How to enjoy Spring with Kotlin

for

柳原 伸弥 / Shinya Yanagihara

Developer Advocate

VMware Japan

October 12, 2023

About me

Shinya Yanagihara

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

- **SNS:**
  - @yanashin18681
  - @yanashin@mastodon.social
- **My motto:**

  “Work for the Developer Experience”

Qustion

Qustion

My Answer

I love Kotlin and Spring.

Spring Framework 5.0 goes GA

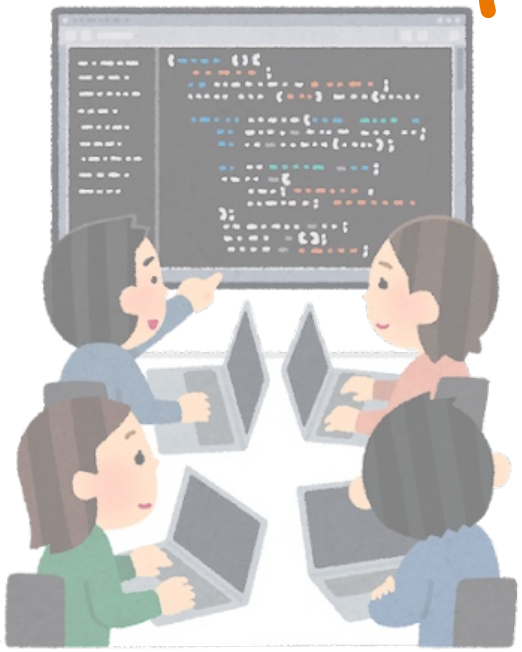
RELEASES | JUERGEN HOELLER | SEPTEMBER 28, 2017 | 47 COMMENTS

Dear Spring community,

It is my pleasure to announce that, after more than a year of milestones and RCs and almost two years of development overall, Spring Framework 5.0 is finally generally available as 5.0.0.RELEASE from repo.spring.io and Maven Central!

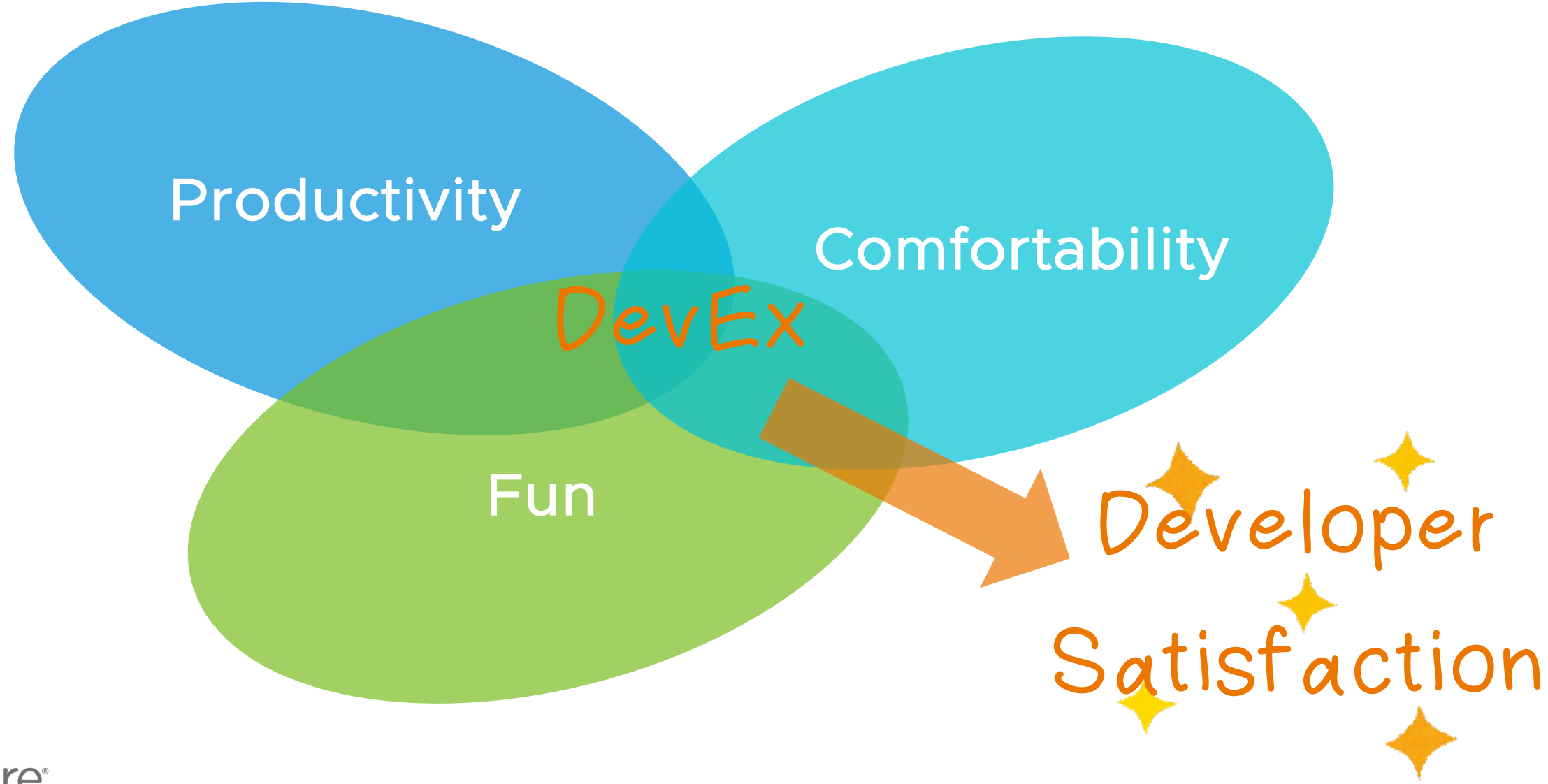
This brand-new generation of the framework is ready for 2018 and beyond: with support for JDK 9 and the Java EE 8 API level (e.g. Servlet 4.0), as well as comprehensive integration with Reactor 3.1, JUnit 5, and the Kotlin language. On top of that all, Spring Framework 5 comes with many functional API variants and introduces a dedicated reactive web framework called Spring WebFlux, next to a revised version of our Servlet-based web framework Spring MVC.

It's all about the Developer Experience.

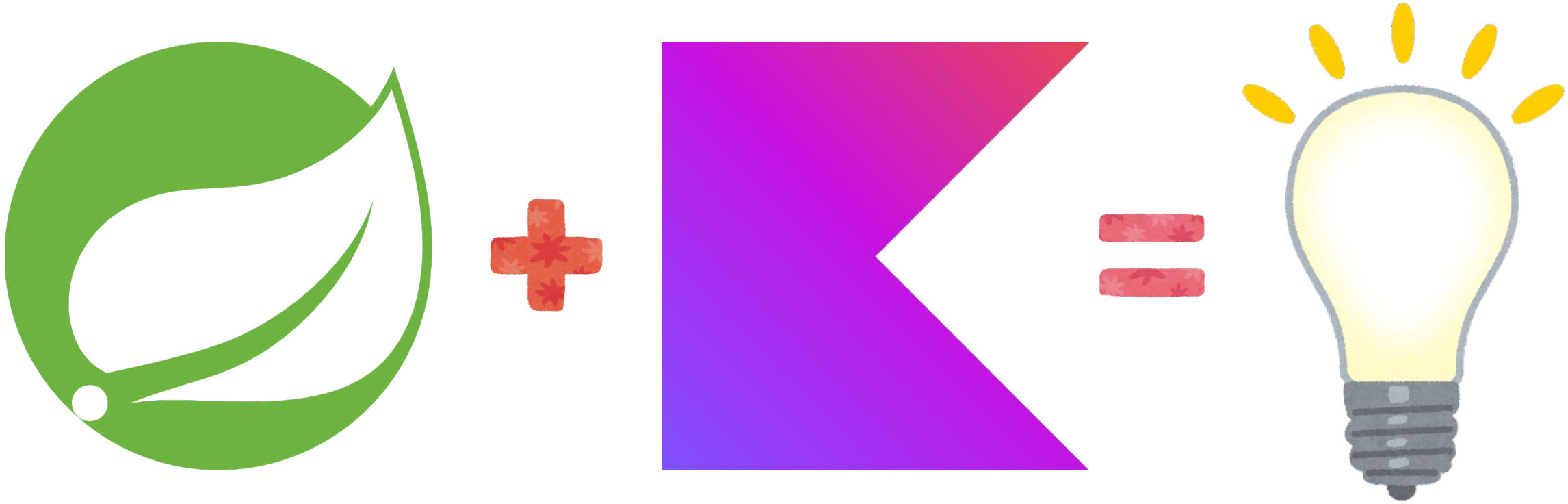


What is the Developer Experience

Developer Experience



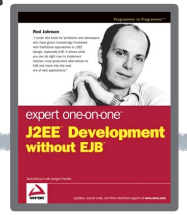
Developer Satisfaction with Spring and Kotlin



Developer Experience for Spring

The Origin of Spring

Java Development without EJB



Spring Cloud

Simplifying the Complexity of Distributed Applications

Spring Boot

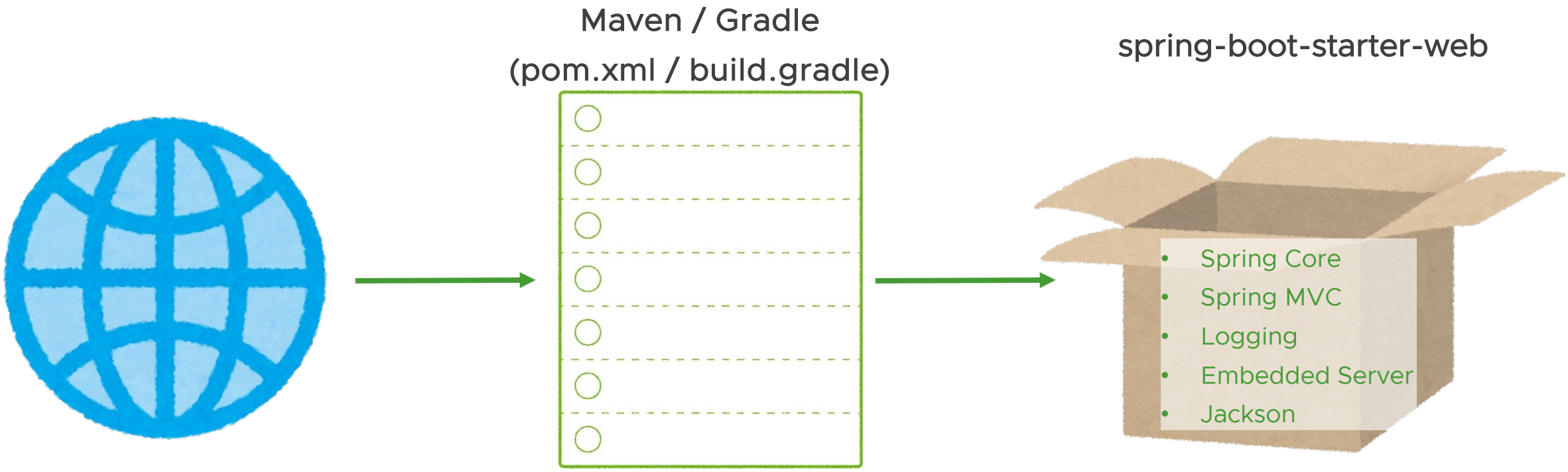
Autonomous configuration and execution

And more...

A I/ML, Modular Monolith, and etc...

