Lecture 10

COMP 3717- Mobile Dev with Android Tech

第10讲

COMP 3717 - 使用Android技术进行移动开发

Asynchronous programming

- The execution of one task isn't dependent on another (aka. non blocking)
- All our code in this course so far has run *synchronously*
  - The execution of each operation depends on completing the one before it
- Asynchronous operations allow a program to be more efficient
  - E.g., Making a request to a server without freezing the screen

异步编程

- 一个任务的执行不依赖于另一个任务（也称为非阻塞）
- 到目前为止，本课程中的所有代码都是以同步方式运行的
  - 每个操作的执行都依赖于前一个操作的完成
- 异步操作使程序更加高效
  - 例如，在不冻结屏幕的情况下向服务器发送请求

Asynchronous tools

- *AsyncTask*
  - Deprecated, too unstable
- *Executors* and *Futures*
  - Recommended for java
- RxJava & RxKotlin
  - Popular library
- *Coroutines*
  - Recommended for Kotlin

异步工具

- AsyncTask
  - 已弃用，过于不稳定
- Executors 和 Futures
  - 推荐用于 Java
- RxJava & RxKotlin
  - 热门库
- 协程
  - 推荐用于 Kotlin

Coroutines

- A concurrency design pattern
- A coroutine can be compared to a thread but are different at the lower level
- At the lower level, a coroutine saves state and runs it at later time
  - Uses *continuations* under the hood; a special type of callback
  - This can be done on a single thread

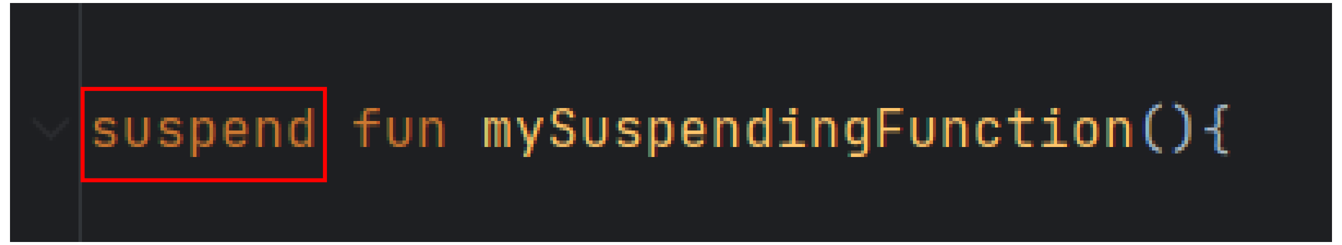
协程

- 一种并发设计模式
- 协程可以与线程相比较，但在底层有所不同
- 在底层 , 协程会保存状态，并在稍后的时间继续运行
  - 底层使用 续体；这是一种特殊的回调
  - 这可以在单个线程上完成

Coroutines (cont.)

- The way a coroutine saves state it through *suspending* functions

- To make a function suspending we use the *suspend* modifier

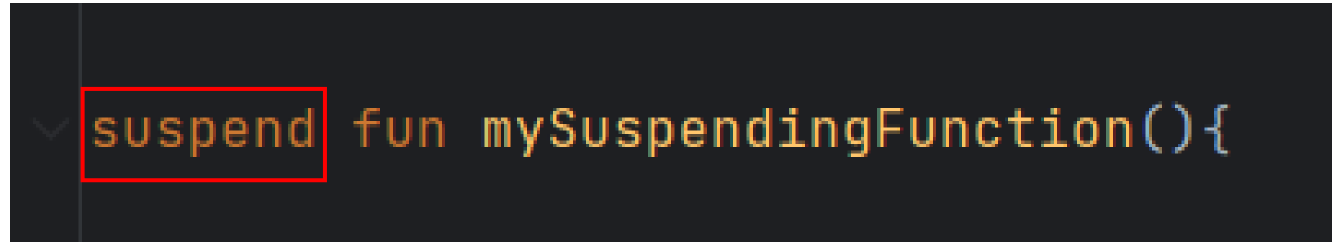


```
suspend fun mySuspendingFunction(){
```

协程（续）

- 协程通过挂起函数来保存状态

- 要使一个函数可挂起，我们使用 *suspend* 修饰符



```
suspend fun mySuspendingFunction(){
```

Coroutines

- When a coroutine calls a suspending function, the coroutine is *suspended*

- Once suspended its state gets saved and the regular flow of operations continue (aka. non-blocking)

- When the suspended operation completes (e.g. an http request), its state is restored back with the regular flow of operations

