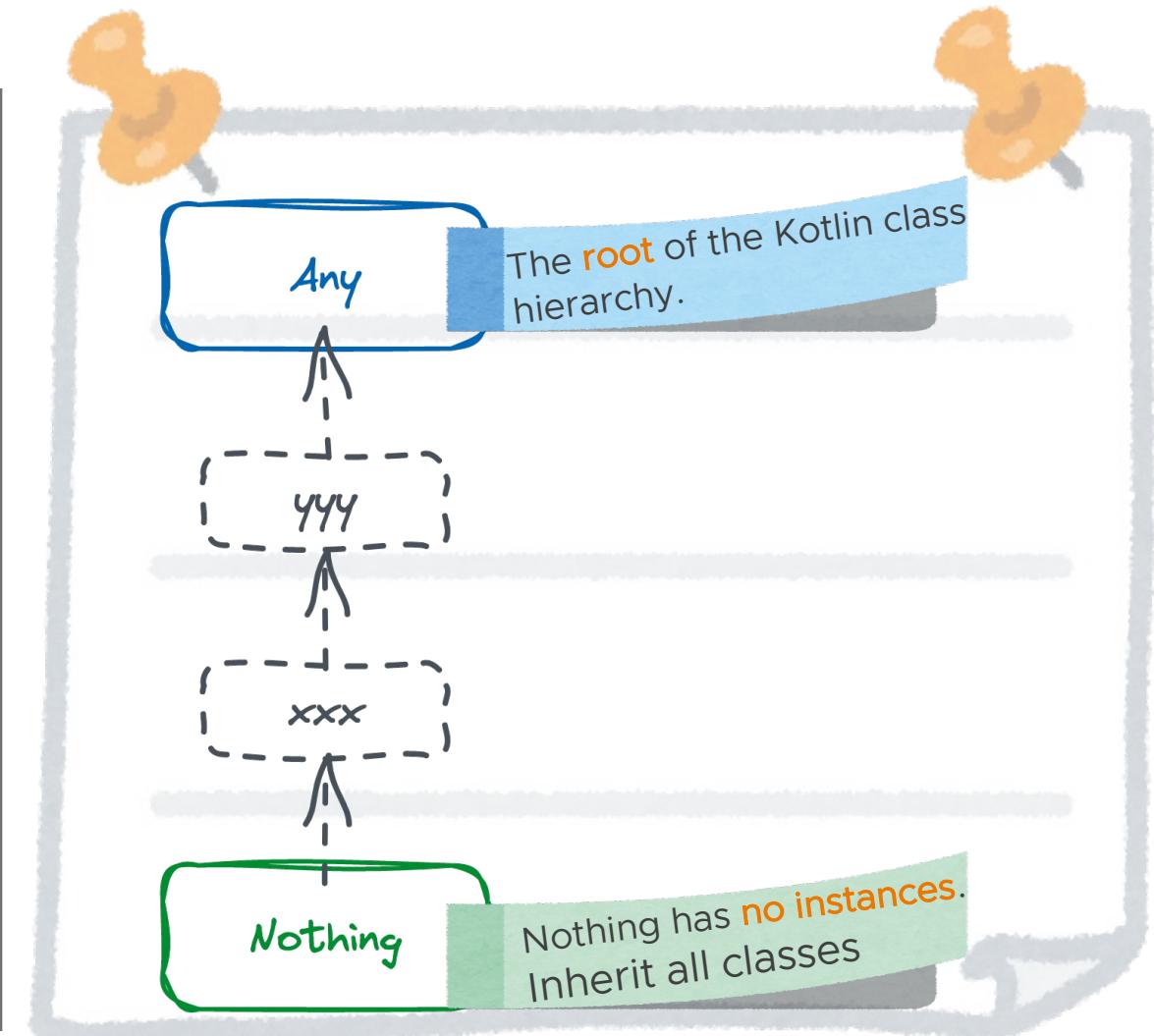
- Any is by default the superclass of all the classes
- This is equal to *Object* class in Java