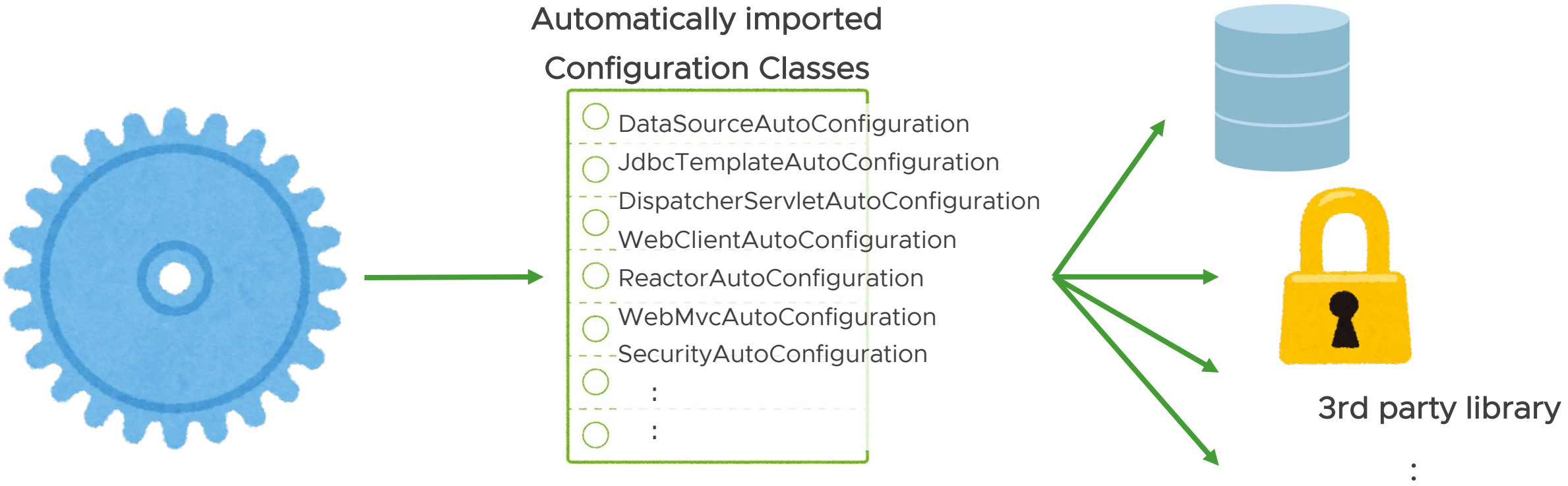
Developer Experience for Spring

Spring Boot Starter



Automatic retrieval of a set of libraries required for development

Auto-Configuration



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

Spring Boot Actuator

Expose operational information about the running application

Endpoint

- Health
- Metrics
- Info
- Dump
- Env

Observability

- Loggers
- Metrics
- Tracing
- Auditing

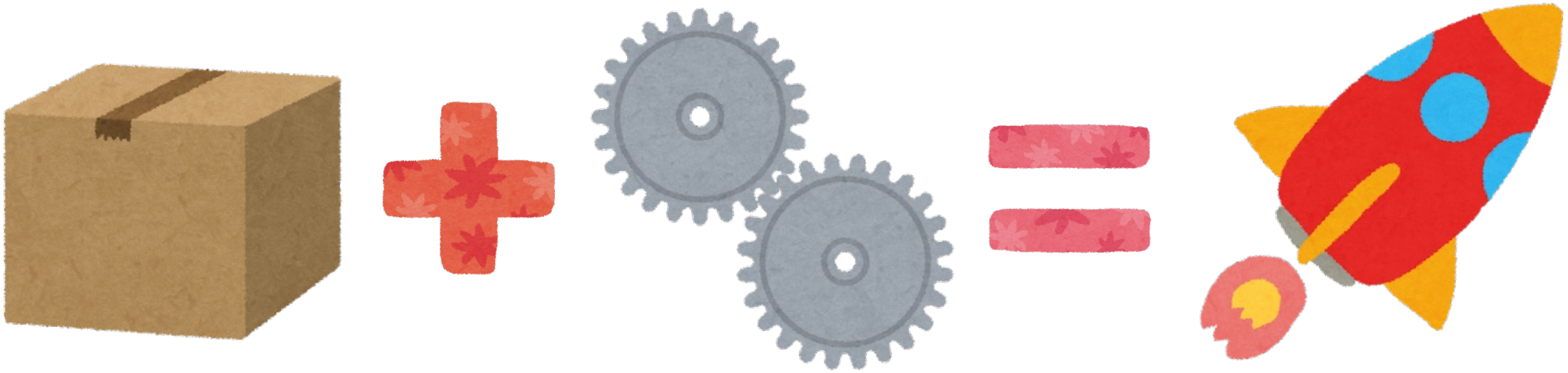
Bring production-ready features to your application without implementing yourself.

Executable JAR & Embedded Server

Application

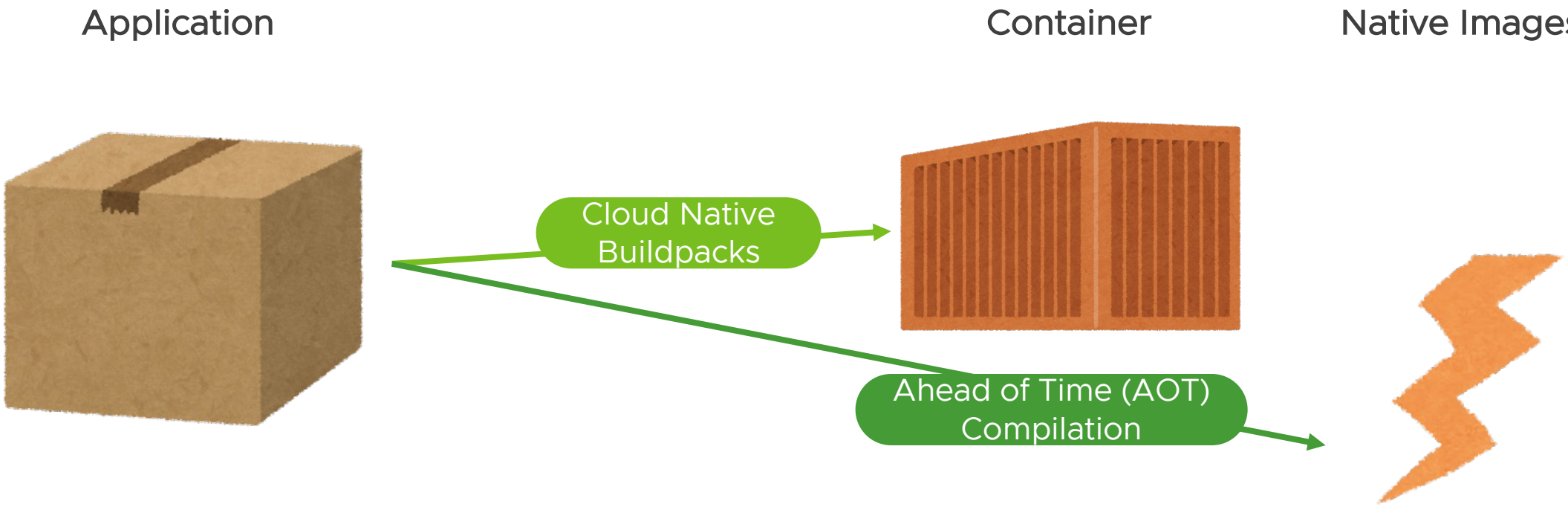
Embedded Server
(Tomcat, Jetty, Undertow)

Executable JAR



Autonomously running application binaries

Containers & Native Images



Easy build convenient container images
and fast-running native images

Developer Experience for Kotlin

✅ Simpler

✅ Easier

✅ Conciser

👨‍💻 Less

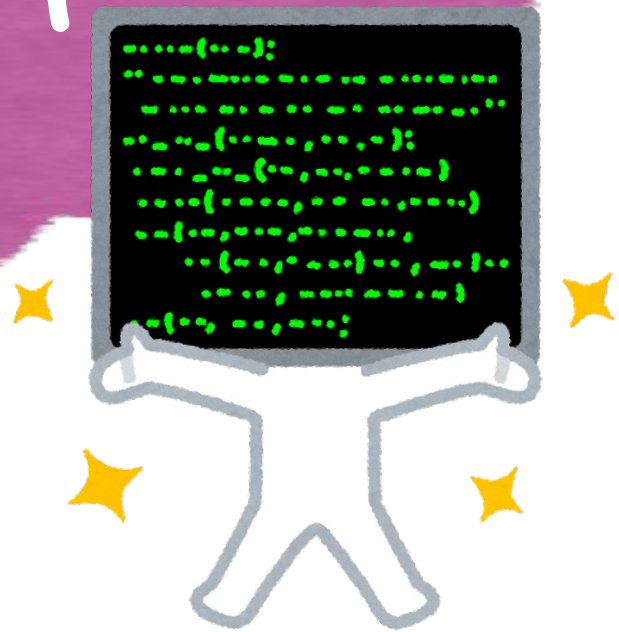
👩‍💻 Lines

👨‍💻 of Code

☕ Java Compatibilty

☕ 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

build.gradle

build.gradle

Users > yanagiharas > Downloads > demo > build.gradle

```
1  import org.jetbrains.kotlin.gradle.tasks.KotlinCompile
2
3  plugins {
4      id 'org.springframework.boot' version '3.1.4'
5      id 'io.spring.dependency-management' version '1.1.3'
6      id 'org.jetbrains.kotlin.jvm' version '1.8.22'
7      id 'org.jetbrains.kotlin.plugin.spring' version '1.8.22'
8  }
9
10 group = 'com.example'
11 version = '0.0.1-SNAPSHOT'
12
13 java {
14     sourceCompatibility = '17'
15 }
16
17 repositories {
18     mavenCentral()
19 }
20
21 dependencies {
22     implementation 'org.springframework.boot:spring-boot-starter'
23     implementation 'org.jetbrains.kotlin:kotlin-reflect'
24     testImplementation 'org.springframework.boot:spring-boot-starter-test'
25 }
26
27 tasks.withType(KotlinCompile) {
28     kotlinOptions {
29         freeCompilerArgs += '-Xjsr305=strict'
30         jvmTarget = '17'
31     }
32 }
33
34 tasks.named('test') {
35     useJUnitPlatform()
36 }
37
```

Kotlin JVM plugin

It enables you to use Kotlin

build.gradle.kts

build.gradle.kts

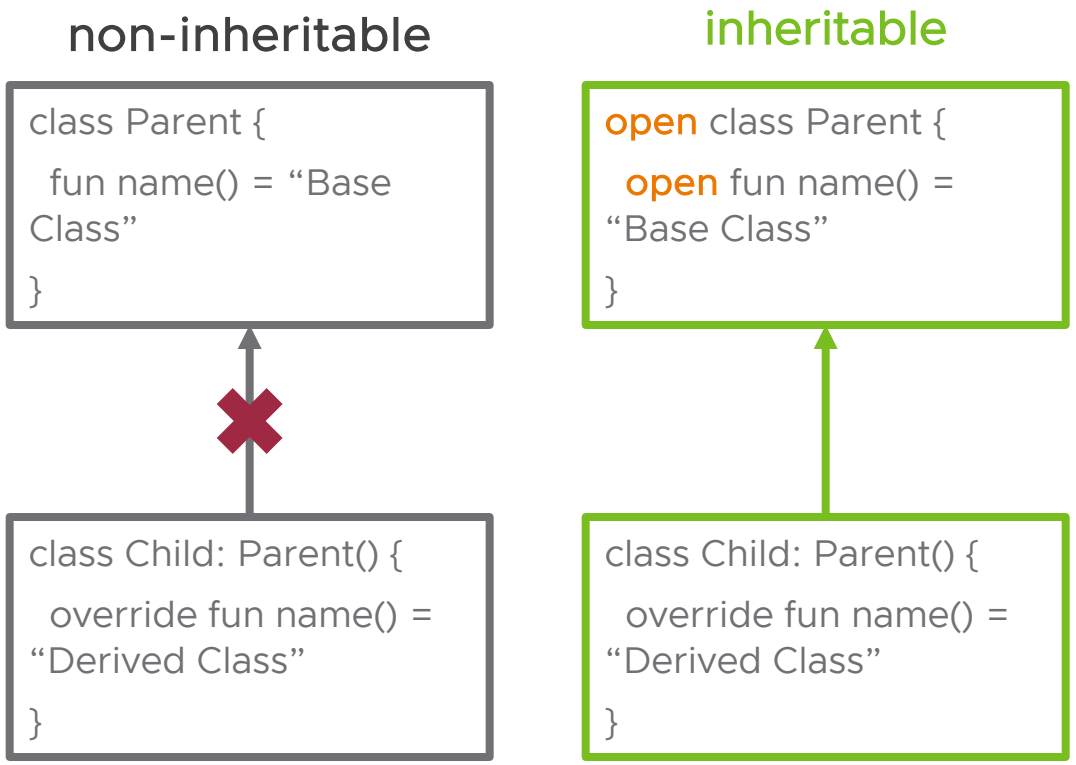
Users > yanagiharas > Downloads > demo > build.gradle.kts

```
1  import org.jetbrains.kotlin.gradle.tasks.KotlinCompile
2
3  plugins {
4      id("org.springframework.boot") version "3.1.4"
5      id("io.spring.dependency-management") version "1.1.3"
6      kotlin("jvm") version "1.8.22"
7      kotlin("plugin.spring") version "1.8.22"
8  }
9
10 group = "com.example"
11 version = "0.0.1-SNAPSHOT"
12
13 java {
14     sourceCompatibility = JavaVersion.VERSION_17
15 }
16
17 repositories {
18     mavenCentral()
19 }
20
21 dependencies {
22     implementation("org.springframework.boot:spring-boot-starter")
23     implementation("org.jetbrains.kotlin:kotlin-reflect")
24     testImplementation("org.springframework.boot:spring-boot-starter-test")
25 }
26
27 tasks.withType<KotlinCompile> {
28     kotlinOptions {
29         freeCompilerArgs += "-Xjsr305=strict"
30         jvmTarget = "17"
31     }
32 }
33
34 tasks.withType<Test> {
35     useJUnitPlatform()
36 }
37
```