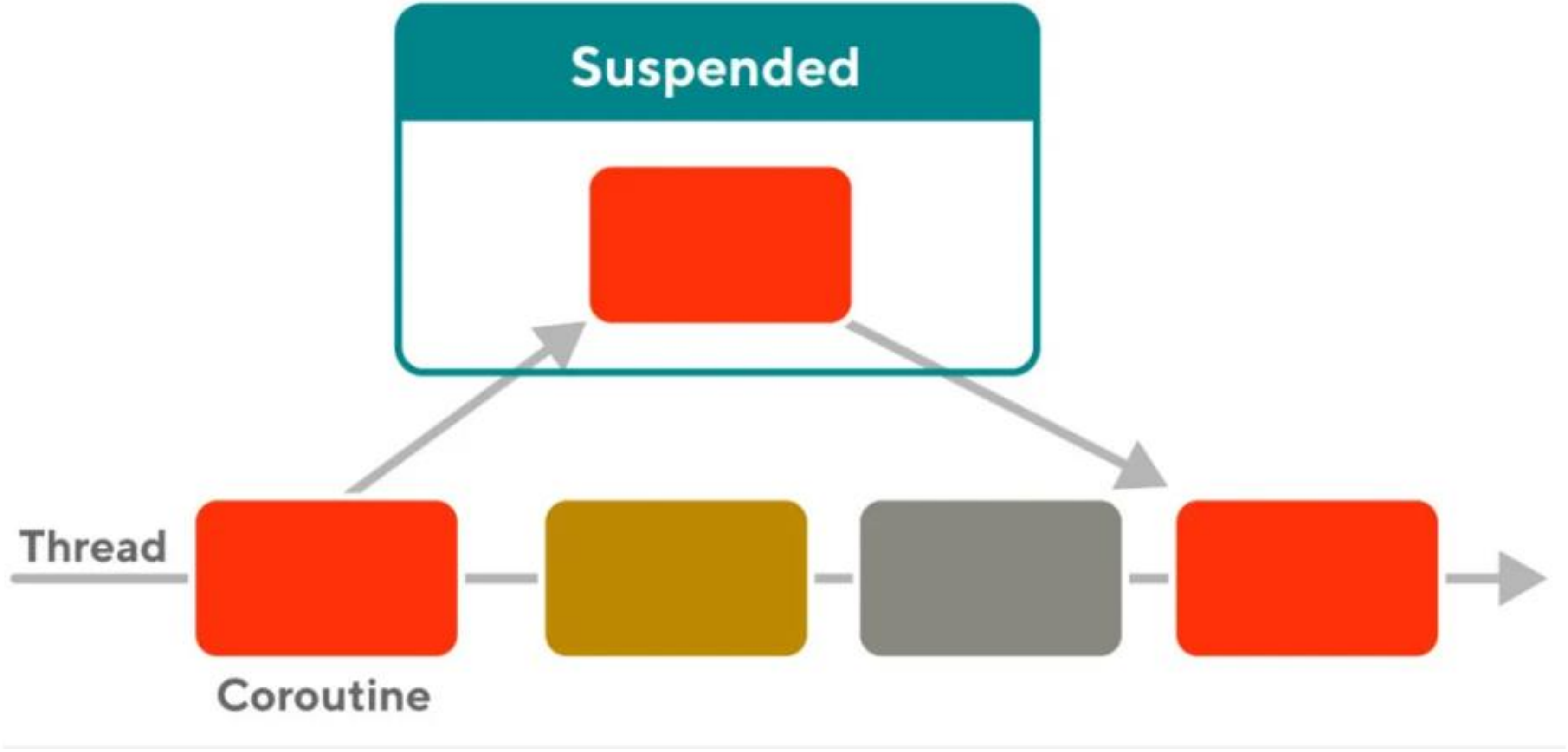
协程

- 当协程调用一个挂起函数时，该协程会被挂起

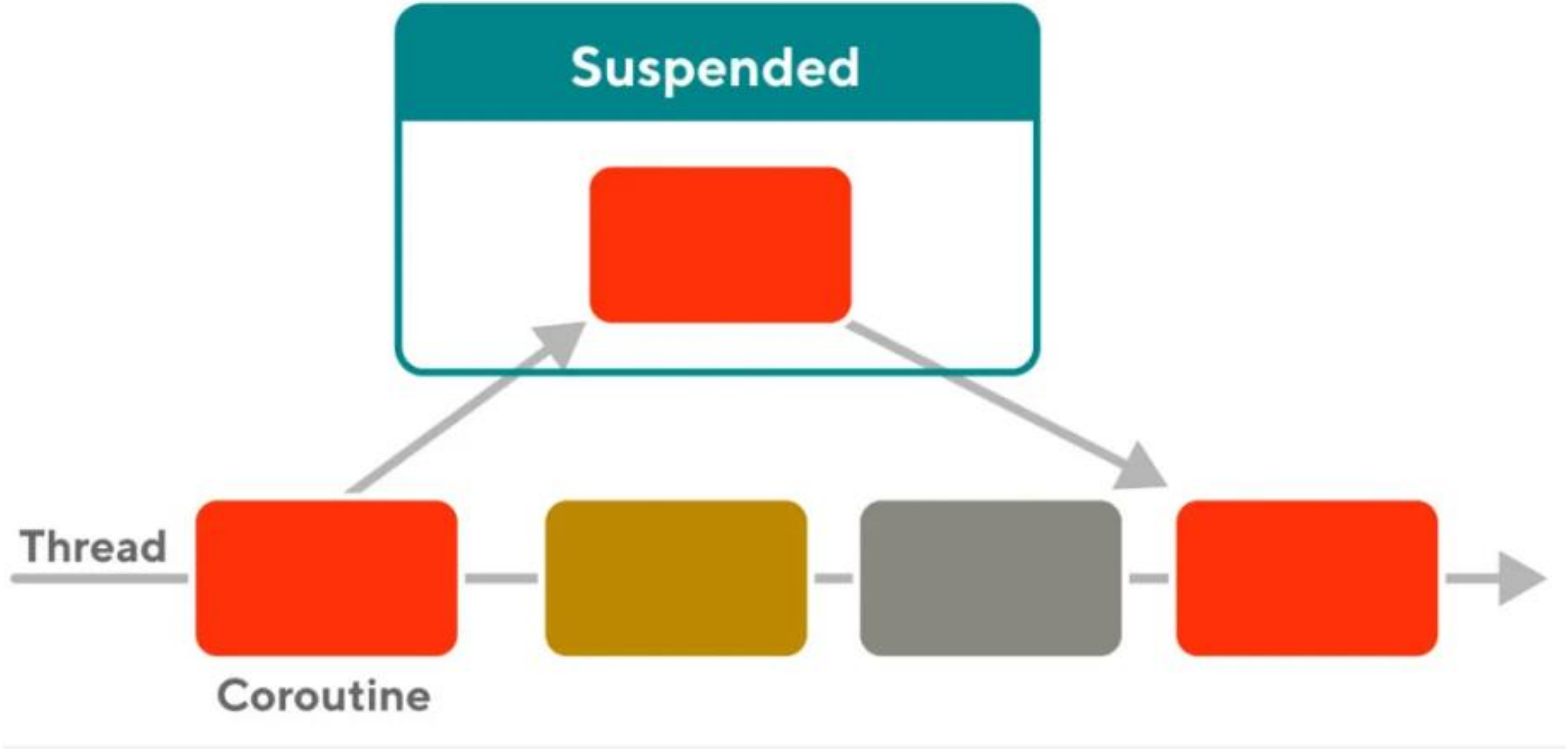
- 一旦被挂起，其状态将被保存，而常规操作流程将继续执行（即非阻塞）

- 当挂起的操作完成（例如一个 HTTP 请求），其状态将恢复，并继续常规操作流程

Coroutines



协程



Coroutines

- You call suspend functions only from other suspend functions or directly within a coroutine

- To create a coroutine we use a coroutine builder
  - runBlocking
    - Blocks current thread, usually used at top level of application
  - launch
    - Non-blocking; returns a *Job* object and does not provide a result
  - async
    - Non-blocking; returns a *Deferred Job* which provides a result