Unit

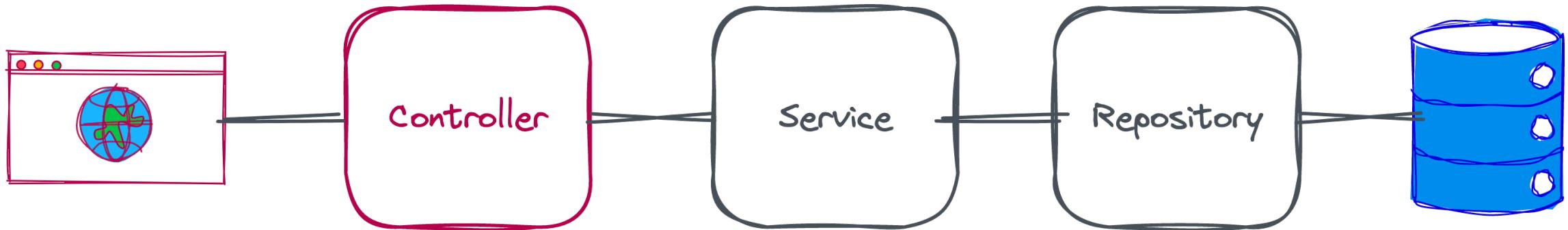
- Unit class is equal to void type in Java
- The superclass of Unit is *Any*

Nothing

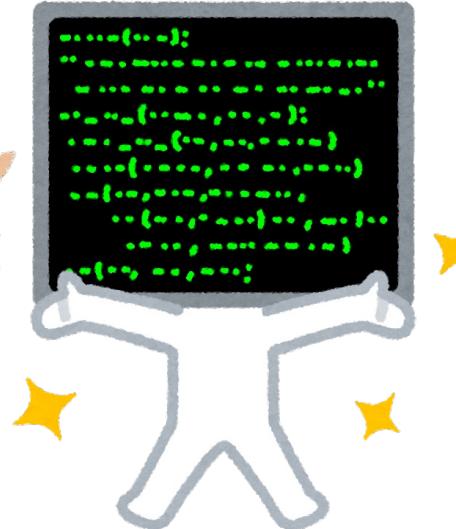
- Nothing is non-open (final class) which **can't be extended**
- We **can't create the instances**
- This is usually used to represent the return type of function which will always throw an exception.
- The superclass of Nothing is Any.



Code service layer



The application is now complete. Let's take a look at the last entry point



Unremarkable controller and entry point

```
@Controller
class BookController(val service: BookService) {

    @GetMapping("/")
    fun index(model: Model): String {
        model.addAttribute(attributeName: "books", service.getBooks())

        return "index"
    }

    @PostMapping("/search")
    fun search(@ModelAttribute searchBook: SearchBook, model: Model): String {
        val searchResult = service.search(searchBook)
        model.addAttribute(attributeName: "books", searchResult)
        return "index"
    }

    @PostMapping("/book/delete/{id}")
    fun delete(@PathVariable id: Long): String {
        service.delete(id)
        return "redirect:/"
    }

    @PostMapping("/book/register")
    fun register(@ModelAttribute book: NewBook): String {
        logger.info("Author: ${book.author}")
        logger.info("Name: ${book.name}")
        service.register(book)
        return "redirect:/"
    }
}
```

```
@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

val Any.logger: Logger
    get() = LoggerFactory.getLogger(this.javaClass)
```

Extension properties
With Kotlin extensions,
you can add a new
property to an existing
class.

Extensions

Extensions

Kotlin provides the ability to extend new functionality to a class without inheriting from the class.

Extension functions

Extension functions allow you to add functions even if you did not create the class yourself. It is possible to extend the core library's `Int` class, for example.