Kotlin - Spring compiler plugin

It helps Spring implementations with Kotlin

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



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

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”
}
```

Spring AOP requires a class to be "open” in its mechanism.
This plugin implicitly opens classes that should be open.

@Component
@Async
@Transactional
@Cacheable
@SpringBootTest

The plugin specifies the following annotations

Kotlin Basics: Class and Constructor

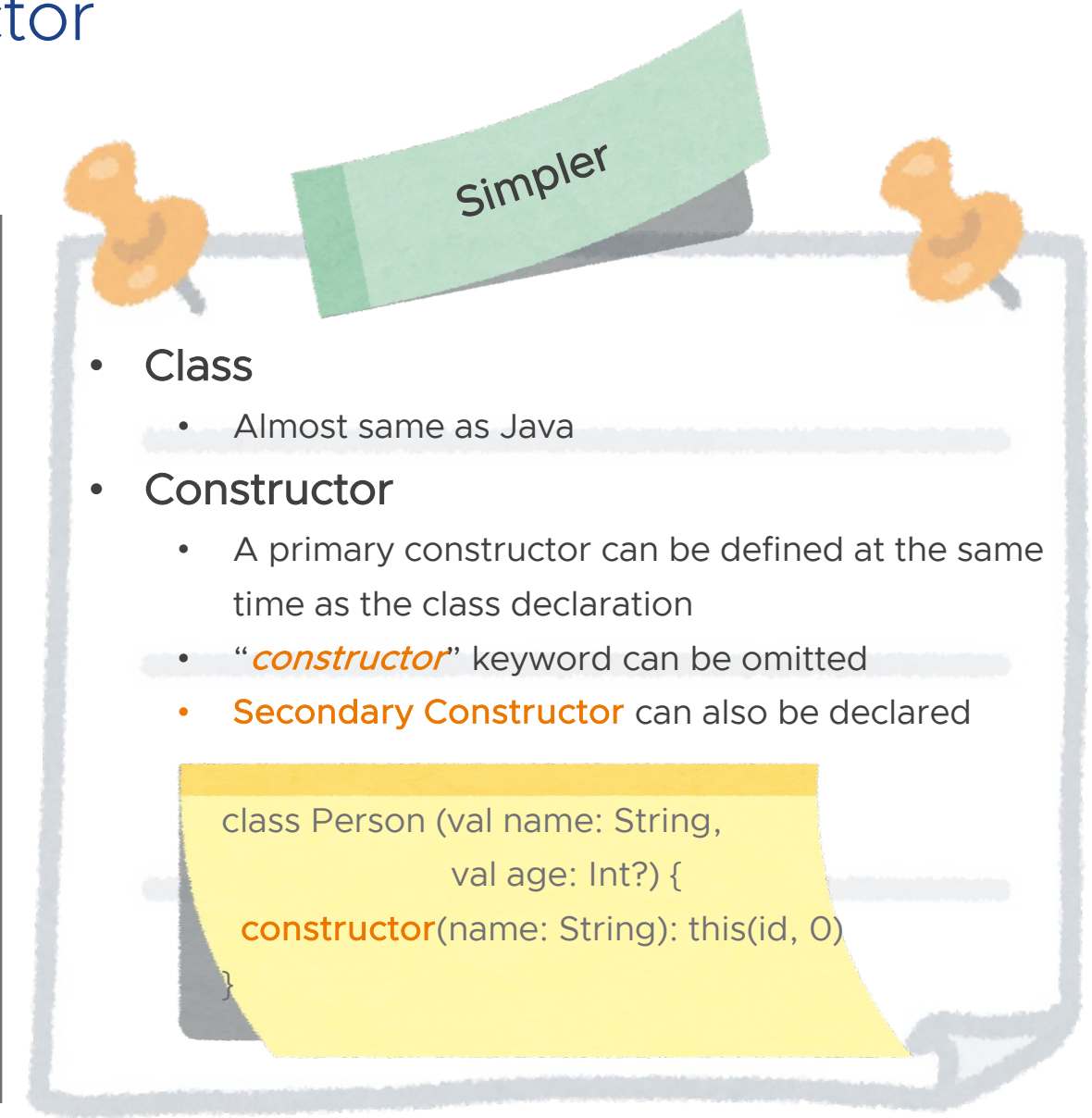
Java

```
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

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }

}
```

Kotlin

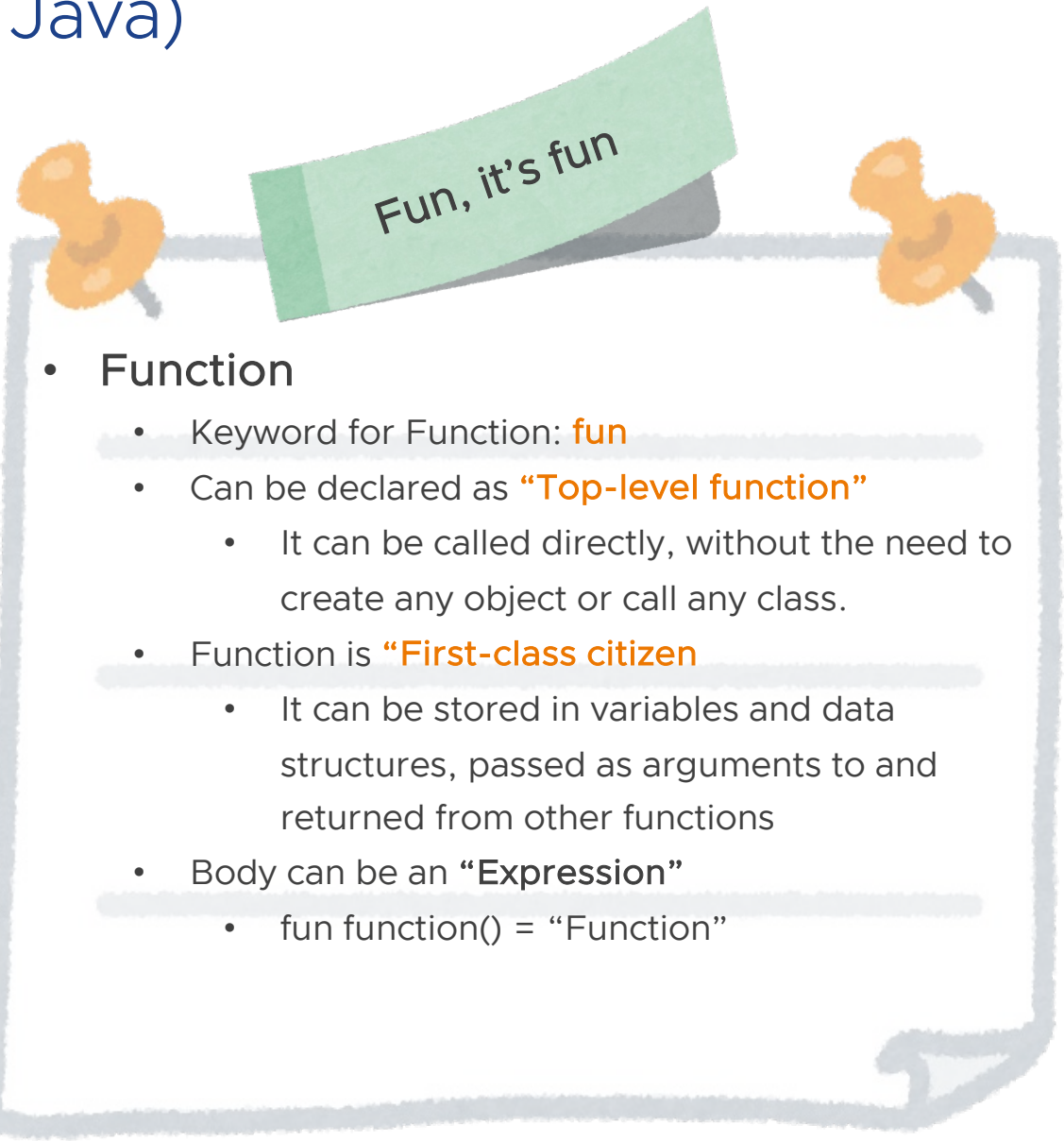
```kotlin
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

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

Top-level function

Function is first-class citizen

Kotlin Basics: Function (Method in Java)

