

Building Restful Services Using Spring Boot and Kotlin

About Me

- Dilip
- Building Software's since 2008
- Teaching in **UDEMY** Since 2016

Why this course on Kotlin ?

Adoption [\[edit \]](#)

In 2018, Kotlin was the fastest growing language on GitHub with 2.6 times more developers compared to 2017.^[50] It is the fourth most loved programming language according to the 2020 Stack Overflow Developer Survey.^[51]

Kotlin was also awarded the O'Reilly Open Source Software Conference Breakout Award for 2019.^[52]

Many companies/organizations have used Kotlin for backend development:

- • Google^[53]
- Norwegian Tax Administration^[54]
- Gradle^[55]
- • Amazon^[56]
- Square^[57]
- JetBrains^[58]
- Flux^[59]
- Allegro^[60]
- OLX^[61]
- Shazam^[62]
- • Pivotal^[63]
- Rocket Travel^[64]
- Meshcloud^[65]
- Zalando^[66]

Some companies/organizations have used Kotlin for web development:

- JetBrains^[67]
- Data2viz^[68]
- Fritz2^[69]
- Barclay's Bank^[70]

A number of companies have publicly stated they were using Kotlin:

- DripStat^[71]
- Basecamp^[72]
- Pinterest^[73]
- Coursera^[74]
- • Netflix^[75]
- • Uber^[76]
- Square^[77]
- Trello^[78]
- Duolingo^[79]
- Corda, a distributed ledger developed by a consortium of well-known banks (such as [Goldman Sachs](#), [Wells Fargo](#), [J.P. Morgan](#), [Deutsche Bank](#), [UBS](#), [HSBC](#), [BNP Paribas](#), [Société Générale](#)), has over 90% Kotlin code in its codebase.^[80]

What's Covered ?

- Kotlin Introduction
- Code and Explore Kotlin Fundamentals
- Kotlin and SpringBoot Integration
- Build a RestFul Service using Kotlin and SpringBoot
- Unit/Integration tests using JUnit5 and Kotlin

Targeted Audience

- Experienced Java Developers
- Any Java Developer who is interested in learning Kotlin can enroll in this course
- Any Java/Kotlin Developer who is interested in building applications using SpringBoot can enroll in this course

Source Code

Thank You!

Prerequisites

- Java 17 (Java 11 or Higher is needed)
- Prior Java Experience is a must
- Prior Spring Framework Experience is a must
- Experience Writing JUnit tests
- **IntelliJ** or any other IDE

Kotlin Introduction

What is Kotlin?

- Kotlin is a modern object oriented and functional programming language
- Kotlin is a statically typed Programming language like Java
 - All the types are resolved at compile time
- This is free and open source language licensed under Apache 2.0

Who uses Kotlin ?

Adoption [\[edit \]](#)

In 2018, Kotlin was the fastest growing language on GitHub with 2.6 times more developers compared to 2017.^[50] It is the fourth most loved programming language according to the 2020 Stack Overflow Developer Survey.^[51]

Kotlin was also awarded the O'Reilly Open Source Software Conference Breakout Award for 2019.^[52]

Many companies/organizations have used Kotlin for backend development:

- Google^[53]
- Norwegian Tax Administration^[54]
- Gradle^[55]
- Amazon^[56]
- Square^[57]
- JetBrains^[58]
- Flux^[59]
- Allegro^[60]
- OLX^[61]
- Shazam^[62]
- Pivotal^[63]
- Rocket Travel^[64]
- Meshcloud^[65]
- Zalando^[66]

Some companies/organizations have used Kotlin for web development:

- JetBrains^[67]
- Data2viz^[68]
- Fritz2^[69]
- Barclay's Bank^[70]

A number of companies have publicly stated they were using Kotlin:

- DripStat^[71]
- Basecamp^[72]
- Pinterest^[73]
- Coursera^[74]
- Netflix^[75]
- Uber^[76]
- Square^[77]
- Trello^[78]
- Duolingo^[79]
- Corda, a distributed ledger developed by a consortium of well-known banks (such as [Goldman Sachs](#), [Wells Fargo](#), [J.P. Morgan](#), [Deutsche Bank](#), [UBS](#), [HSBC](#), [BNP Paribas](#), [Société Générale](#)), has over 90% Kotlin code in its codebase.^[80]

Why Kotlin ?

- Kotlin is an expressive language and it has a concise syntax
 - Code readability and maintainability
- Kotlin is a safe language which prevents un-necessary errors
 - Prevents **NullPointerException** using the Nullable and Non-Nullable types
- Interoperable with Java
 - Kotlin and Java works together very well

What kind of applications can we build with Kotlin ?

- Build Server Side Applications
 - Web Applications
 - RestFul Services
 - Messaging Applications
 - Build any kind of applications that we can build using Java
- Widely used in the Android

Popular Framework Support

Frameworks for server-side development with Kotlin



- [Spring](#) [↗] makes use of Kotlin's language features to offer more concise APIs [↗], starting with version 5.0. The online project generator [↗] allows you to quickly generate a new project in Kotlin.
- [Vert.x](#) [↗], a framework for building reactive Web applications on the JVM, offers dedicated support [↗] for Kotlin, including full documentation [↗].
- [Ktor](#) [↗] is a framework built by JetBrains for creating Web applications in Kotlin, making use of coroutines for high scalability and offering an easy-to-use and idiomatic API.
- [kotlinx.html](#) [↗] is a DSL that can be used to build HTML in a Web application. It serves as an alternative to traditional templating systems such as JSP and FreeMarker.