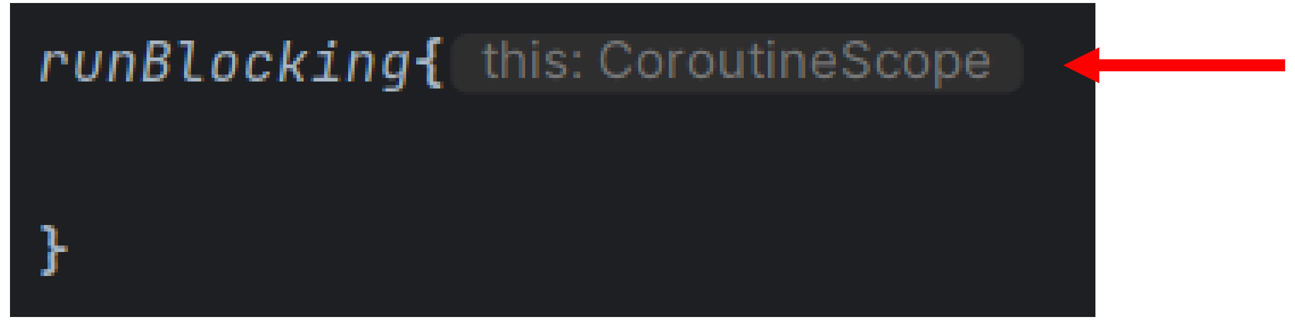
协程

- 只能从其他挂起函数或直接在协程内部调用挂起函数

- 要创建协程，我们使用协程构建器
  - runBlocking
    - 阻塞当前线程，通常在应用程序的顶层使用
  - launch
    - 非阻塞；返回一个*Job*对象且不提供结果
  - async
    - 非阻塞；返回一个*Deferred Job*，该对象提供结果

Kotlin Coroutines (cont.)

- A *CoroutineScope* defines the lifetime of a coroutine and its context
  - A *CoroutineContext* defines how a coroutine is executed
  - Every coroutine will need a *CoroutineScope*

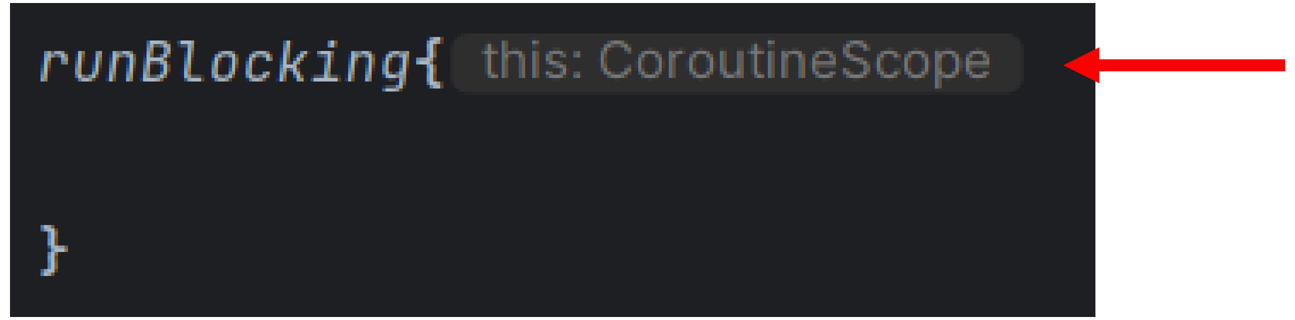


```
runBlocking{ this: CoroutineScope

}
```

- Usually, we must provide our own *CoroutineScope,* but it is created internally using *runBlocking*

Kotlin 协程（续）

- *CoroutineScope* 定义了协程的生命周期及其上下文
  - *CoroutineContext* 定义了协程的执行方式
  - 每个协程都需要一个 *CoroutineScope*

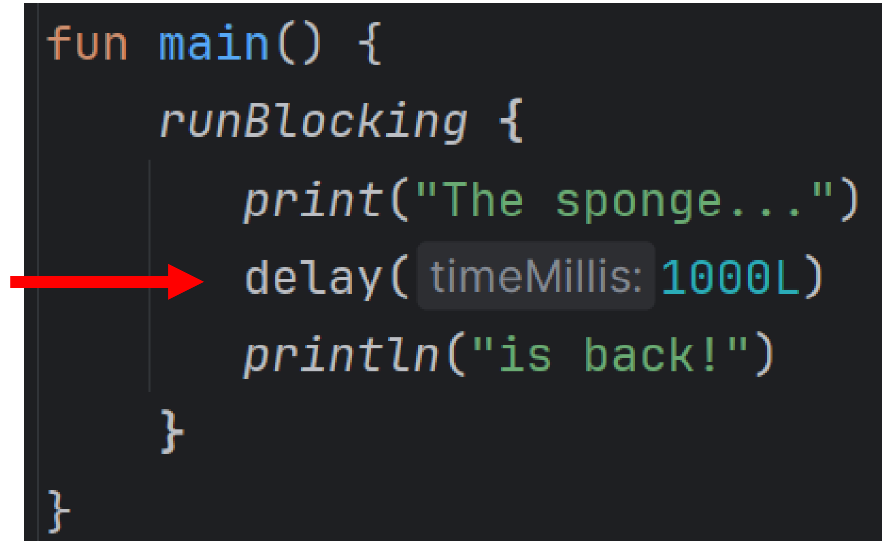


```
runBlocking{ this: CoroutineScope

}
```

- 通常我们必须提供自己的 CoroutineScope，但它会通过 *runBlocking* 在内部创建

Kotlin Coroutines (cont.)

- delay
  - A suspending function that delays the coroutine for a given duration

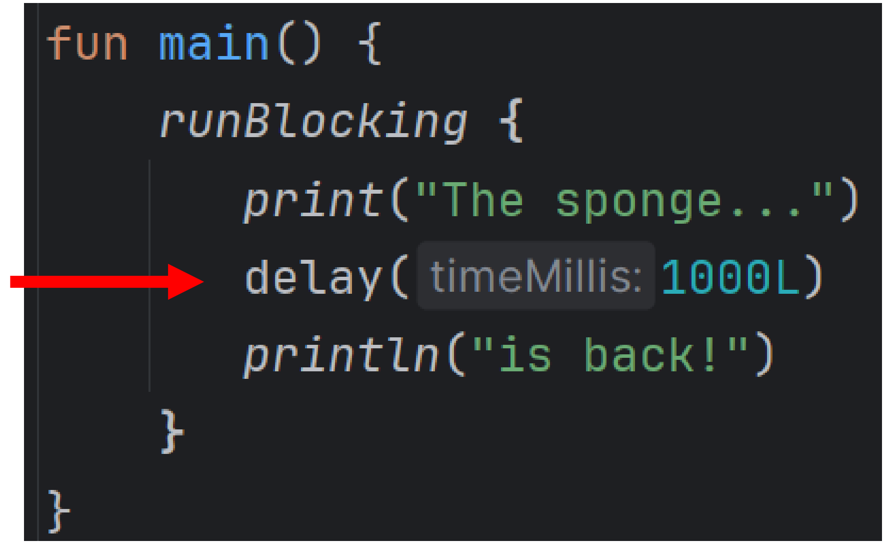


```
fun main() {
    runBlocking {
        print("The sponge...")
        delay( timeMillis: 1000L)
        println("is back!")
    }
}
```