'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
```

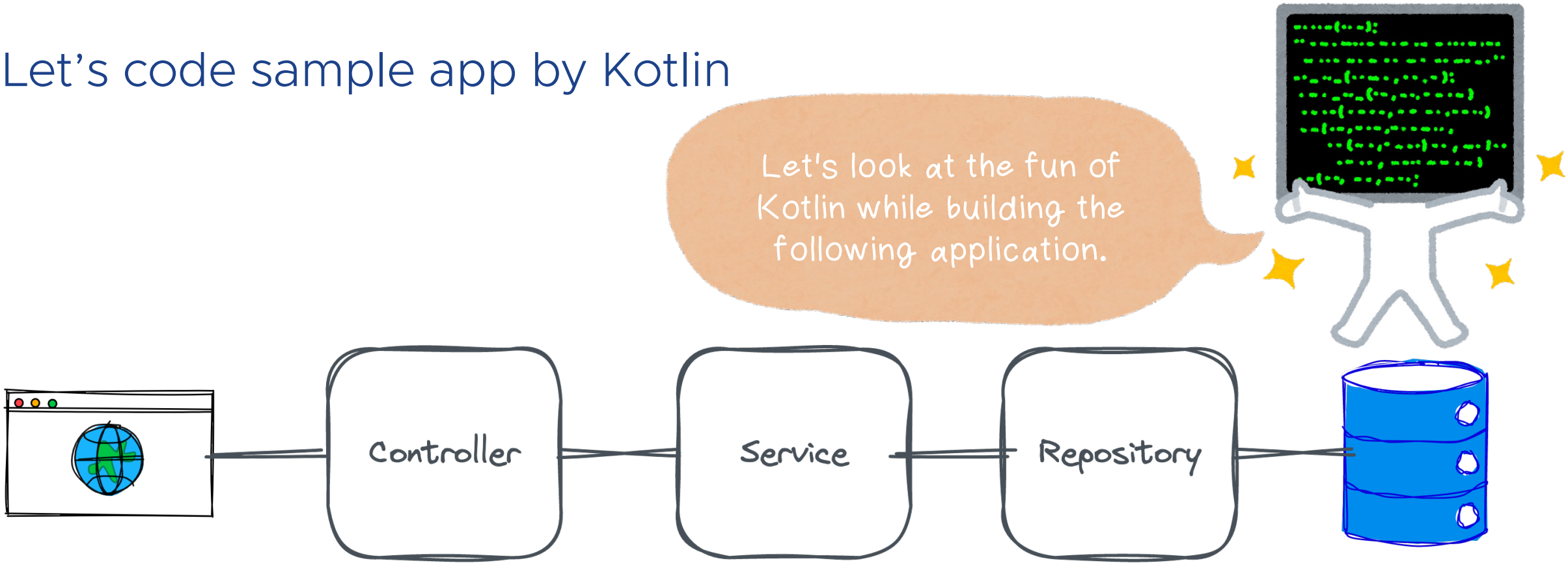
fun add(a: Int, b: Int) = a + b

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

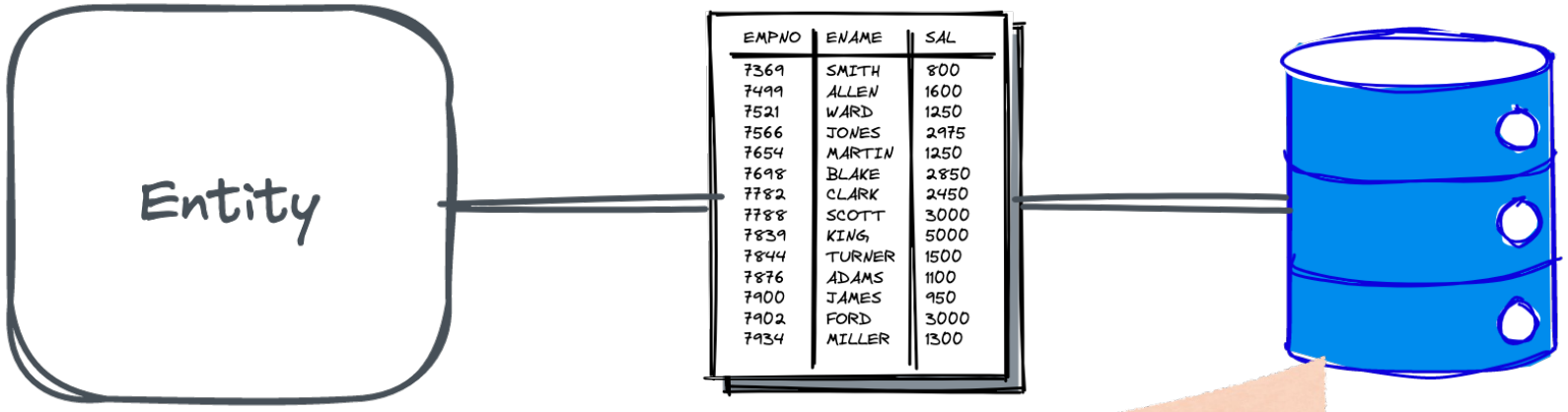


1. Controller to control requests

2. Services to control processing

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

Entity for Data model



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

Kotlin has Syntax like Java's Record.

Data Class

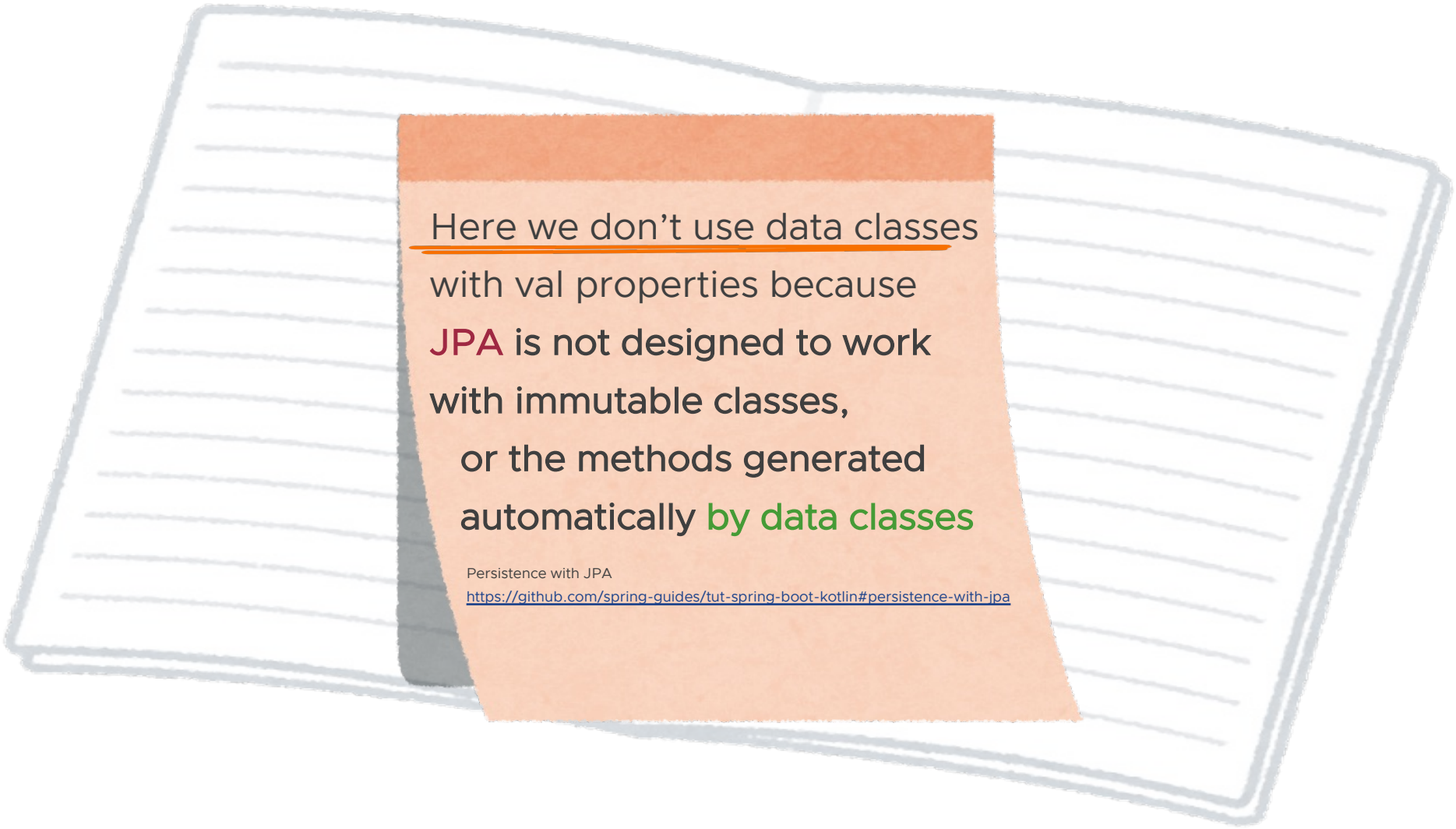
Why not use Java Record for Entity?

Records Can't Be Entities

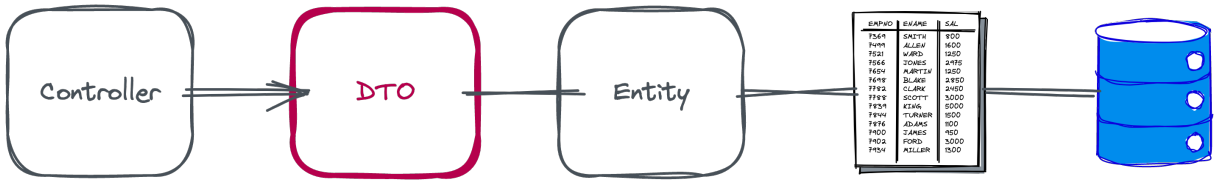
Entities are handled by the JPA provider. JPA providers are responsible for creating the database tables, mapping the entities to the tables, and persisting the entities to the database. In popular JPA providers like Hibernate, entities are created and managed using proxies.

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)



DTO

```
package com.example.demo.dto

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

Entity

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

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

Data class,

it is literally a class for holding data.

The compiler automatically generates the following functions for data classes:

- equals()
- hashCode()
- toString()
- copy()
- componentN()
  - component1(): 1st field
  - component2(): 2nd field

No need for Lombok

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: "")
}
```

Type Safety — Java, Kotlin

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

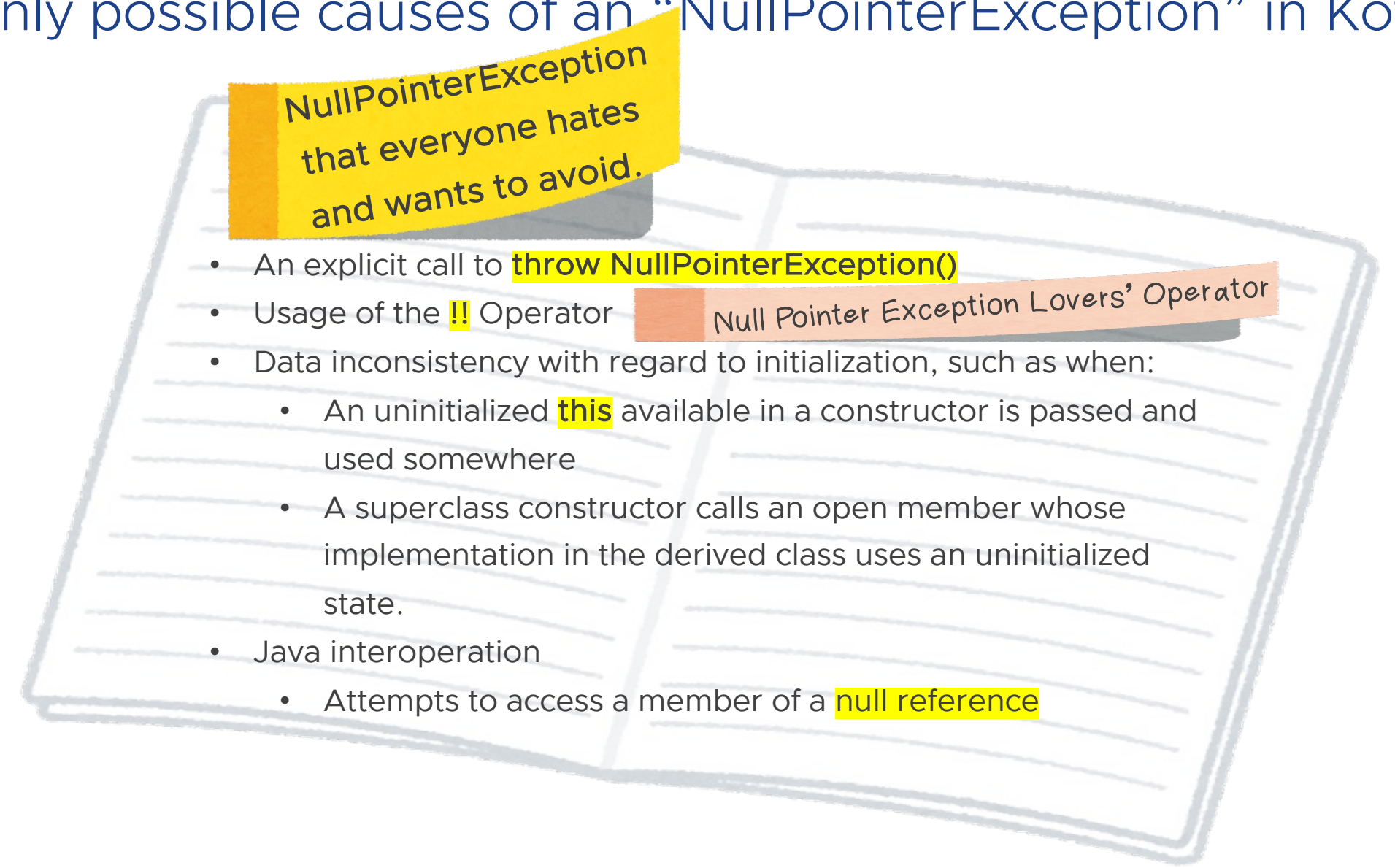
Null Safety — Kotlin

Nullable types in Kotlin are marked with a question mark "?"

Kotlin's type system basically does not allow null references.

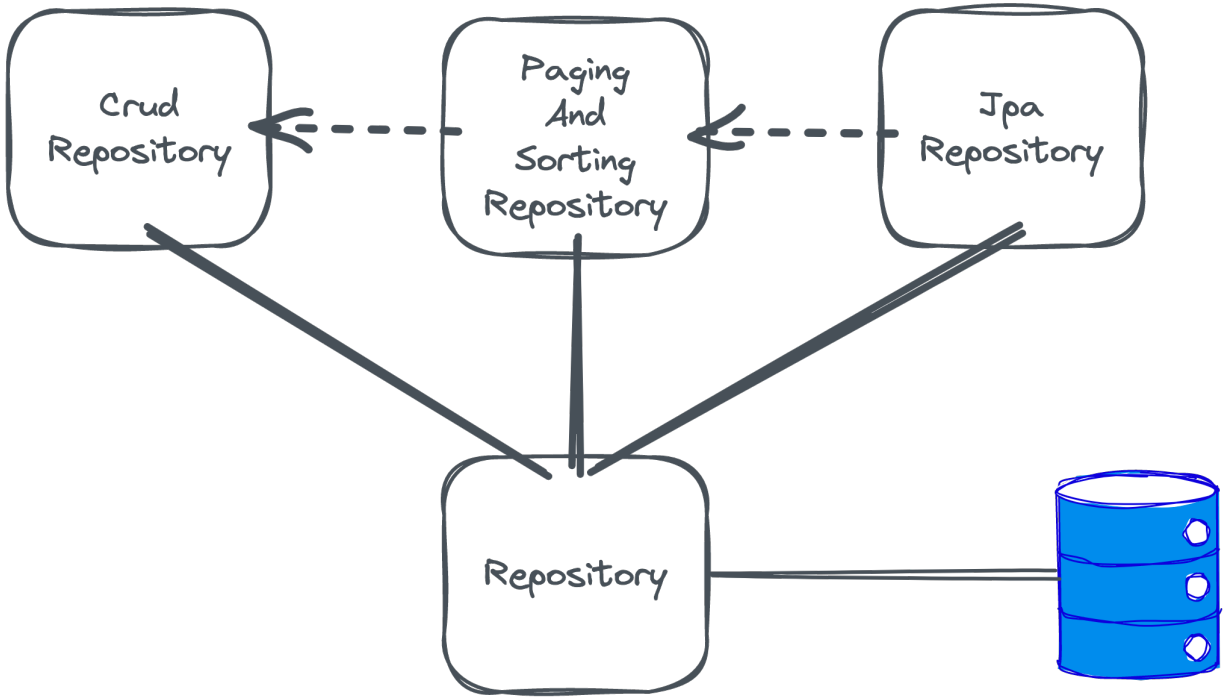
```
var a: String = "abc"
a = null // compilation error