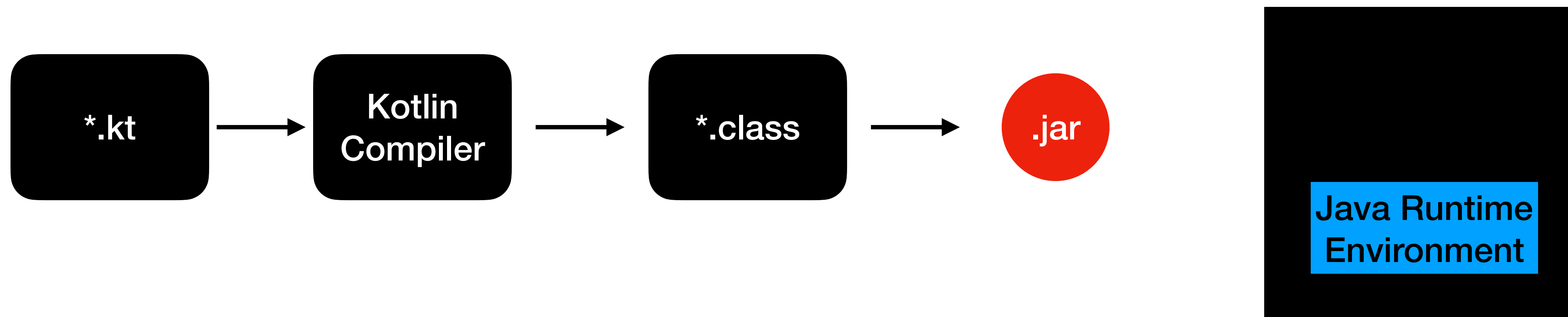


- [Micronaut](#) [↗] is a modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications. It comes with a lot of built-in, handy features.
- [http4k](#) [↗] is the functional toolkit with a tiny footprint for Kotlin HTTP applications, written in pure Kotlin. The library is based on the "Your Server as a Function" paper from Twitter and represents modeling both HTTP Servers and Clients as simple Kotlin functions that can be composed together.
- [Javalin](#) [↗] is a very lightweight web framework for Kotlin and Java which supports WebSockets, HTTP2 and async requests.
- The available options for persistence include direct JDBC access, JPA, as well as using NoSQL databases through their Java drivers. For JPA, the kotlin-jpa compiler plugin adapts Kotlin-compiled classes to the requirements of the framework.

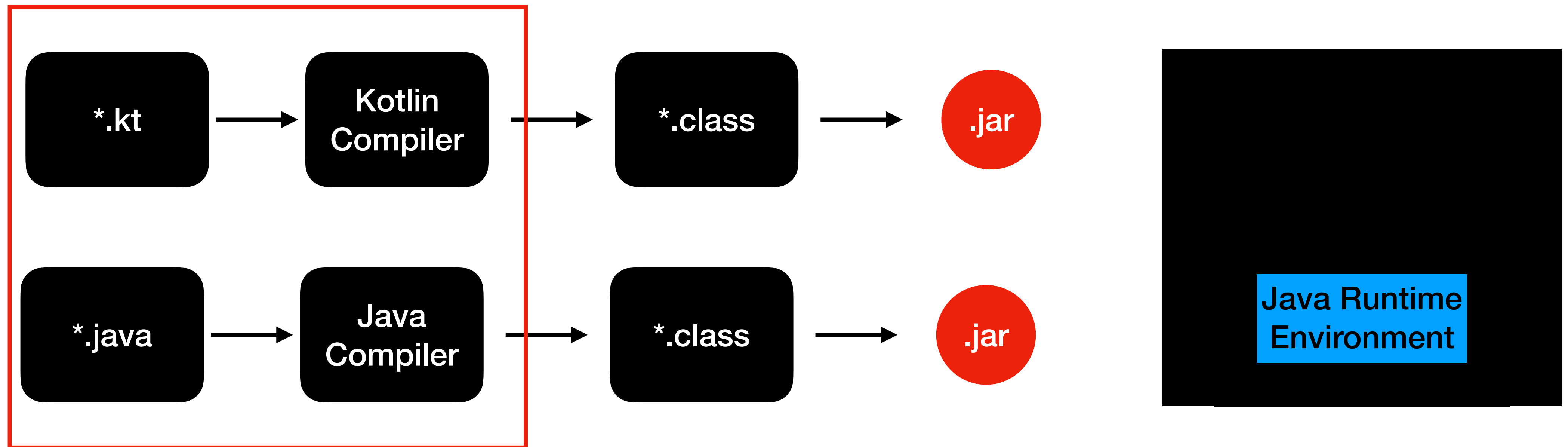
Kotlin Community

**How does Kotlin work with the
JVM ?**

Kotlin Build Process



Kotlin/Java Build Process



val & var

- Any variable in Kotlin should be declared as **val** or **var**
- **val**
 - Variables declared with **val** are immutable variables

```
val name : String = "Dilip"
```

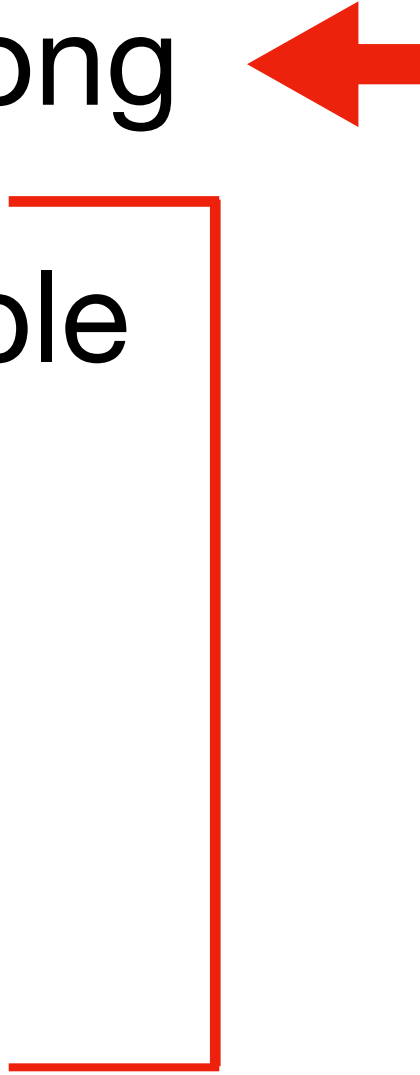
↑ ↑
Name of the Type of the
Variable Variable