- Since *runBlocking* blocks the current thread, the above is like using *Thread.sleep*

Kotlin 协程（续）

- delay
  - 一个挂起函数，用于将协程延迟指定的时长

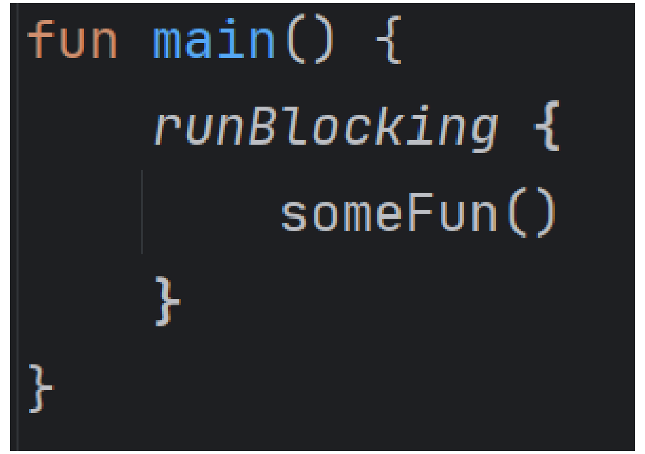


```
fun main() {
    runBlocking {
        print("The sponge...")
        delay( timeMillis: 1000L)
        println("is back!")
    }
}
```

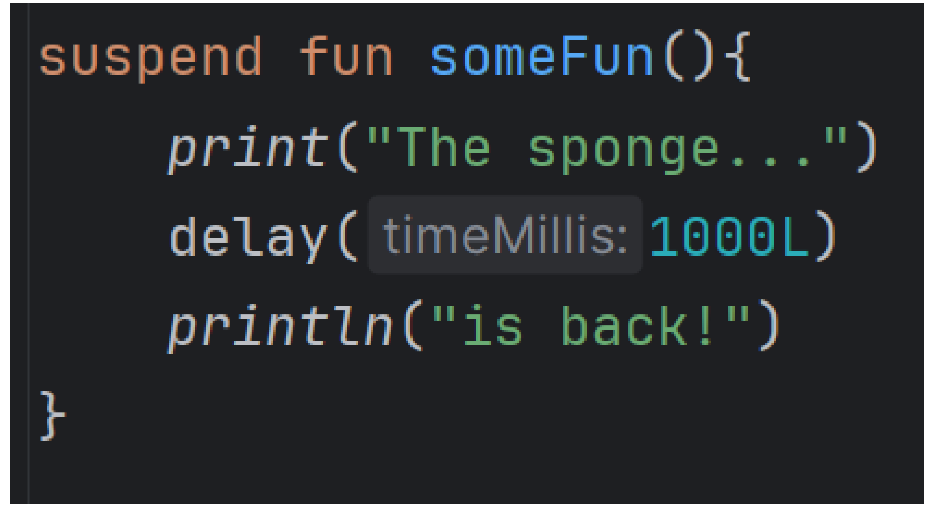
- 由于 *runBlocking* 会阻塞当前线程，因此上述代码的作用类似于使用 *Thread.sleep*

Kotlin Coroutines (cont.)

- Here I am using the same logic but with my own suspend function



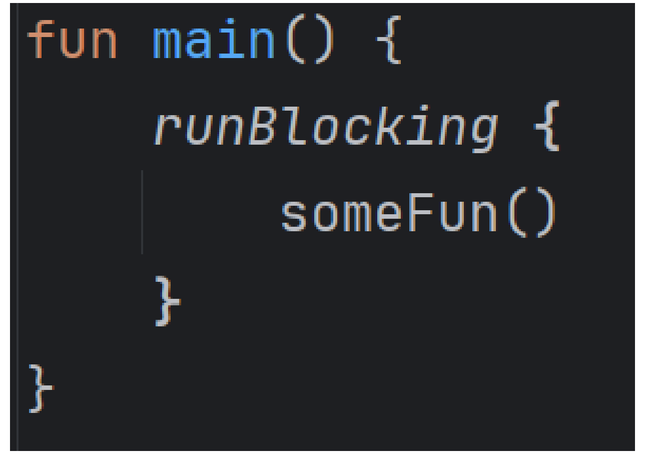
```kotlin
fun main() {
    runBlocking {
        someFun()
    }
}
```



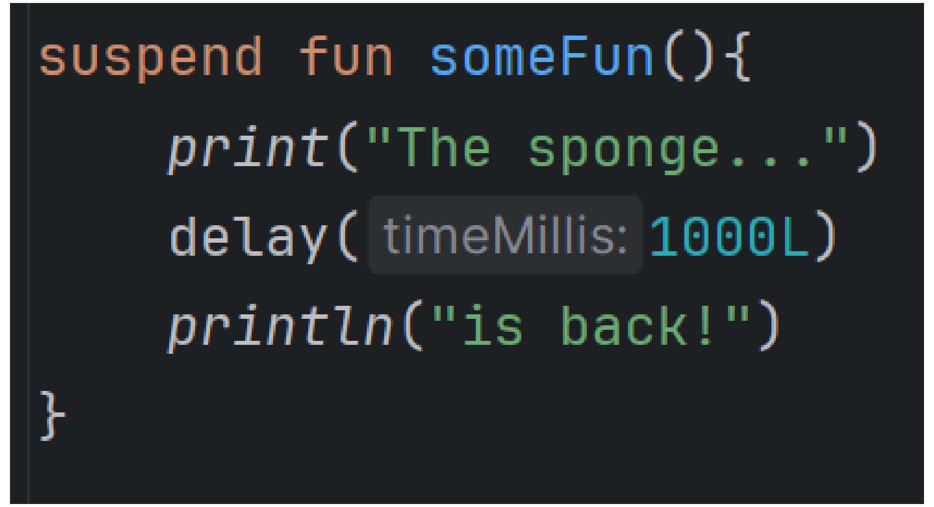
```kotlin
suspend fun someFun(){
    print("The sponge...")
    delay( timeMillis: 1000L)
    println("is back!")
}
```