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

The only possible causes of an “NullPointerException” in Kotlin



- An explicit call to **throw NullPointerException()**
- Usage of the **!!** 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



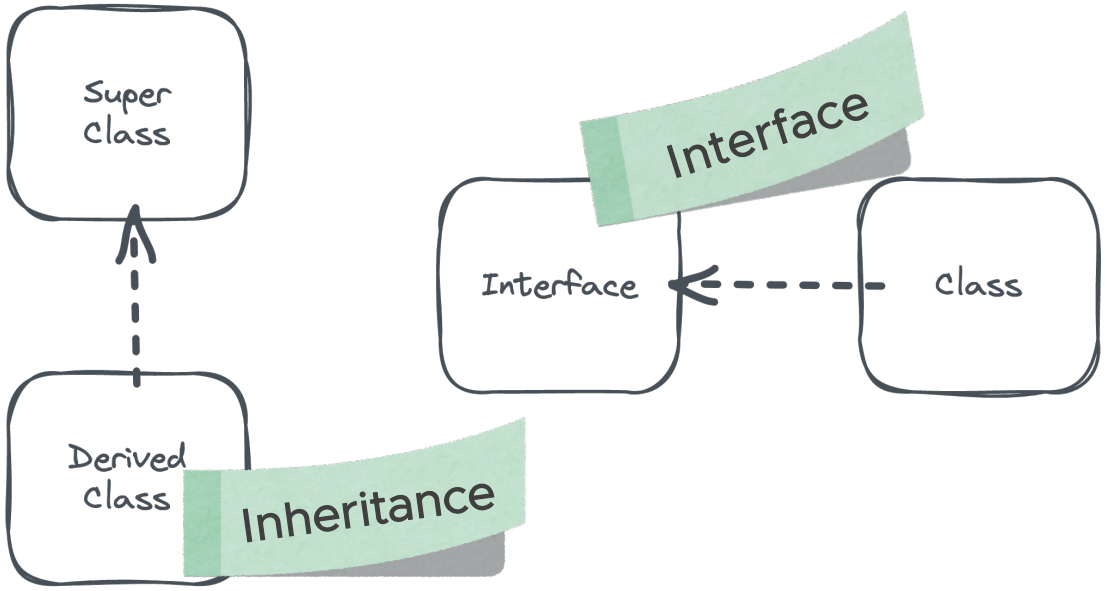
Java

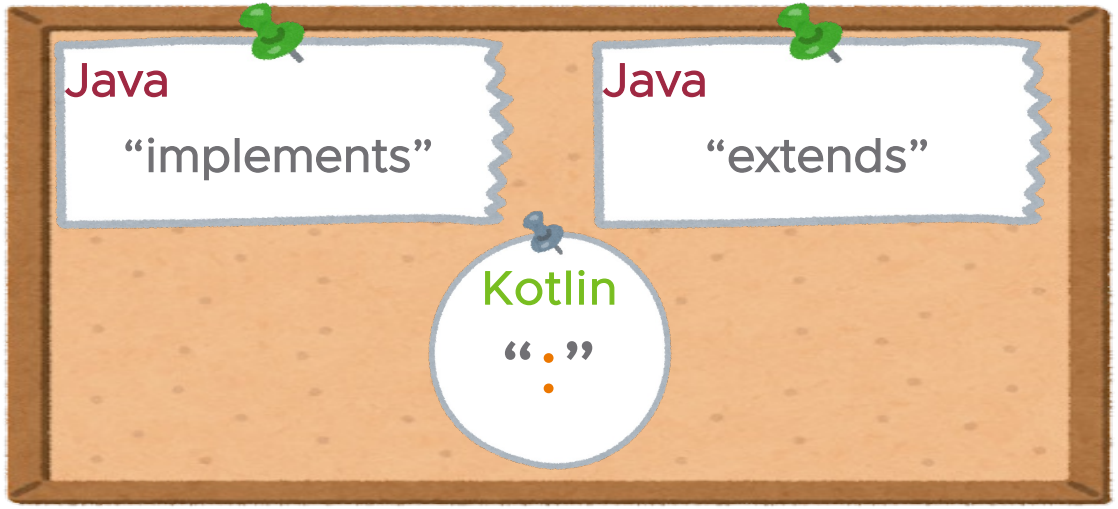
```java
public interface JavaBookRepository extends JpaRepository<Book, Long> {
}
```

Kotlin

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

Kotlin Basics: Interfaces and Inheritance





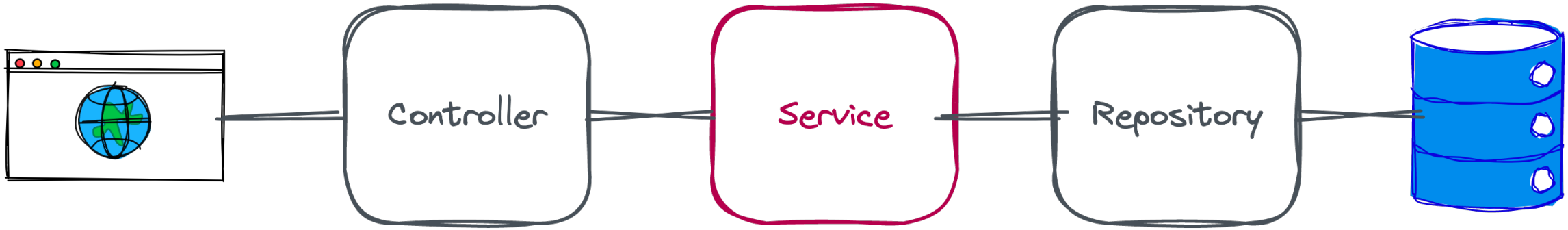
Interfaces and Inheritance,
that's all taken care of by colon “:”.

```
abstract class MySuperClass
interface MyInterface

class MyDerivedClass : MySuperClass(), MyInterface
```

When defining “inheritance”,
initialize by specifying the **primary constructor**
of the super class.

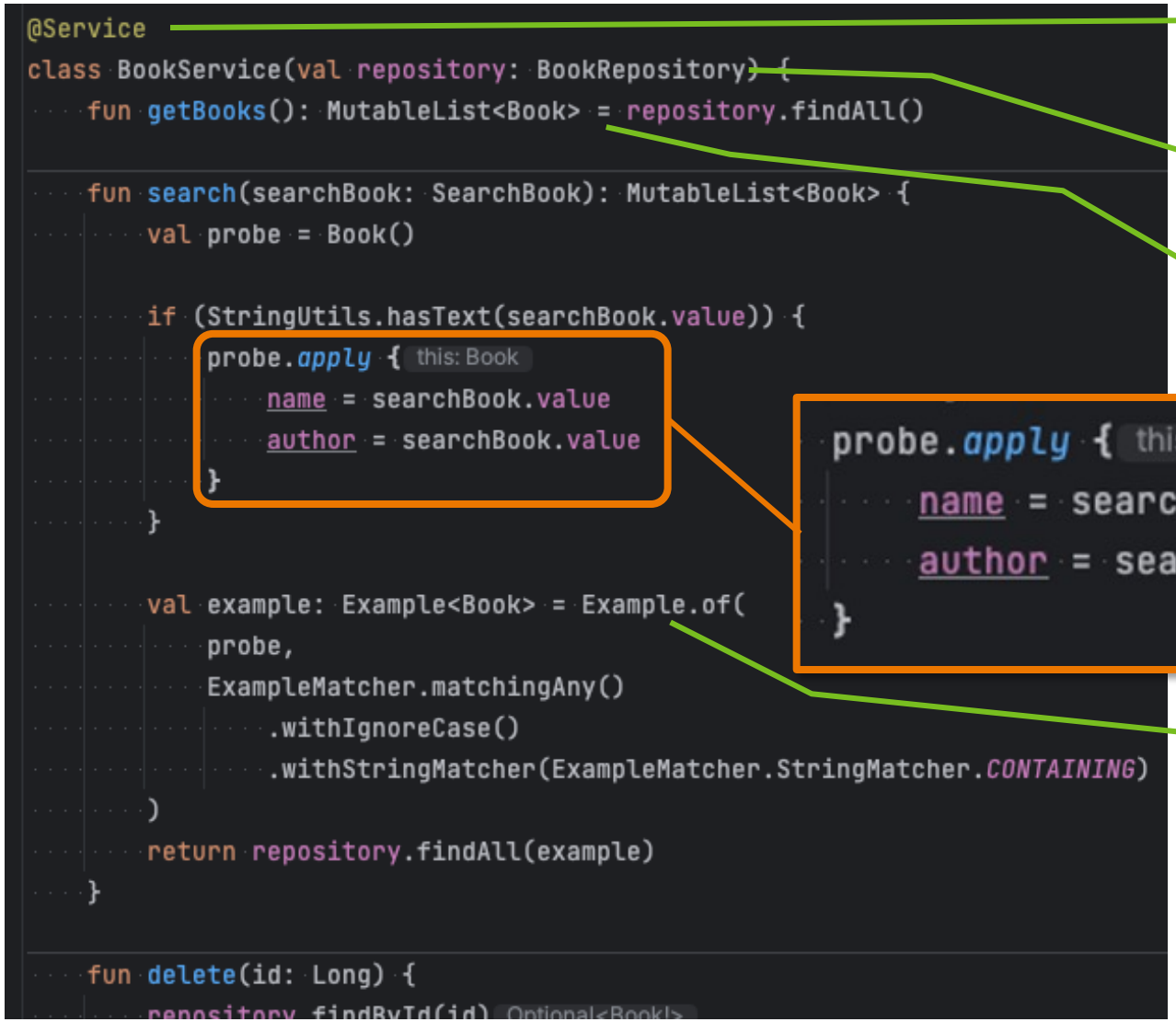
No Need to use the keywords "extends" and "implements" interchangeably like Java.