- **var**
 - Variables declared with **var** are mutable variables

```
var age : Int = 33
```

↑ ↑
Name of the Type of the
Variable Variable

Types in Kotlin

- In Kotlin, there is no distinction between primitives and wrapper types
 - All numbers in Kotlin are represented as types
 - Integer Types - Byte, Short, Int, Long ←
 - Floating-Point Types - Float, Double
 - Character Type - Char
 - Boolean Type - Boolean
- 

String and its related Operations

- String Interpolation

```
val course = "Kotlin Spring"  
println("course : $course and the length of the course is ${course.length}")
```



- Multiline Strings using TripleQuotes

```
val multiLine1 = """ ←  
    ABC  
    DEF  
    """.trimIndent()
```

if-else

- if-else is an expression in kotlin
- Expression always evaluate to a result

```
val result = if(name.length == 4){  
    println("Name is Four Characters")  
    name ←  
}else{  
    println("Name is Not Four Characters")  
    name ←  
}
```

when

- when block allows us to write concise and expressive code when dealing with multiple conditions

```
val medal1 = when(position){  
    1 -> "GOLD"  
    2 -> "SILVER"  
    3 -> {  
        println("Inside position 3")  
        "BRONZE"  
    }  
    else -> "NO MEDAL"  
}
```

Ranges

```
val range = 1..10
for (i in range){
    println(i)
}
```

Creates a Range of 10 values

```
val reverseRange = 10 downTo 1
for (i in reverseRange){
    println(i)
}
for (i in reverseRange step 2){
    println(i)
}
```

Progression of Values in decreasing order

Skip values in the iteration

while


```
fun exploreWhile() {  
    var x = 1  
    ➔ while(x < 5){  
        println("Value of x is $x")  
        x++  
    }  
}
```

doWhile

```
fun exploreDoWhile() {  
    var i = 0  
    ➔ do {  
        println("Inside do while : $i")  
        i++  
    }while (i < 5)  
}
```



break

```
for(i in 1..5){  
    println("i is $i ")  
    if(i==3) break  
}
```




label

```
fun label() {  
    loop@ for(i in 1..5){  
        println("i in label $i: ")  
        innerLoop@ for (j in 1..10){  
            //if(j==2) break@innerLoop  
            if(j==2) break@loop  
        }  
    }  
}
```



return

```
listOf(1,2,3,4,5).forEach each@{  
    //if(it==3) return@forEach  
    if(it==3) return@each  
}
```



Functions in Kotlin


Function

- Functions are primarily used in Object Oriented Language to express some behavior or logic

Function Name
↓
→ `fun printHello() {`
 `println("Hello!")` } Function Body
}

Functions with No return value

- Functions with no return value are represented as **Unit** in Kotlin



```
fun printHello() : Unit {  
    println("Hello!")  
}
```

- Defining Unit is redundant and it can be ignored

Function with parameters and return value

- Functions with expression body can be simplified like below

```
fun addition(x: Int, y : Int) : Int {  
    return x+y  
}
```

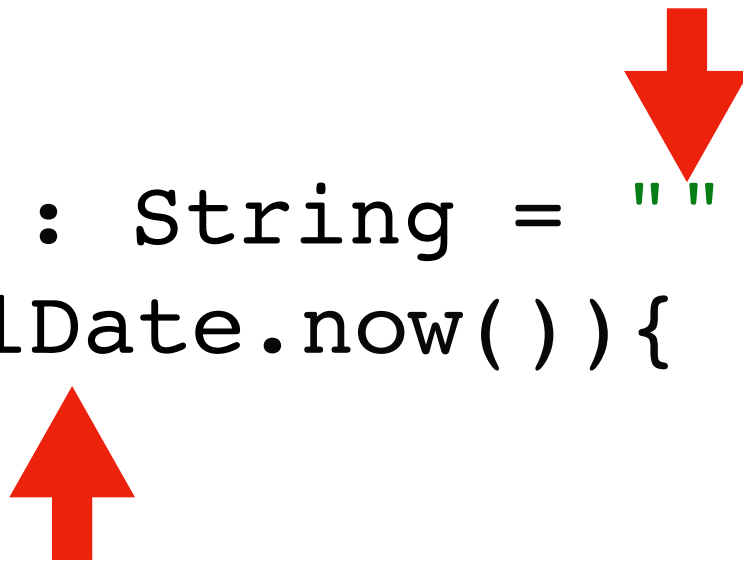
```
fun addition(x: Int, y : Int) = x+y
```

Default Value Parameters & Named Arguments

Default Value Parameters

- This provides a default value to a function parameter when its not passed by the caller


```
fun printPersonDetails(name : String, email : String = "",  
                        dob : LocalDate = LocalDate.now()) {  
    println("Name is $name and the email is $email and the dob is $dob")  
}
```



Named Arguments


- The caller can invoke the function by using the variable name

```
fun printPersonDetails(name : String, email : String = "",  
                        dob : LocalDate = LocalDate.now()) {  
    println("Name is $name and the email is $email and the dob is $dob")  
}
```