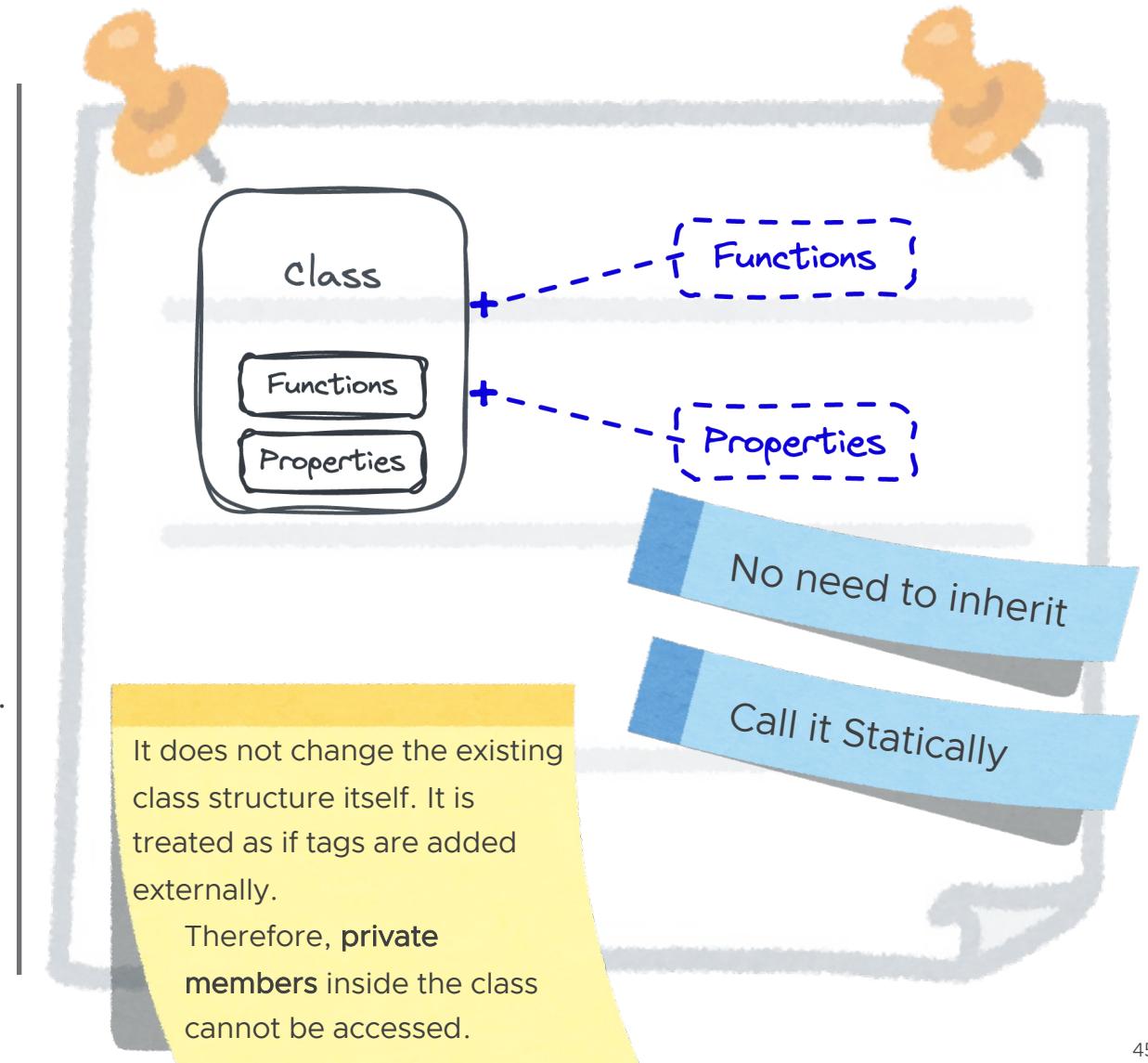
```
[0] fun Int.square(): Int = this * this
[1] println("10 * 10 = ${10.square()}")
10 * 10 = 100
```

- Extended functions are used **statically**

Extension properties

Like functions, properties can be added to existing classes.

```
[0] val String.size: Int
     get() = this.length
[1] println("This is an Extension property".size)
29
```