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 Annotation

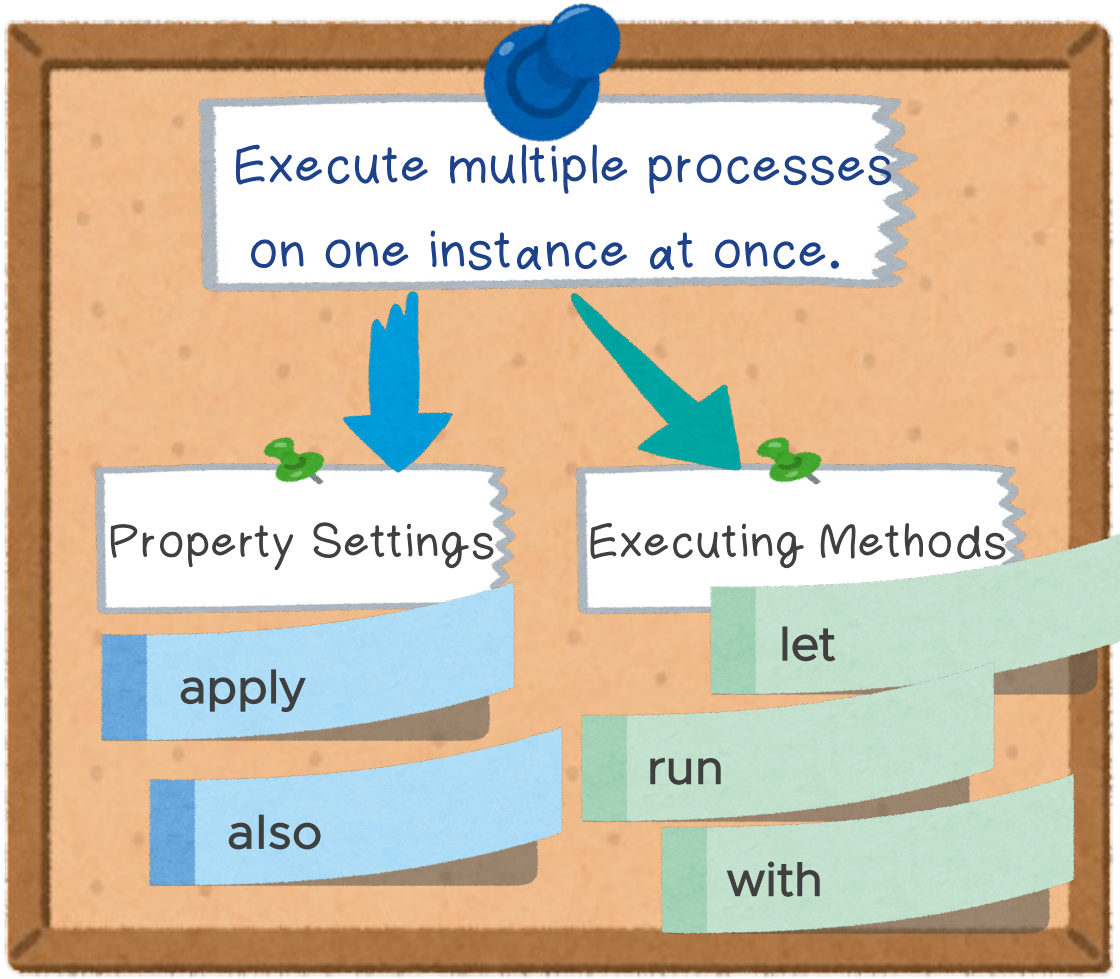
✅ Primary Constructor

✅ Function as Expression

✅ Scope Functions

✅ Spring Data JPA
Query by Example

Kotlin Basics: Scope Functions



Scope Functions,

their purpose is to execute a block of code within the context of an object.

	this	it
return receiver	apply	also
return result	run / with	let

When you call such a function on an object with a lambda expression provided, it forms a temporary scope.

In this scope, you can access the object without its name.