Kotlin 协程（续）

- 这里我使用了相同的逻辑，但用的是我自己定义的挂起函数



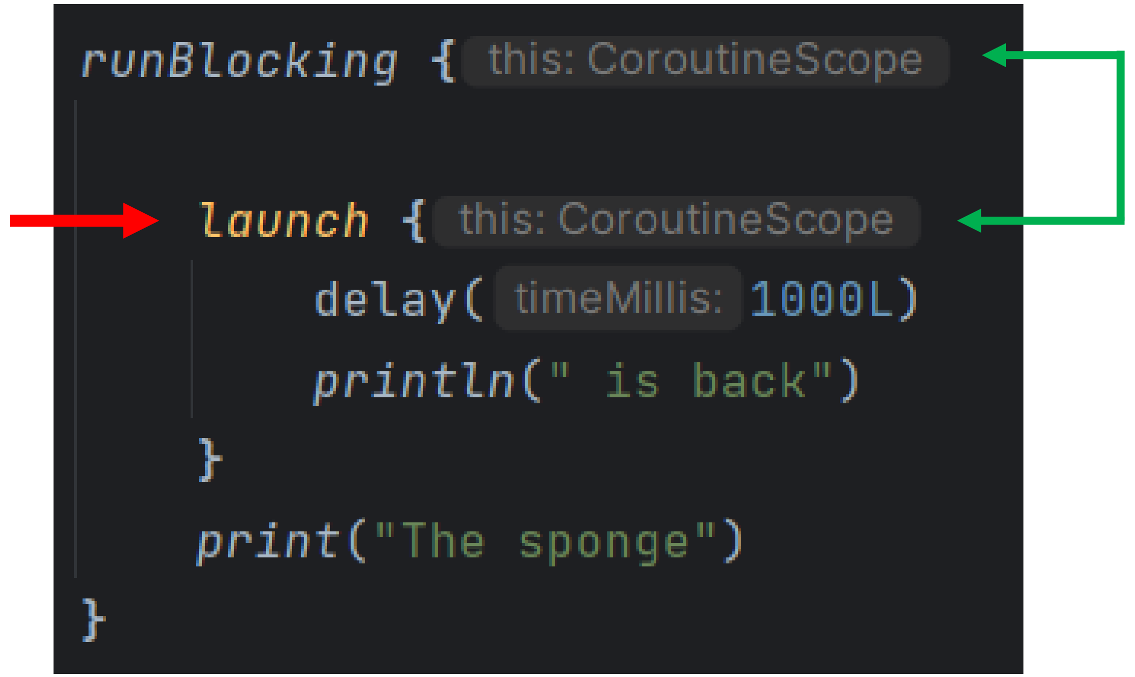
```kotlin
fun main() {
    runBlocking {
        someFun()
    }
}
```



```kotlin
suspend fun someFun(){
    print("The sponge...")
    delay( timeMillis: 1000L)
    println("is back!")
}
```

Kotlin Coroutines (cont.)

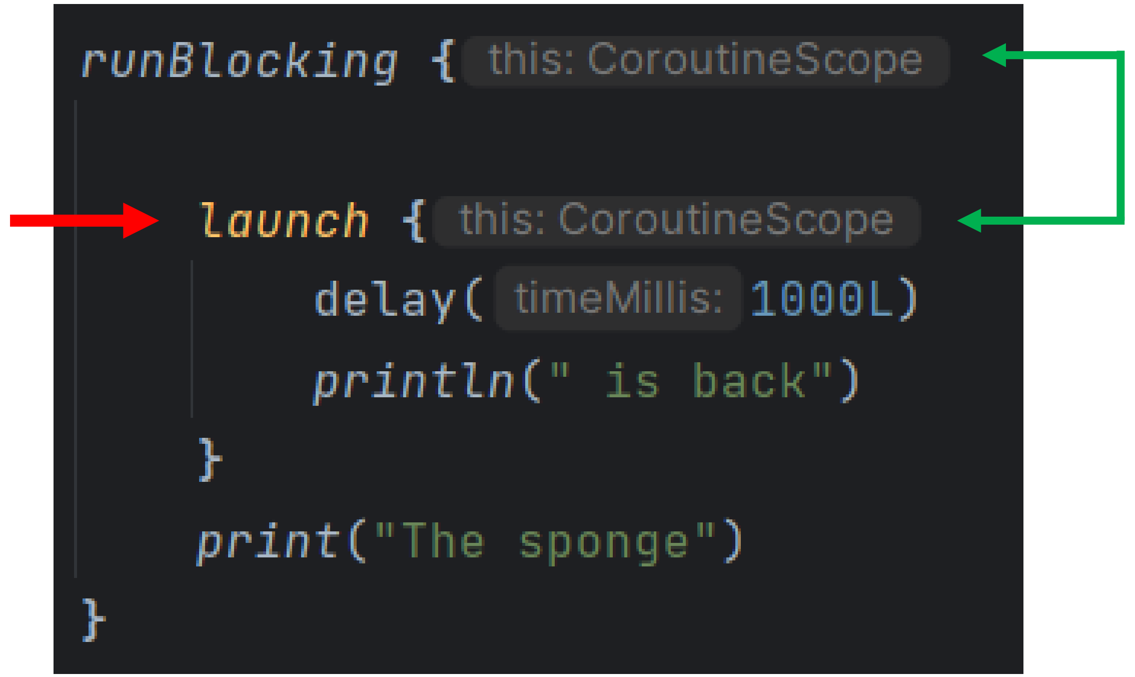
- The launch coroutine builder creates a non-blocking coroutine
  - It can only be called with a *CoroutineScope*
  - In this case, it inherits runBlocking's *CoroutineScope*



```
runBlocking { this: CoroutineScope

    launch { this: CoroutineScope
        delay( timeMillis: 1000L)
        println(" is back")
    }
    print("The sponge")
}
```