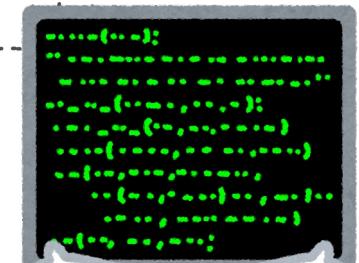
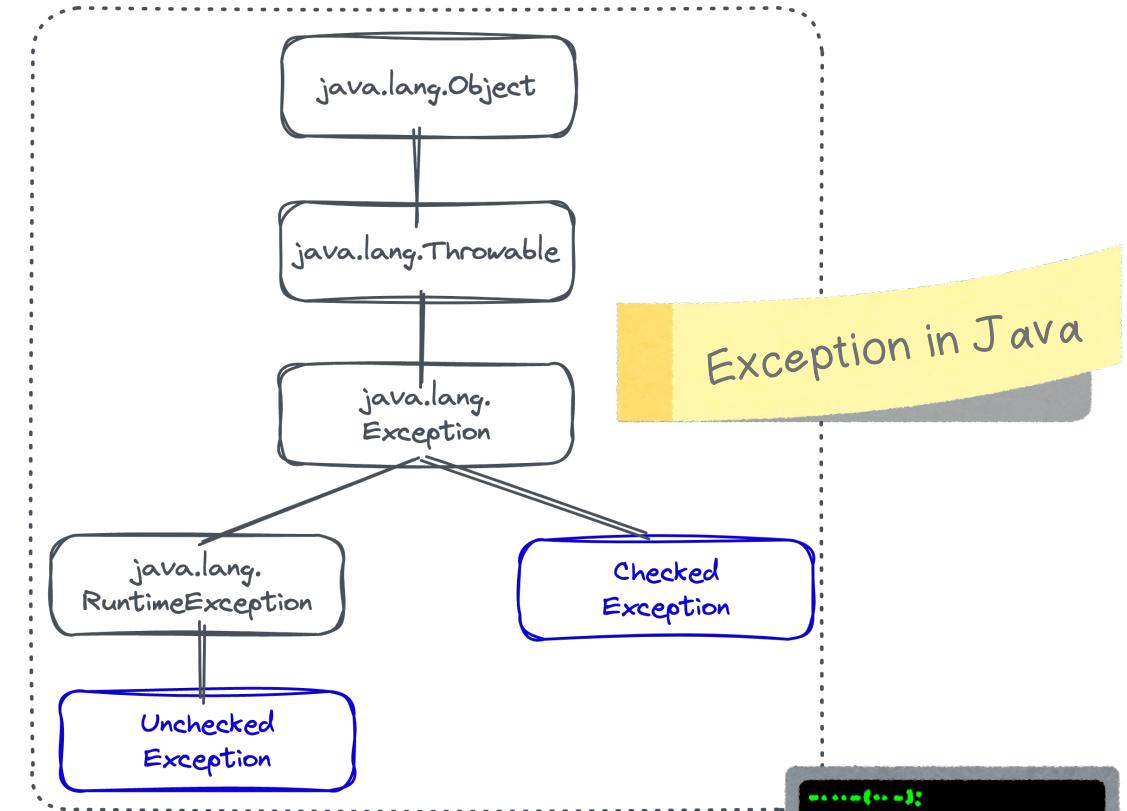
Now, you can run the Kotlinized Spring Boot



Exception Handling: It's not enough that it runs

```
fun delete(id: Long) {  
    repository.findById(id) Optional<Book!>  
        .map {  
            book: Book ->  
            repository.delete(book)  
            true ^map  
        } Optional<Boolean!>  
        .orElseThrow {  
            RuntimeException(  
                "No book at $id"  
            )  
        }  
}
```

In this code example, a Runtime Exception is thrown,
but in Kotlin it does not have to be thrown to cause a compile error.
Kotlin makes no distinction between checked and unchecked exceptions.



Exception Handling in Kotlin

try-catch

```
try {
    doSomething()
} catch (e: RuntimeException) {
    throw e
}
```

runCatching

```
val result: Result<Any> = runCatching { doSomething() }

result
    .onSuccess { doSomethingOnSuccess() }
    .onFailure { doSomethingOnFailure() }
```

The [runCatching](#) function executes the given block and returns an object of type [Result](#).

Return class and runCatching function

```
val result: Result<Type> =
    runCatching { process_that_mayCause_an_exception() }
```

If an exception is thrown within the block, the [runCatching](#) function will not throw it out.

The return value and the exception can be obtained from the [Result](#) object returned by the [runCatching](#) function.

This time is the last. And just one more thing...

If you're interested in Kotlin,
you may have seen that story...



Gradle Kotlin DSL

Kotlin DSL is now the default for new Gradle builds.

blog.gradle.org

Kotlin DSL is Now the Default for New Gradle Builds

April 13, 2023 · Paul Merlin · General

Kotlin DSL for Gradle was introduced in version 3.0 of the Gradle Build Tool in August 2016 and released as 1.0 in Gradle 5.0. Since then, it's been adopted by many organizations and has improved the authoring experience of many Gradle builds.

Kotlin DSL is now the default choice for new Gradle builds. If you're starting a project with Gradle, including in IntelliJ IDEA (Android Studio, Giraffe), Kotlin DSL is the default option. Such is the case for most of the Gradle ecosystem, or those who prefer to use it.

In this post, we will explore the benefits of using Kotlin DSL as the default choice for new Gradle builds. We will also discuss what we can do in the future to make Kotlin DSL even better.