- Caller can invoke the function using the name of the function arguments, in no particular order

```
printPersonDetails(dob = LocalDate.parse("2000-01-01") , name = "Dilip", email =  
"abc@gmail.com")
```



Top Level Functions & Properties

Top Level Functions

- Functions that does not belong to a class are **top-level** functions
 - In Java , functions can only be part of class
 - In Java applications, you can find classes that just has some static methods which holds some common logic that can be used across the app
- Kotlin avoids this by using top level functions that can be part of a **Kotlin file** not a class

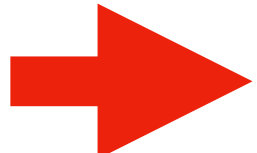
Top Level Properties

- In Kotlin, properties that does not belong to class are called top-level properties
 - In Java, you can only define properties in a class or an interface
 - Most common use case in a Java application is you may have to be define static constants in a class file that can be used across the app
- Kotlin avoids these by allowing us to create properties that can be part of a **Kotlin file** not a class

Class

Class in Object Oriented Programming

- Class in object oriented programming is fundamentally the blueprint for creating objects



```
class Person {  
    → fun action() {  
        println("Person Walks")  
    }  
}
```

Instance of the class:

```
val person = Person() // new keyword is not needed
```



```
person.action()
```

Constructors in Kotlin

Constructors in Kotlin

- Constructors is a concept in object oriented programming through which we can create an Object with initial values

```
class Person(val name: String,
              val age: Int) {
    fun action() {
        println("Person Walks")
    }
}
```

1 Primary Constructor

2

Instance of the class:



```
val person = Person("Alex", 25)
```



Secondary Constructor

- This is an alternative way of defining constructors

```
class Item(){  
    var name : String = ""  
    → constructor(_name : String) : this(){  
        name = _name  
    }  
}
```



- **constructor** keyword
- **this()** call to the actual class is mandatory

Recommended approach for constructors

- Use Primary Constructors whenever possible
- Use default values for overloaded constructor scenario

```
class Person(val name: String = "",  
             val age: Int = 0) {  
  
    fun action() {  
        println("Person Walks")  
    }  
  
}
```

Use Secondary Constructors only necessary

Initializer code using init block

- init code block can be used to run some initialization logic during the instance creation

```
init {  
  
    println("Inside Init Block")  
  
}
```

data class

- Classes just holds the data can be categorized as data classes
 - DTOs, domain classes and value object classes fall under this category
 - In Java, these type of classes are also Java Beans

➔ `data class Course(
 val id: Int,
 val name: String,
 val author: String
)`

- Automatically generates the equals(), hashCode() and toString() methods

Usage & Benefits of data class

- Data classes are primarily for classes that's going to behave as a data container
- It autogenerates a lot of functionalities for you when you add the **data** modifier to the class
- Its pretty easy to create a clone of the object using the **copy()** function

When to use Custom Getter/Setter ?

Use it when you have the need to implement the custom logic for setting or retrieving the properties.

Inheritance

- Inheritance is supported in Kotlin
- Kotlin Concepts:
 - **Any** is the superclass for any class in Kotlin
 - **Object** Class in Java
 - All the classes in Kotlin are **final**
 - Extending classes are not allowed by default

Inheritance - Extending Classes

- Kotlin allows inheritance when **open** modifier is used

➡ `open class User(val name: String) {`
 `open fun login() {`
 `println("Inside user login")`
 `}`
`}`

- Subclass extending the **User** Class

`class Student(name: String) : User(name)` ⬅

Inheritance - Overriding Functions

- Mark the function with **open** modifier

```
open class User(val name: String) {  
    ➡ open fun login() {  
        println("Inside user login")  
    }  
}  
  
class Student(name: String): User(name, age) {  
    ➡ override fun login() {  
        ➡ super.login()  
        println("Inside Student login")  
    }  
}
```

Inheritance - Overriding Variables

- Similar to functions using the **open** and **override** keywords

```
open class User(val name: String) {
```

```
➔ open val isLoggedIn : Boolean = true  
}
```

```
class Student(name: String) : User(name, age) {
```

```
➔ override val isLoggedIn : Boolean = true  
}
```

object keyword

- This keyword allows us to create a class and an instance of the class at the same time
- Equivalent to a **singleton** pattern in java

➡ `object Authenticate {`

```
    fun authenticate(userName : String, password: String){  
        println("User Authenticated for userName : $userName")  
    }  
}
```

Usage:

```
fun main() {
```

➡ `Authenticate.authenticate("Dilip", "abc")`

```
}
```


- Limitations
- You cannot inject constructor arguments to object classes

companion object

- Kotlin does not have the support for the **static** keyword
- **companion object** can be used to introduce static functionalities that are tied to the class.
- Using object inside the class requires you to use the **companion** keyword
- You can have variables and functions
- Usage

`Student.country()`

```
class Student(  
    name: String,  
    override val age: Int = 0  
) : User(name, age) {
```



```
    companion object {  
        const val noOfEnrolledCourses = 10  
  
        fun country(): String {  
            return "USA"  
        }  
    }  
}
```

Interface

- Interfaces in oops defines the contract which has some abstract methods
- The classes that implements the interface needs to implement it.
 - This is similar to Java
- Interfaces can have abstract and non-abstract methods in it
- It cannot contain any state

Interface

- Interface with abstract method

➡ `interface` CourseRepository {

 `fun` getById(id: Int): Course

}



➡ `class` SqlCourseRepository : CourseRepository {

 ➡ `override fun` getById(id: Int): Course {
 `return` Course(id = id,
 "Kafka For Developers using Spring Boot",
 "Dilip Sundarraaj")
 }