Kotlin 协程（续）

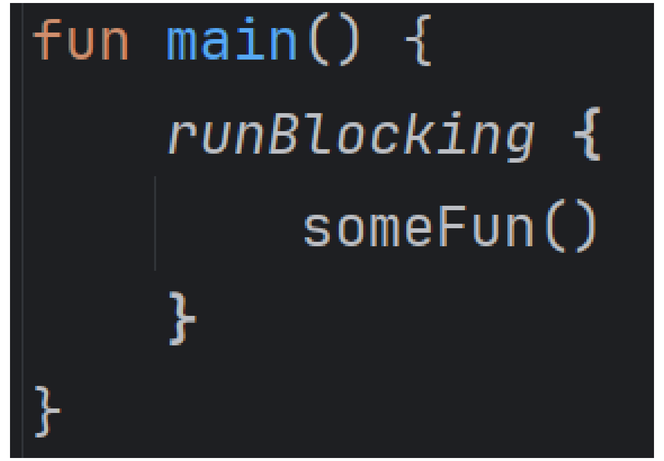
- 启动 launch 协程构建器会创建一个非阻塞的协程
  - 它只能在拥有 *CoroutineScope* 的情况下被调用
  - 在这种情况下，它 继承了 runBlocking 的 *CoroutineScope*



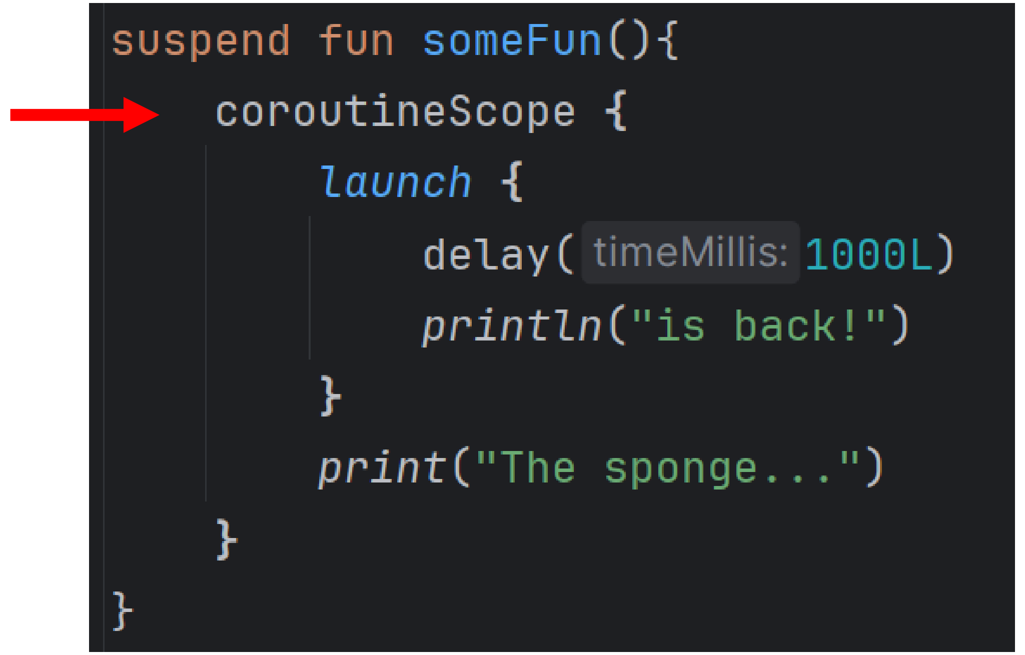
```
runBlocking { this: CoroutineScope

    launch { this: CoroutineScope
        delay( timeMillis: 1000L)
        println(" is back")
    }
    print("The sponge")
}
```

Kotlin Coroutines (cont.)

- We could also move all of this in to a suspend function and provide our own *CoroutineScope* using the *coroutineScope*



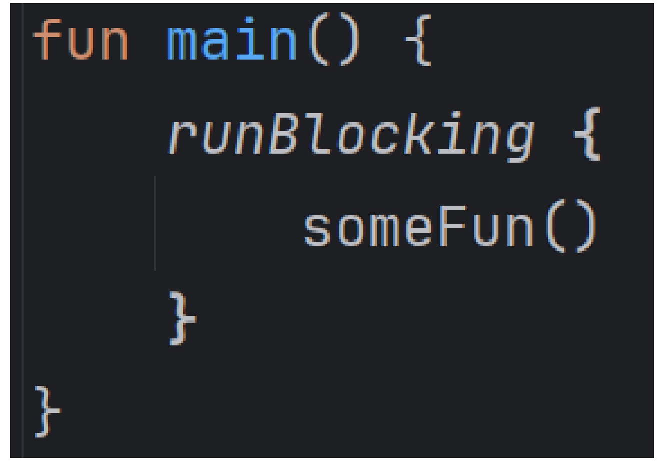
```
fun main() {
    runBlocking {
        someFun()
    }
}
```



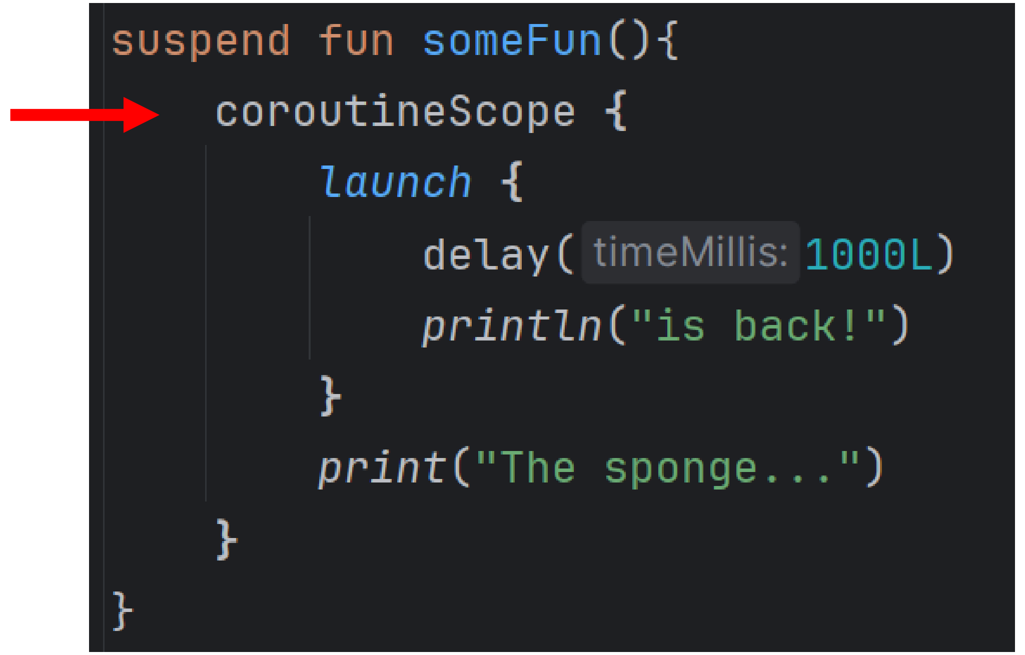
```
suspend fun someFun(){
    coroutineScope {
        launch {
            delay( timeMillis: 1000L)
            println("is back!")
        }
        print("The sponge...")
    }
}
```