The backlogged tasks that still remains

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

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

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

✅ Nothing

```
@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, Unit, Any

Any

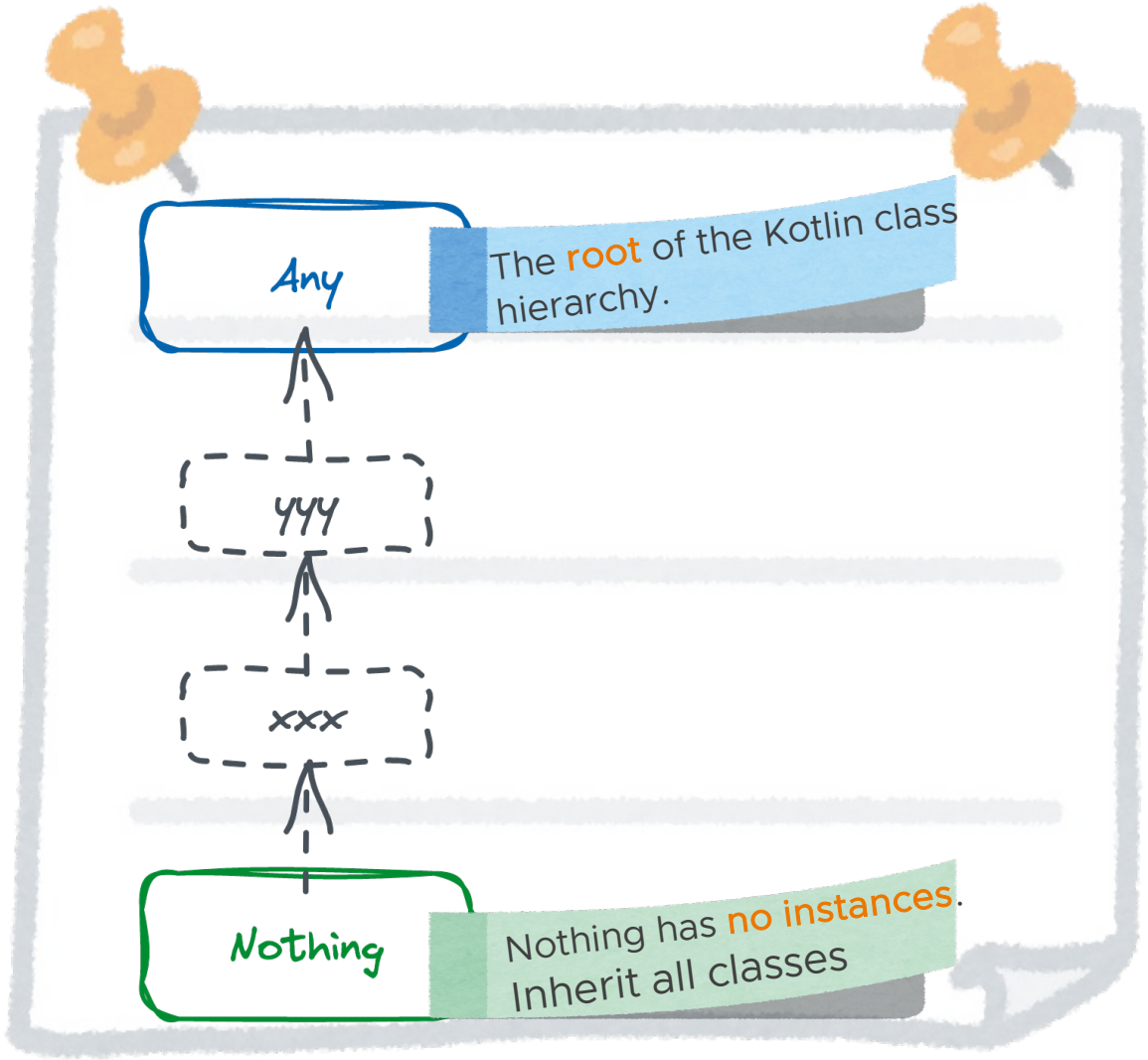
- Any is by default the superclass of all the classes
- This is equal to *Object* class in Java

Unit

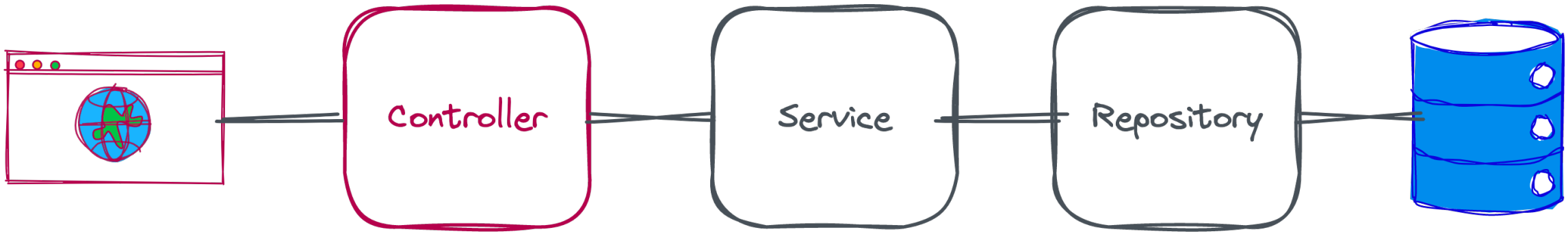
- Unit class in equal to void type in Java
- The superclass of Unit is *Any*

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.

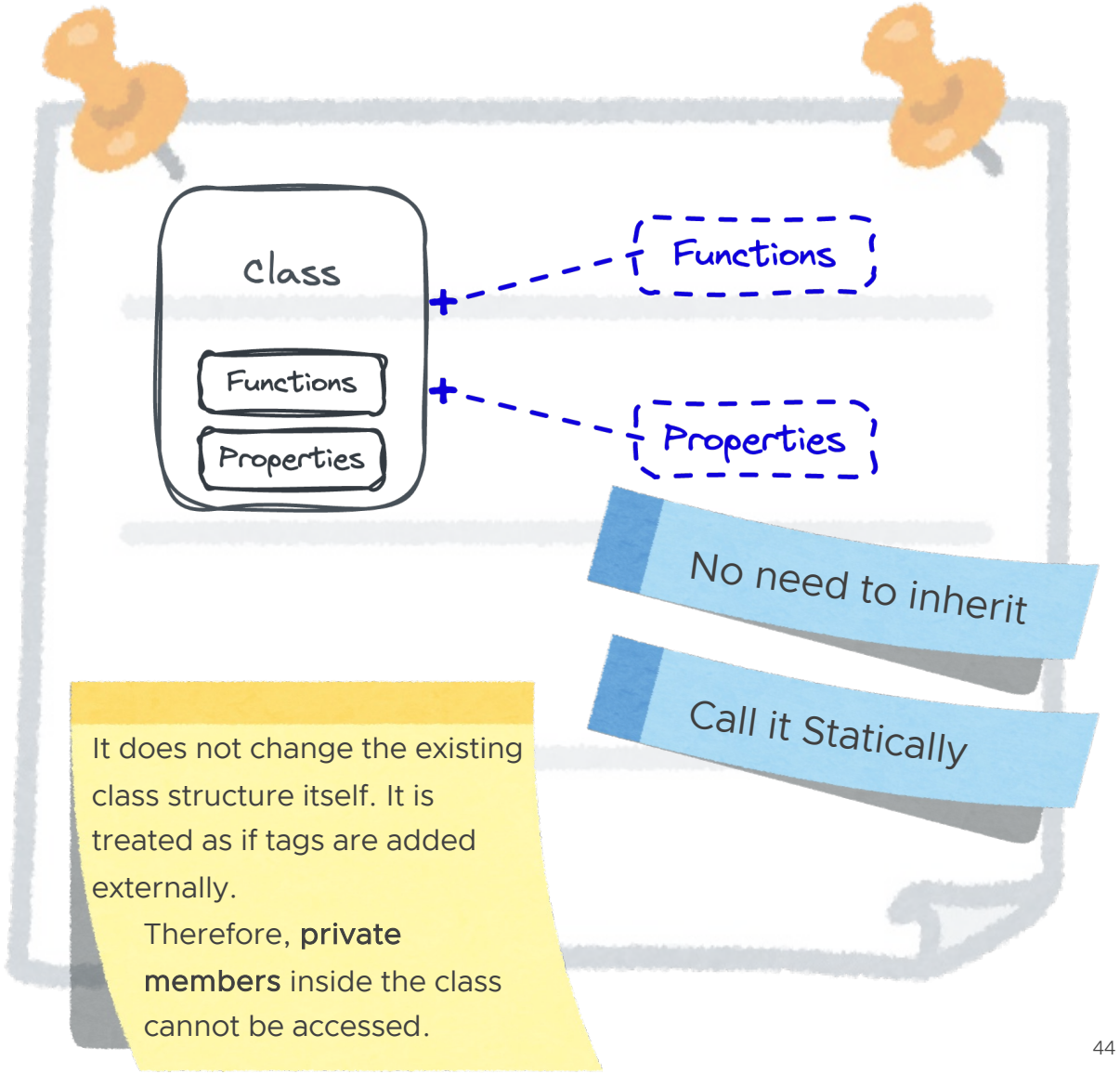
```
[0] fun Int.square(): Int = this * this
[1] println("10 * 10 = ${10.square()}")
10 * 10 = 100
```

- Extended functions are used **statically**

Extension properties

Like functions, properties can be added to existing classes.

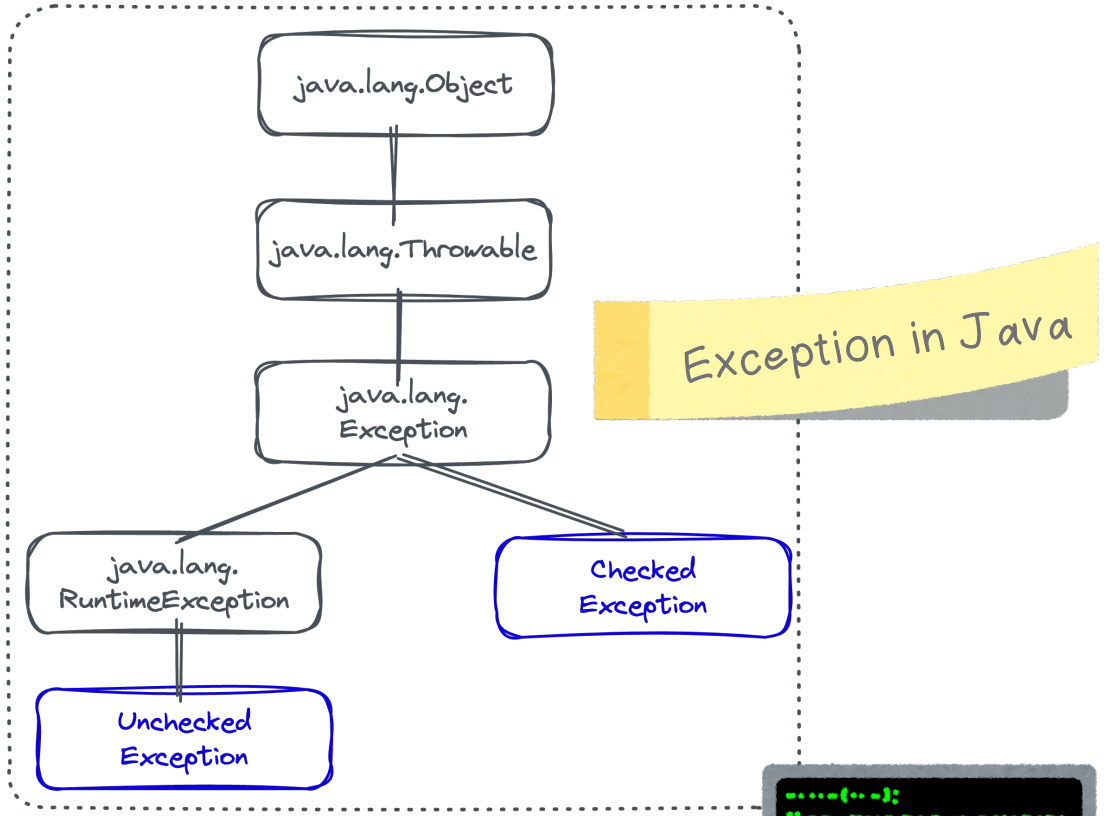
```
[0] val String.size: Int
    get() = this.length
[1] println("This is an Extension property".size)
29
```



Now, you can run the “Kotlin”ized 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

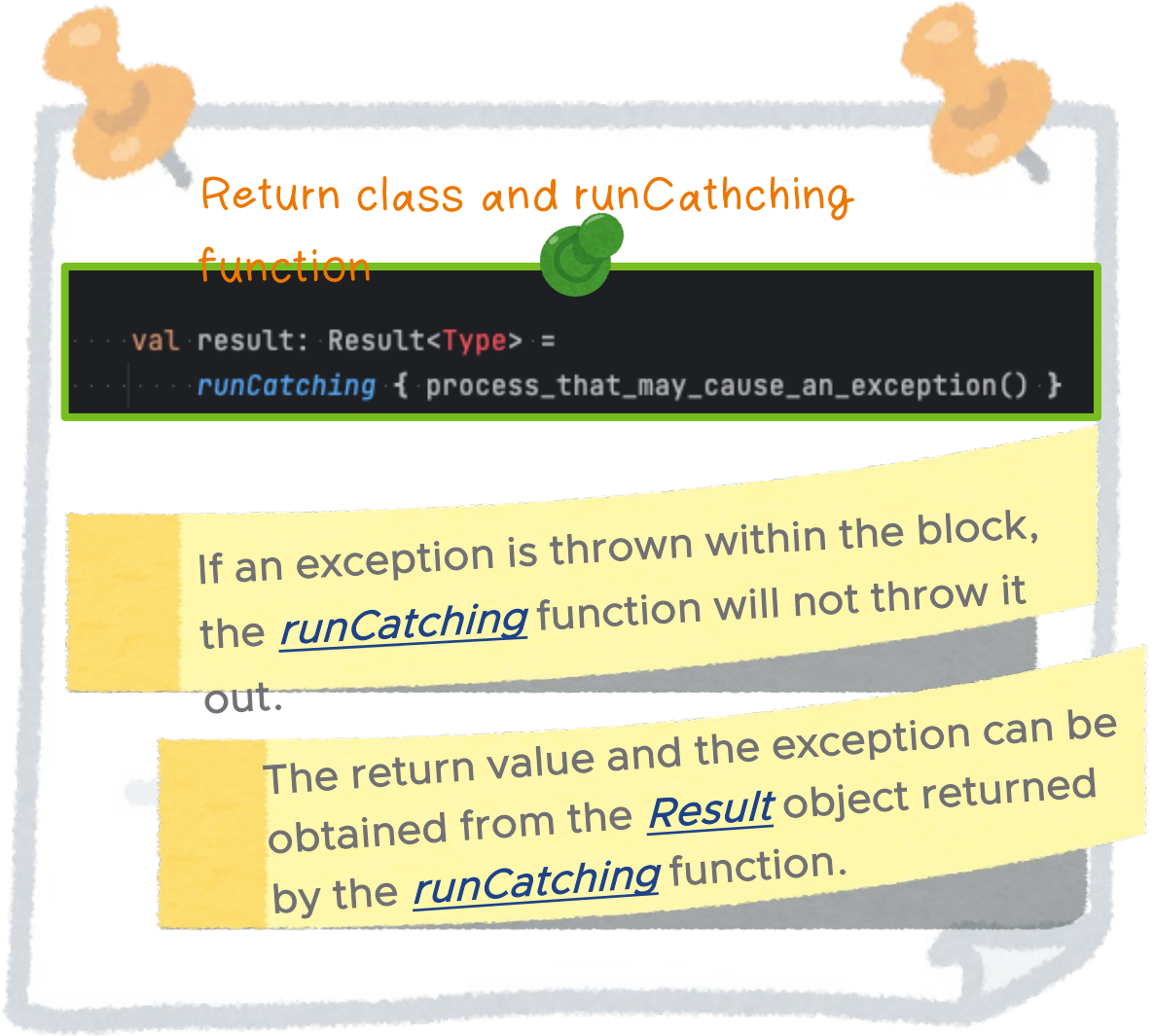
try-catch

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

runCatching

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

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

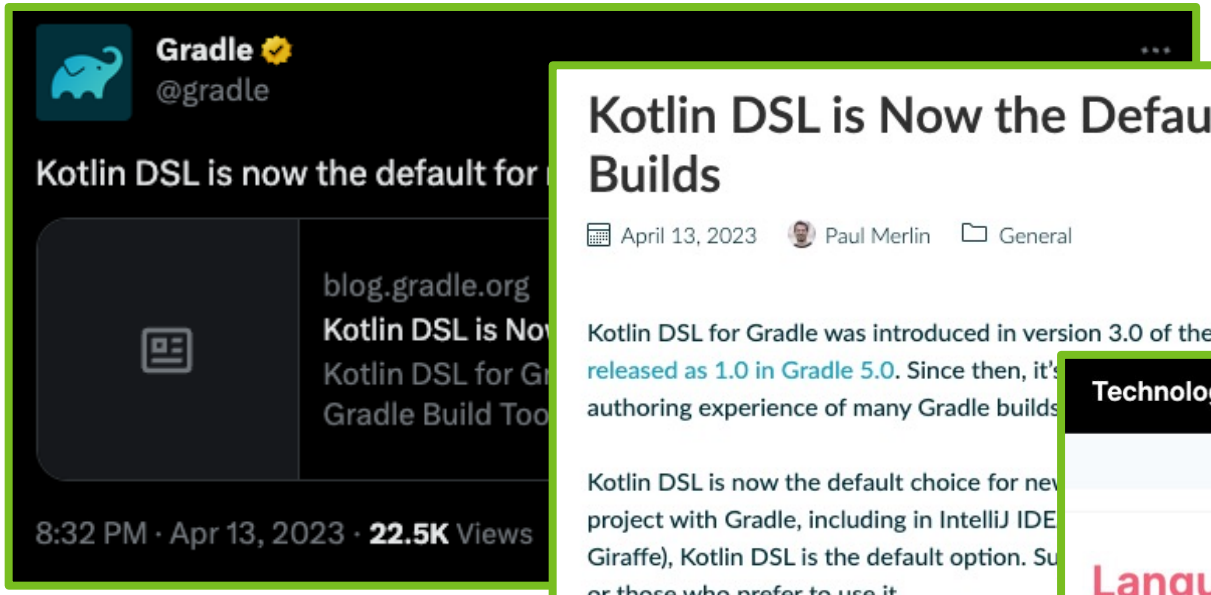
The *runCatching* function executes the given block and returns an object of type *Result*.

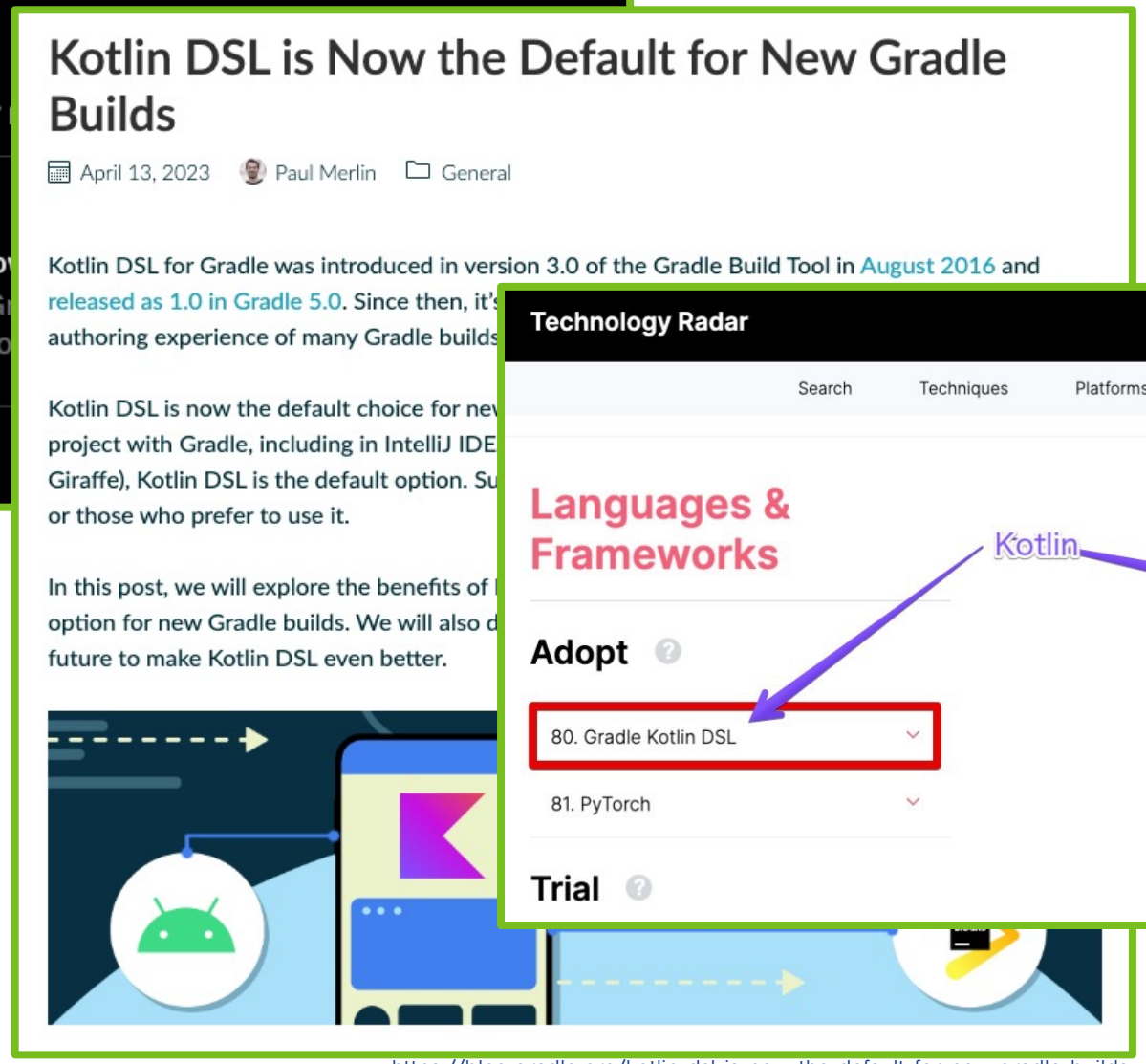


This time is the last. And just one more thing…

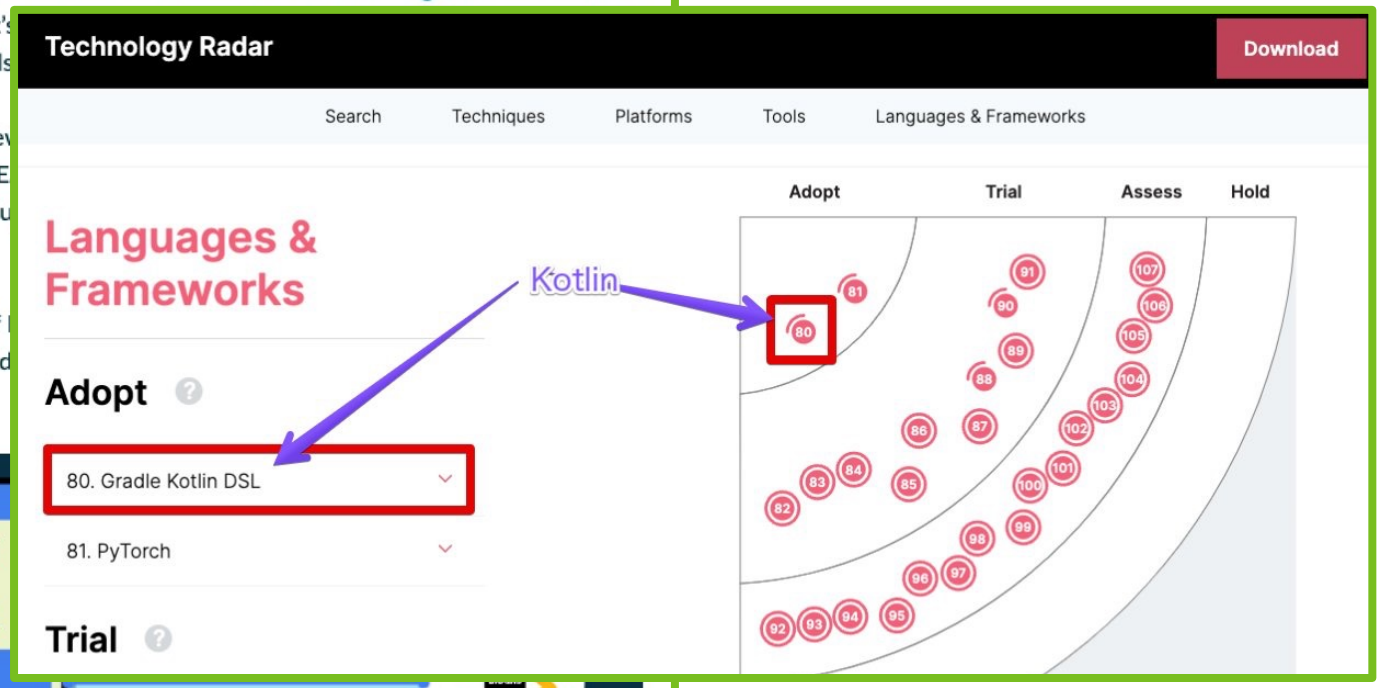
If you're interested in Kotlin,
you may have seen that story...

Gradle Kotlin DSL





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



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

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

Simple 5 steps

Step 1: Replace ' with "

build.gradle