8:32 PM · Apr 13, 2023 · 22.5K Views

Kotlin DSL is Now the Default for New Gradle Builds

April 13, 2023 · Paul Merlin · General

Kotlin DSL for Gradle was introduced in version 3.0 of the Gradle Build Tool in [August 2016](#) and released as 1.0 in [Gradle 5.0](#). Since then, it's been adopted by many organizations and has improved the authoring experience of many Gradle builds.

Kotlin DSL is now the default choice for new Gradle builds. If you're starting a project with Gradle, including in IntelliJ IDEA (Android Studio, Giraffe), Kotlin DSL is the default option. Such is the case for most of the Gradle ecosystem, or those who prefer to use it.

In this post, we will explore the benefits of using Kotlin DSL as the default choice for new Gradle builds. We will also discuss what we can do in the future to make Kotlin DSL even better.



Technology Radar

Search Techniques Platforms Tools Languages & Frameworks

Languages & Frameworks

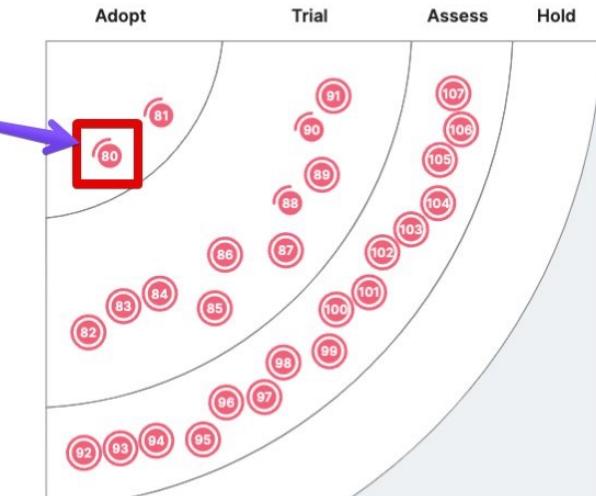
Adopt

80. Gradle Kotlin DSL

81. PyTorch

Trial

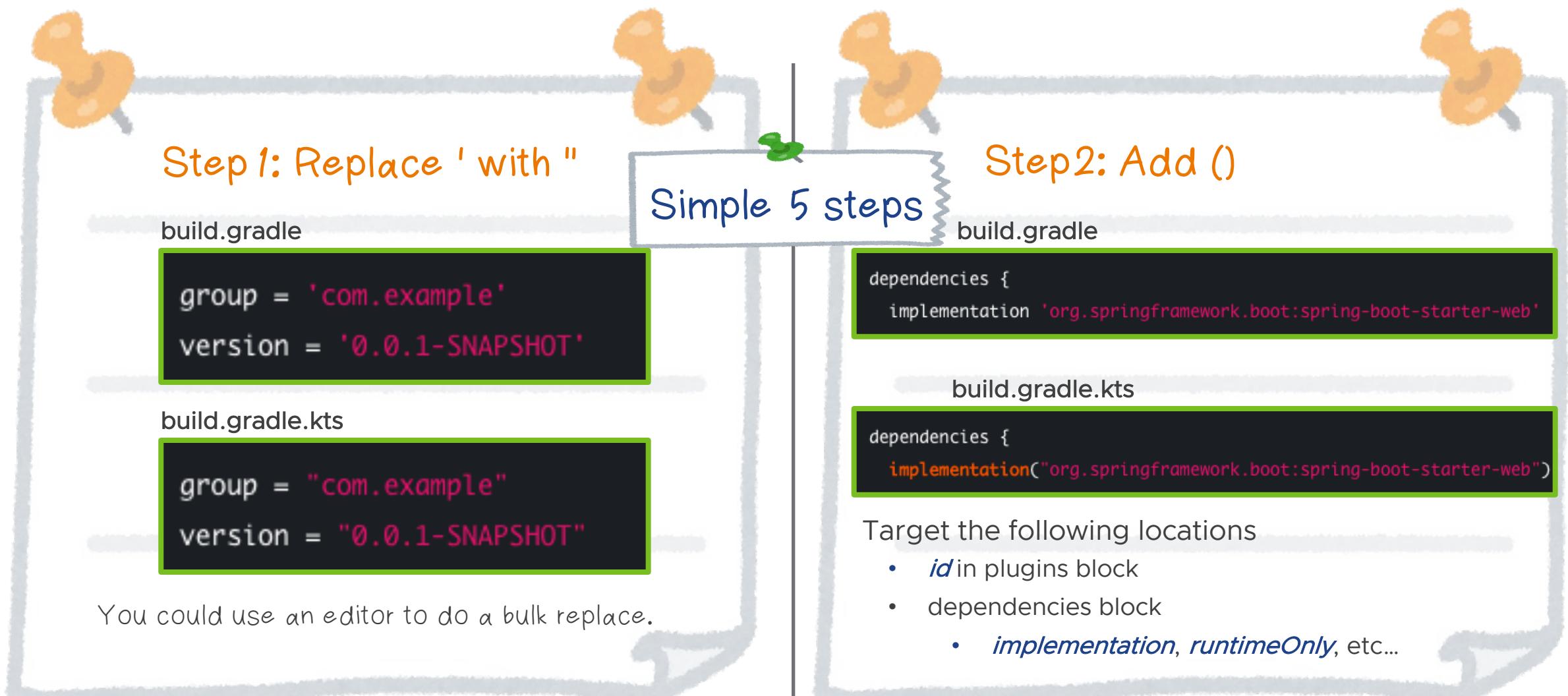
Kotlin



<https://www.thoughtworks.com/radar/languages-and-frameworks>

<https://blog.gradle.org/kotlin-dsl-is-now-the-default-for-new-gradle-builds>

Petit Tips on migration from “build.gradle” to “build.gradle.kts”



Petit Tips on migration from “build.gradle” to “build.gradle.kts”

Step3: Add =

In Groovy, you can omit the '=' assignment operator when assigning properties, which is required in Kotlin

build.gradle

```
java {  
    sourceCompatibility JavaVersion.VERSION_17  
    targetCompatibility JavaVersion.VERSION_17  
}
```

build.gradle.kts

```
java {  
    sourceCompatibility = JavaVersion.VERSION_17  
    targetCompatibility = JavaVersion.VERSION_17  
}
```

Step4: Rename “build.gradle” to “build.gradle.kts”

 Build & Error

 gradlew assemble

 error

Repeat Steps 1 - 3,
commenting out any lines that cause errors.



Step 5: Fix lines that cause errors

build.gradle

```
test {  
    reports.html.enabled = false  
}
```

build.gradle.kts

```
tasks.withType<Test> {  
    reports {  
        html.isEnabled = false  
    }  
}
```

build.gradle

```
task clean(type: Delete) {  
    delete rootProject.buildDir  
}
```

build.gradle.kts