Kotlin 协程（续）

- 我们还可以将所有这些移至一个挂起函数中，并提供我们自己的 *CoroutineScope* 并使用 *coroutineScope*



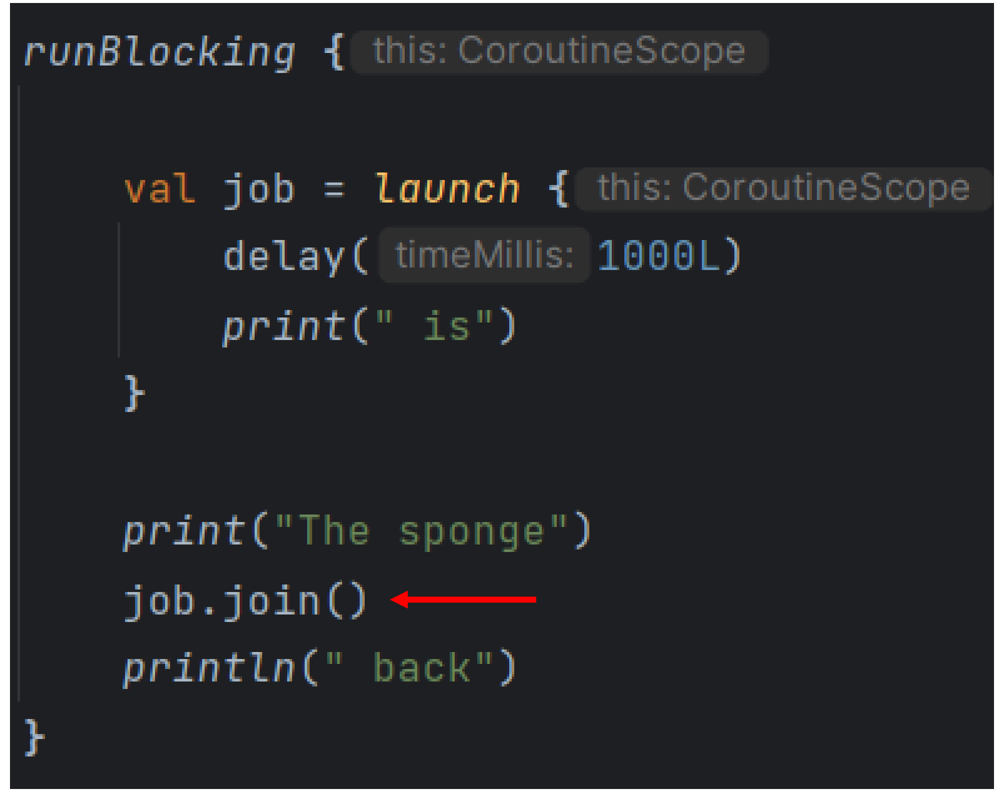
```
fun main() {
    runBlocking {
        someFun()
    }
}
```



```
suspend fun someFun(){
    coroutineScope {
        launch {
            delay( timeMillis: 1000L)
            println("is back!")
        }
        print("The sponge...")
    }
}
```

Kotlin Coroutines (cont.)

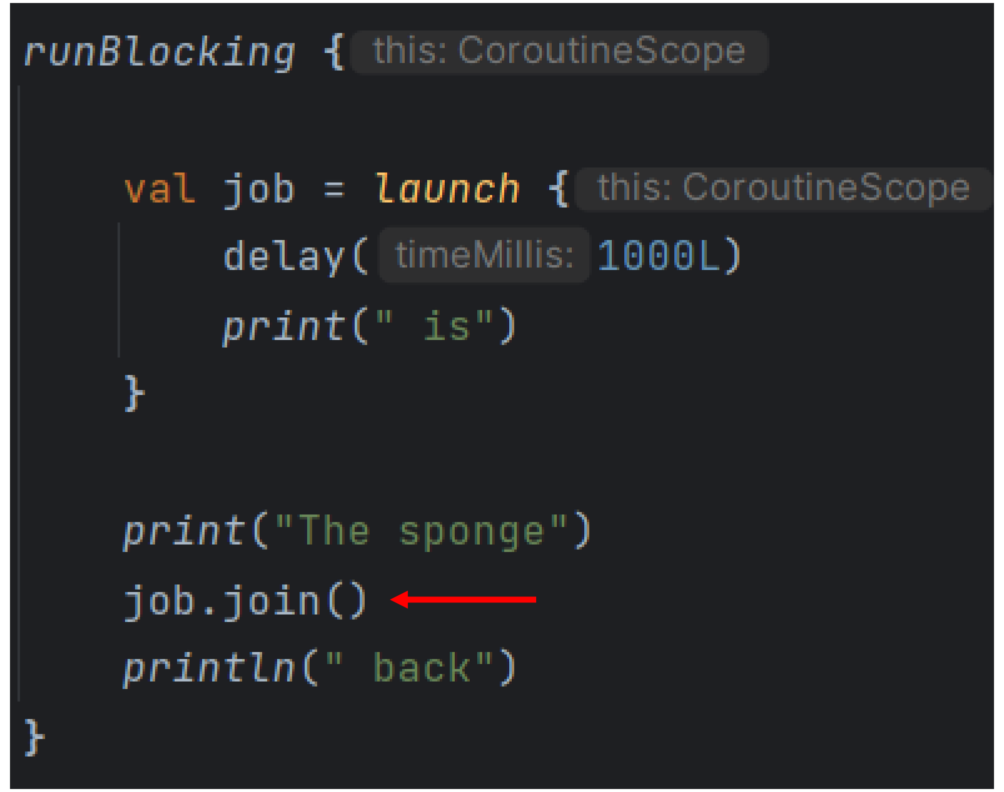
- The returned *Job* object can be used to manage our coroutine
  - *join:* Waits until coroutine is finished



```
runBlocking { this: CoroutineScope

    val job = launch { this: CoroutineScope
        delay( timeMillis: 1000L)
        print(" is")
    }

    print("The sponge")
    job.join() ←
    println(" back")
}
```