}

- Interface with abstract/non-abstract method

`interface` CourseRepository {
 `fun` getById(id: Int): Course

➡ `fun` save(course: Course): Int {
 `println`("course : \$course")
 `return` course.id
}

`class` SqlCourseRepository : CourseRepository {

 `override fun` getById(id: Int): Course {
 `return` Course(id = id,
 "Kafka For Developers using Spring Boot",
 "Dilip Sundarraaj")
 }

➡ `override fun` save(course: Course): Int {
 `println`("course in SqlCourseRepository : \$course")
 `return` course.id
}

}

Visibility Modifiers in Kotlin

Visibility Modifiers in Kotlin

- There are four visibility modifiers in Kotlin :
 - public, protected, private and internal
- public
 - This is the default access modifiers
- private
 - This marks the function or variable accessible only to that class
- protected
 - A protected member is visible in the class and subclass
- internal
 - This is new in Kotlin. Anything that's marked with internal is private to the module that's published using the Gradle or Maven

Type Checking & Casting

- Kotlin has some handy operators

- **is** operator

- Check a particular value is of a certain type

```
val name = "Dilip"
```

```
val result = name is String
```

true or false

- **as** operator

- Cast a value to a certain type

```
val name = "Dilip" as String
```

- If cast is not possible then it throws **java.lang.ClassCastException**

Nulls in Kotlin



Handling Nulls in Kotlin

- Kotlin handles nulls little differently compared to Java
- Kotlin has the concept of **Nullable** and **Non-Nullable** types
- These can be assigned to variables or properties

Nullable Type

- A variable or property can hold a null value
- How to define a Nullable Type?

```
val nameNullale : String? = null
```



Or

```
val nameNullale : String? = "Dilip"
```



Non Nullable Type

- A variable or property can hold only a non-null value
- How to define a Non-Nullable Type?



```
val nameNonNull : String = "Dilip"
```

Or

null value is not allowed

```
val nameNonNull = "Dilip"
```



By Default , types are Non-Nullable

Dealing with Nulls

Safe Call Operator

- Use the safe call operator to invoke functions safely on it


```
val length = nameNullable?.length
```



Elvis Operator

- Return a Default value if null


```
val length = nameNullable?.length ?: 0
```



Not Null Assertions

- Making sure the value is not null after some updates

```
val movie =  
    saveMovie(Movie(null,  
                    "Avengers"))  
  
println(movie.id!!)
```



Collections in Kotlin

Collections

- Kotlin re-uses the collections from Java
 - Kotlin does not have collections of their own
 - Kotlin added a lot of new features through extension functions
- Kotlin has two types of collections
 - Mutable Collections
 - Immutable Collections

Immutable Collection

- Collection is not modifiable once created

```
val names = listOf("Alex", "Ben", "Chloe")
```



Map

```
mapOf("dilip" to 33, "scooby" to 4)
```



Set

```
setOf("adam", "ben", "chloe")
```



Mutable Collection

- Modifying the data in the collection is allowed

```
val namesMutableList = mutableListOf("Alex", "Ben", "Chloe")
```



```
mutableMapOf("dilip" to 33, "scooby" to 4)
```

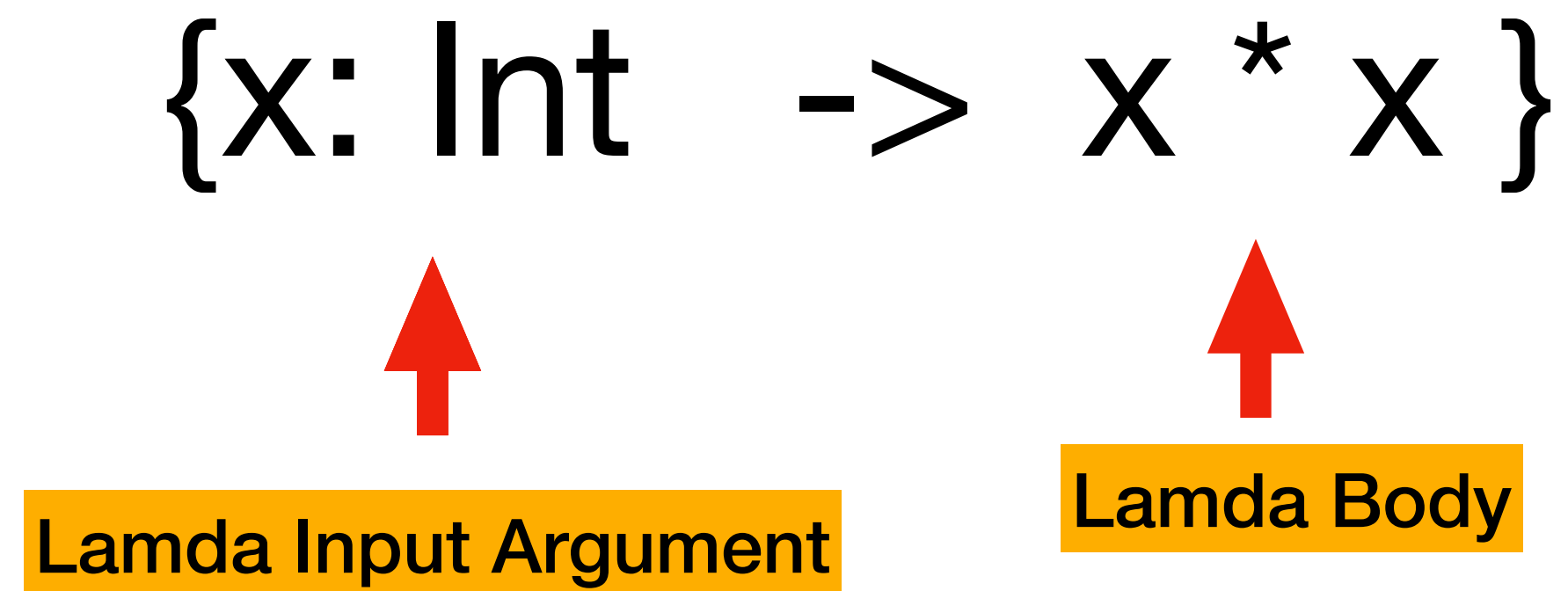


```
mutableSetOf("adam", "ben", "chloe")
```



What is a Lambda Expression ?