```
tasks.create<Delete>("clean") {  
    delete(rootProject.buildDir)  
}
```

(Step 6): Accumulate tips as you fix it

Learn as you accumulate the
revisions that occur with each project!

Key Takeaways



Spring has been and always will be fun

The productivity and fun from Kotlin is huge

Learning Kotlin and then
using it for Spring is a good thing

Learning Kotlin from Spring and
having fun with it is also a good thing

Spring Academy

spring ACADEMY Courses Learning Path Instructors My Account

Learn from the experts.
Build like a boss.

On-demand education developed and curated by the world's foremost experts in Spring

Check Out Courses →

“Spring Academy is an invaluable resource for expanding your knowledge and skills.”
Josh Long
Spring Developer Advocate

Real-world training, from the best in the community

Created by the stewards of the Spring framework, Spring Academy is a comprehensive, project-based learning platform providing the Spring community with the knowledge and experience needed to stay ahead of the curve.

My Account

COURSES

Learn everything you want to know

Begin or advance your mastery of Spring with hands-on courses created by experts.

Topics

- Java 4
- REST 3
- Security 3
- Database 3
- Testing 3
- Deployment 3
- Observability 3
- Modernization 2
- Tooling 1

Skill Levels

- Beginner 2
- Intermediate 2

Status

Beginner

Building a REST API with Spring Boot

In this beginner course, you'll learn how to build a complete REST API from start to finish with Spring Boot. With our interactive labs, you'll get hands-on practice every step of the way — bootstrapping with Spring Initializr, through authenticating & authorizing with Spring Security.

Start Course →

21 Lessons | 5h 18m

Beginner

Introduction to the Spring Professional Learning Path

Here's your introduction to the Spring Professional learning path, where you'll explore the most commonly used features of Spring. If you're an experienced Java developer, and new to Spring Framework, this learning path is for you.

Start Course →

4 Lessons | 34m

You can try today! <https://spring.academy/>

2023

SPRING  10TH ANNIVERSARY

#springio23

THANK YOU!

Shinya Yanagihara

@yanashin18618



BARCELONA MAY 18-19 / WWW.SPRINGIO.NET