```
group = 'com.example'
version = '0.0.1-SNAPSHOT'
```

build.gradle.kts

```
group = "com.example"
version = "0.0.1-SNAPSHOT"
```

You could use an editor to do a bulk replace.

Step2: Add ()

build.gradle

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-web'
```

build.gradle.kts

```
dependencies {
  implementation("org.springframework.boot:spring-boot-starter-web")
```

Target the following locations

- *id* in plugins block
- dependencies block
  - *implementation*, *runtimeOnly*, etc...

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

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

Step5: 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

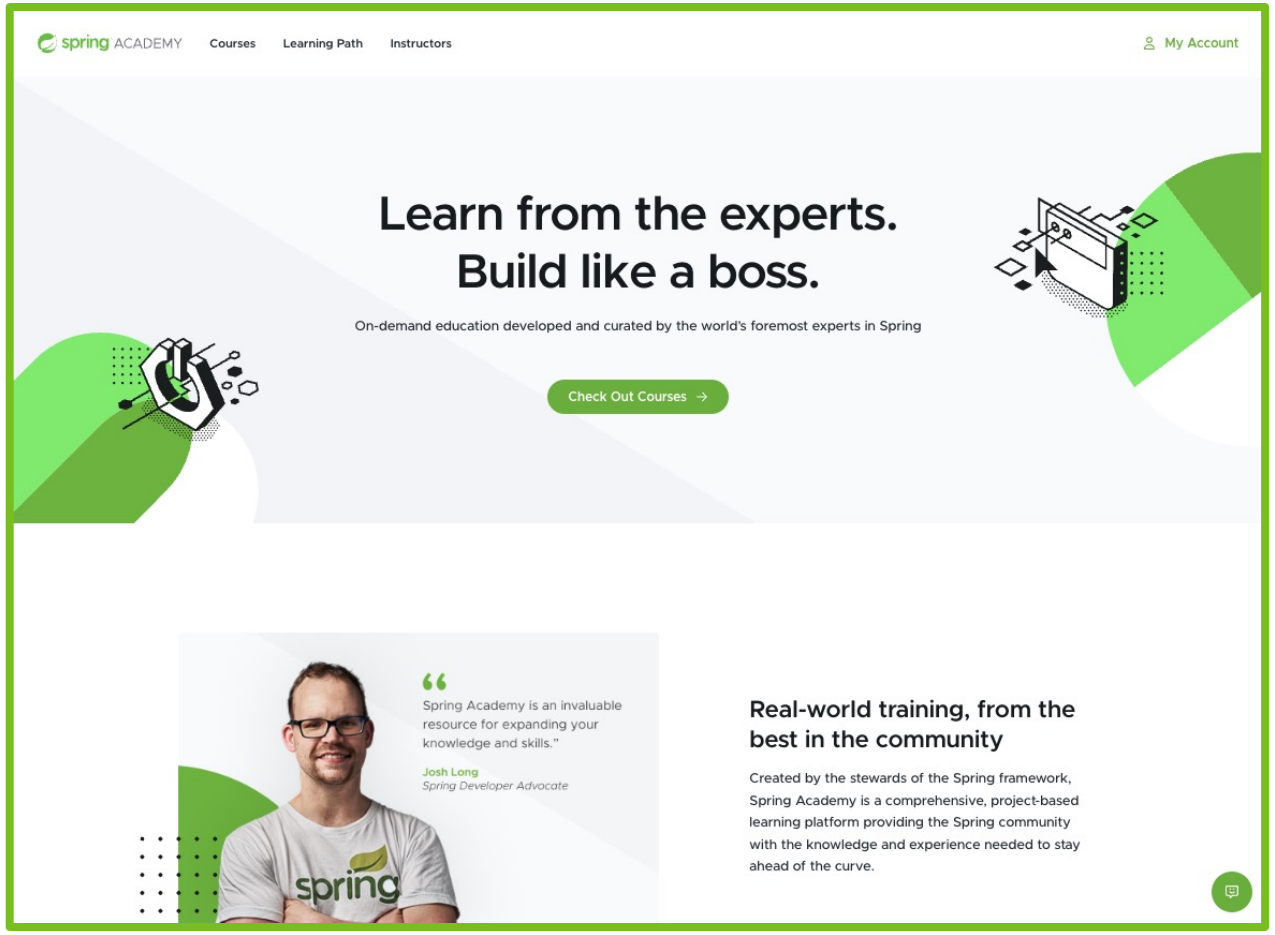
```
tasks.create<Delete>("clean") {
    delete(rootProject.buildDir)
}
```

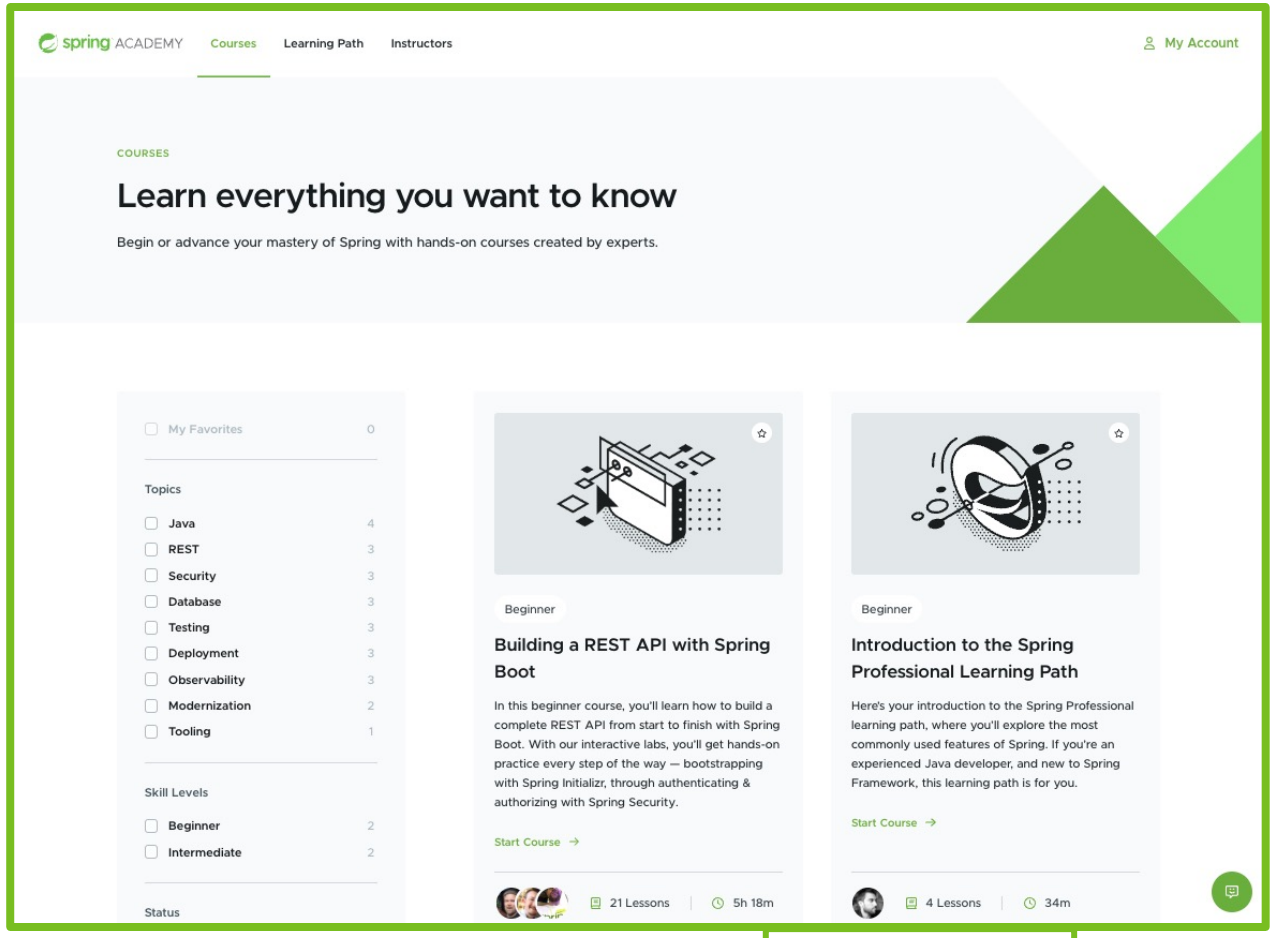
(Step6): Accumulate tips as you fix it

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

Key Takeaways

Spring Academy





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

Thank You
شكرا لك
Merci