- Lambda Expressions are small piece of code that can be passed to other functions



Benefit of Lambda Expression

- You can assign the behavior to a variable

```
val add = { x : Int -> x+x }
```

```
fun add(x:Int) = x+x
```

- The advantage here is that , you can pass the lambda as an argument to other functions

```
listOf(1, 2, 3) ←  
  .forEach {  
    val result = add(it)  
    println(result)  
  }
```

2 4 6

Collections & Operations on it

- Using Collections are very common in any application development
- Performing some logic on each elements in the list
- Filter out some elements in the collection
- Collections has operators to perform operations on it

filter



- filter
 - This is used to filter out the elements from the collection

```
val numList = listOf(1, 2, 3, 4, 5, 6) ←  
  
val result = numList.filter {  
    it >= 5 ←  
} ↑
```

map

- This operator is fundamentally used to transfer the elements from one form to other form

```
val numList = listOf(1, 2, 3, 4, 5, 6)
```



```
val result = numList.map {  
    it.toDouble()   
}  

```

```
result : [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
```

flatMap

- This operators can be used if the collection has another collection
- Used to flatten the list of lists that is operating on and returns the result as a single list

map

```
val list = listOf(listOf(1,2,3),listOf(4,5,6) )  
val result = list.map { outerList ->   
    outerList.map {   
        it.toDouble()  
    }  
}  
println("result : $result")
```

result : [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]

flatMap

```
val list = listOf(listOf(1,2,3),listOf(4,5,6) )  
val result = list.flatMap { outerList ->  
    outerList.map {  
        it.toDouble()  
    }  
}  
println("result : $result")
```

result : [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]



Lazy Evaluation of Collections using Sequences

- This is an alternative API to work with collections
- The operations on the elements of the collection are evaluated lazily

Not a sequence

```
val namesList = listOf("alex", "ben", "chloe")  
  
    .filter { it.length >= 4 } // ["alex", "chloe"]  
  
    .map { it.uppercase() } // ["ALEX", "CHLOE"]
```

sequence

```
val namesListUsingSequence = listOf("alex", "ben", "chloe")  
    .asSequence()  
    .filter { it.length >= 4 } // "alex" "ben" "chloe"  
    .map { it.uppercase() } // "ALEX" "CHLOE"  
    .toList() "ALEX" "CHLOE"
```

Terminal Operator

Whats the benefit of using Sequences ?

- Sequences perform better when dealing with collections that are extremely big
 - Does not create intermediate collection for each operator
 - Sequences are lazy, does not apply the operations for all the elements in the collection

Arrays in Kotlin

- Arrays in Kotlin are represented using the `Array<T>` class
- Arrays in Kotlin are immutable
- How can we create an Array ?

```
val namesArray = arrayOf("alex", "ben", "chloe")
```

```
val emptyArray= emptyArray<String>()
```

Exceptions in Kotlin

- All Exception classes in Kotlin extends the Throwable class
- Exceptions are handled pretty much the standard way using the try-catch block
- Kotlin does not have checked exceptions

Checked Exceptions

Java

```
import java.io.File;
import java.io.FileInputStream;

public class Test {

    public static void main(String[] args) {

        File file = new File( pathname: "not_existing_file.txt");
        FileInputStream stream = new FileInputStream(file);

    }
}
```



Kotlin

```
val file = File("file.txt")
val stream = FileInputStream(file)
```

Scope Functions

Scope Functions

- These functions are there in Kotlin to execute a piece of code within the context of the object
- These functions form a temporary scope, that's why the name

Scope Functions in Action

With Scope Function

```
var nameNullable : String? = null  
  
nameNullable?.run {  
    printName(this)  
    println("Completed!")  
}
```

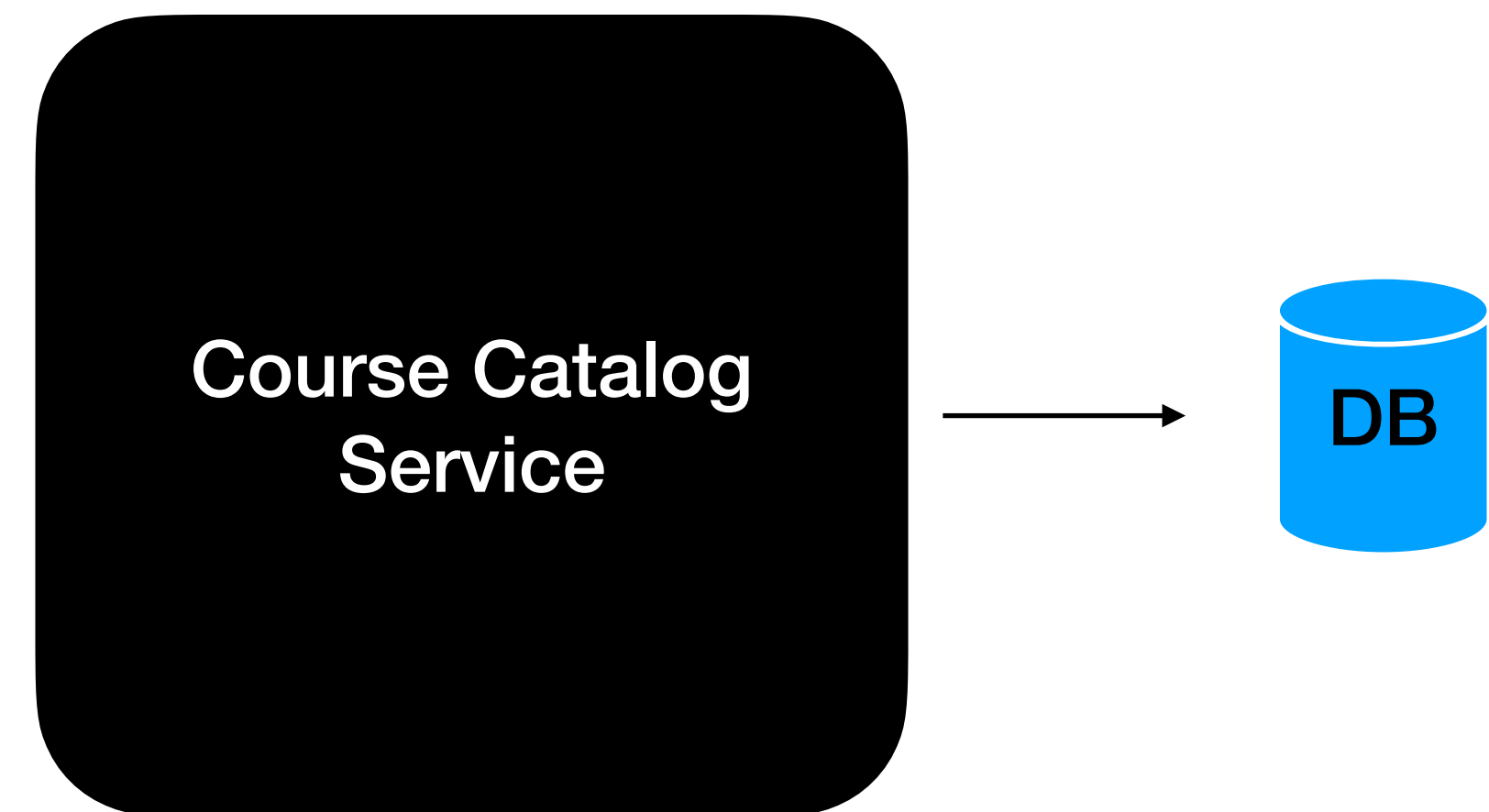
Scope Function accepts the lambda

Without Scope Function

```
if(nameNullable!=null){  
    printName(nameNullable)  
    println("Completed!")  
}
```


Overview of the Application

- Build a Course Catalog Service
- RestFul API that manages the course catalog for an online learning platform
- Use DB for storing the course Information



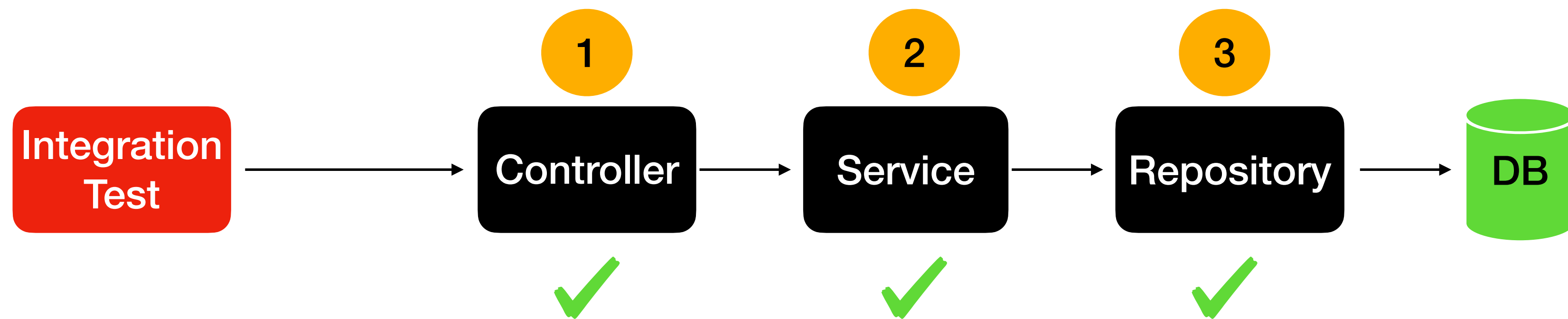
Automated Testing Using JUnit5

Automated Tests

- Automated Tests plays a vital role in delivering quality Software
- Two types of Automated Tests:
 - Integration Tests
 - Unit Tests

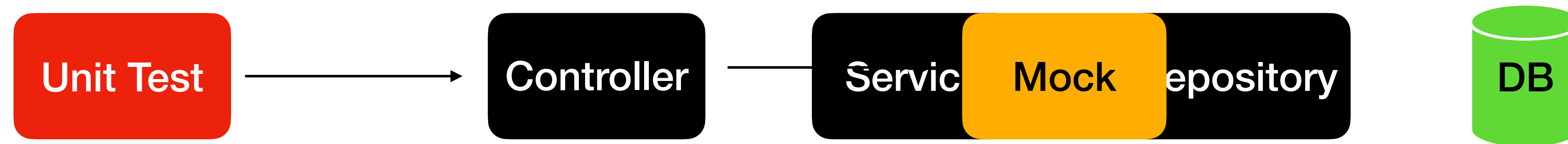
Integration Tests

- Integration test is a kind of test which actually test the application end to end



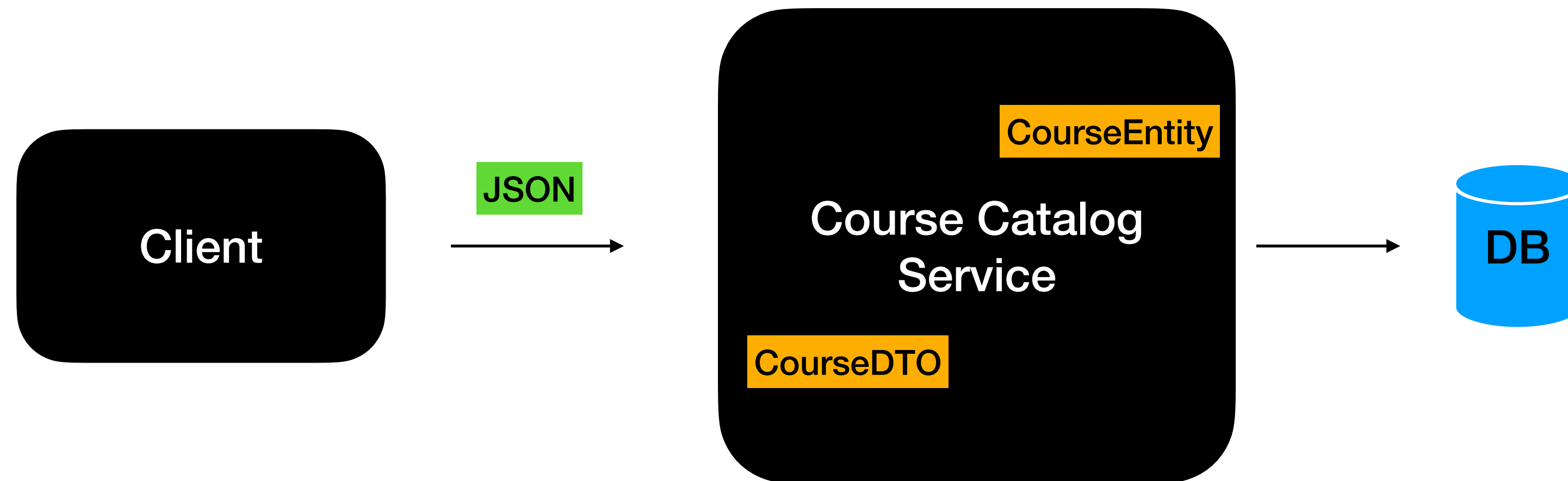
Unit Tests

- Unit test is a kind of test which tests only the class and method of interest and mocks the next layer of the code

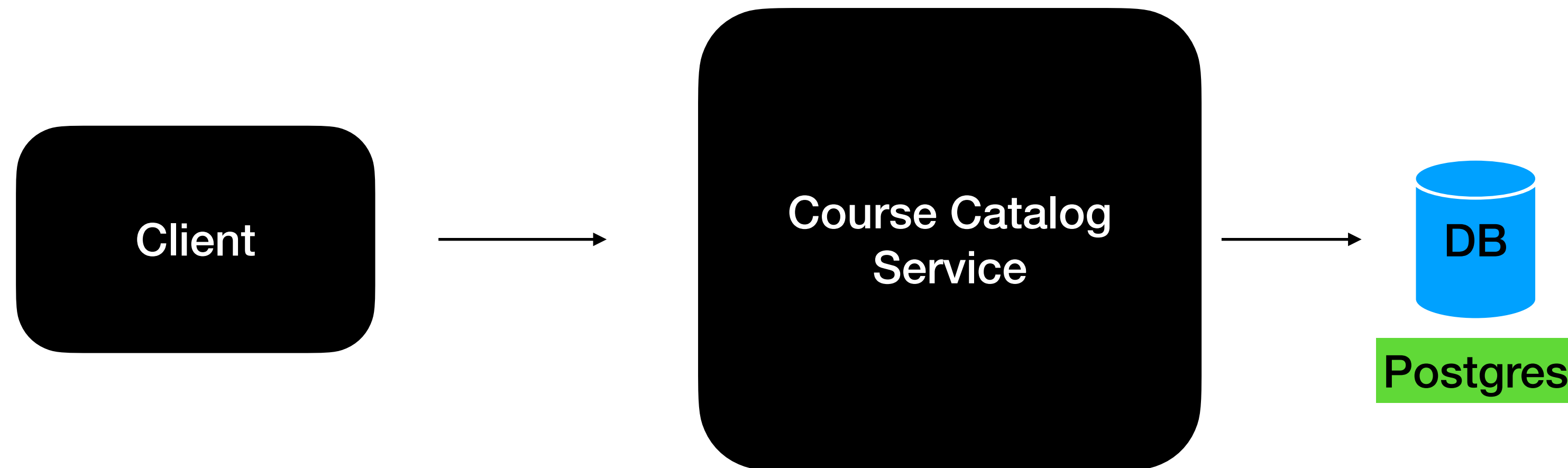


Course Catalog Service

• \



Course Catalog Service



JVM related Kotlin Annotations

- **@JvmOverloads** - This is used to instruct the JVM to create the overloaded version of methods when dealing with default values
- **@JvmField** - This is used to instruct the JVM to create the variable as a field
- **@JvmName** - This is used to instruct the JVM to create a custom name for a class or function
- **@JvmStatic** - This is used to expose the function that's part of the companion object or object as a static function