Kotlin 协程（续）

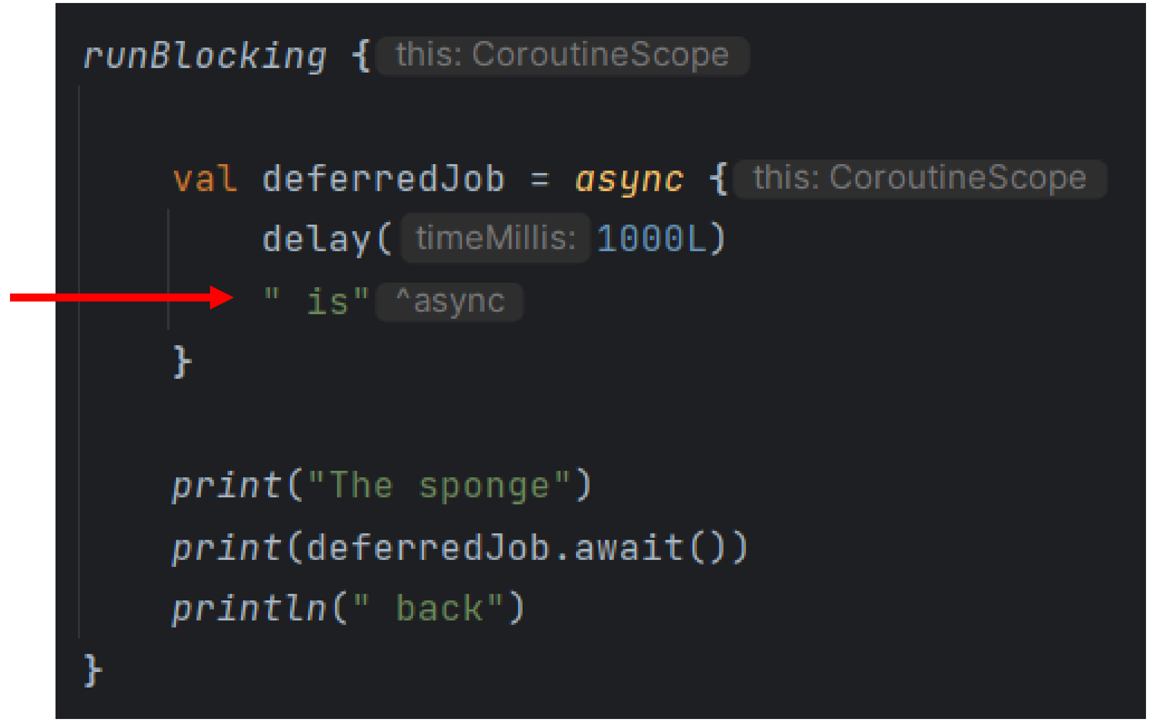
- 返回的 *Job* 对象可用于管理我们的协程
  - *join*：等待协程完成



```
runBlocking { this: CoroutineScope

    val job = launch { this: CoroutineScope
        delay( timeMillis: 1000L)
        print(" is")
    }

    print("The sponge")
    job.join() ←
    println(" back")
}
```

Kotlin Coroutines (cont.)

- We use *async* over *launch* when we want a returned result
  - await: Waits until coroutine is finished and provides a result

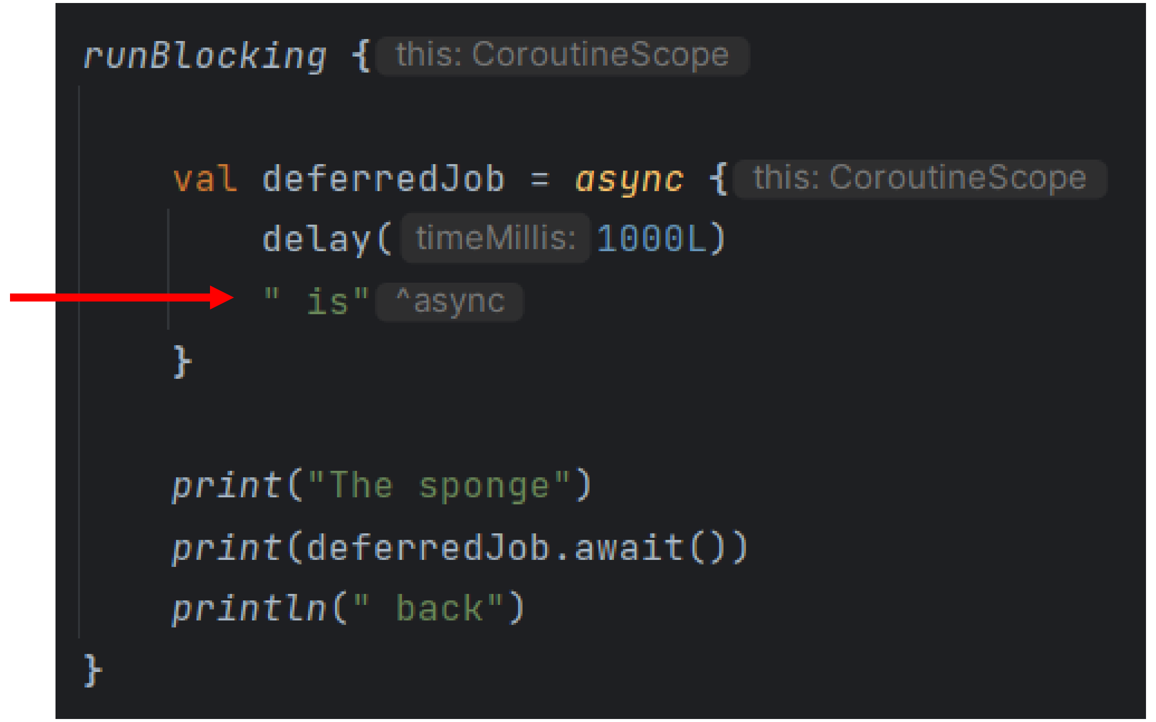


```
runBlocking { this: CoroutineScope

    val deferredJob = async { this: CoroutineScope
        delay( timeMillis: 1000L)
        " is" ^async
    }

    print("The sponge")
    print(deferredJob.await())
    println(" back")
}
```

- In this example, *deferredJob* returns a *Deferred* of type *String*

Kotlin 协程（续）

- 当我们需要返回结果时，使用 *async* 而不是 *launch*
  - await：等待协程完成并返回结果

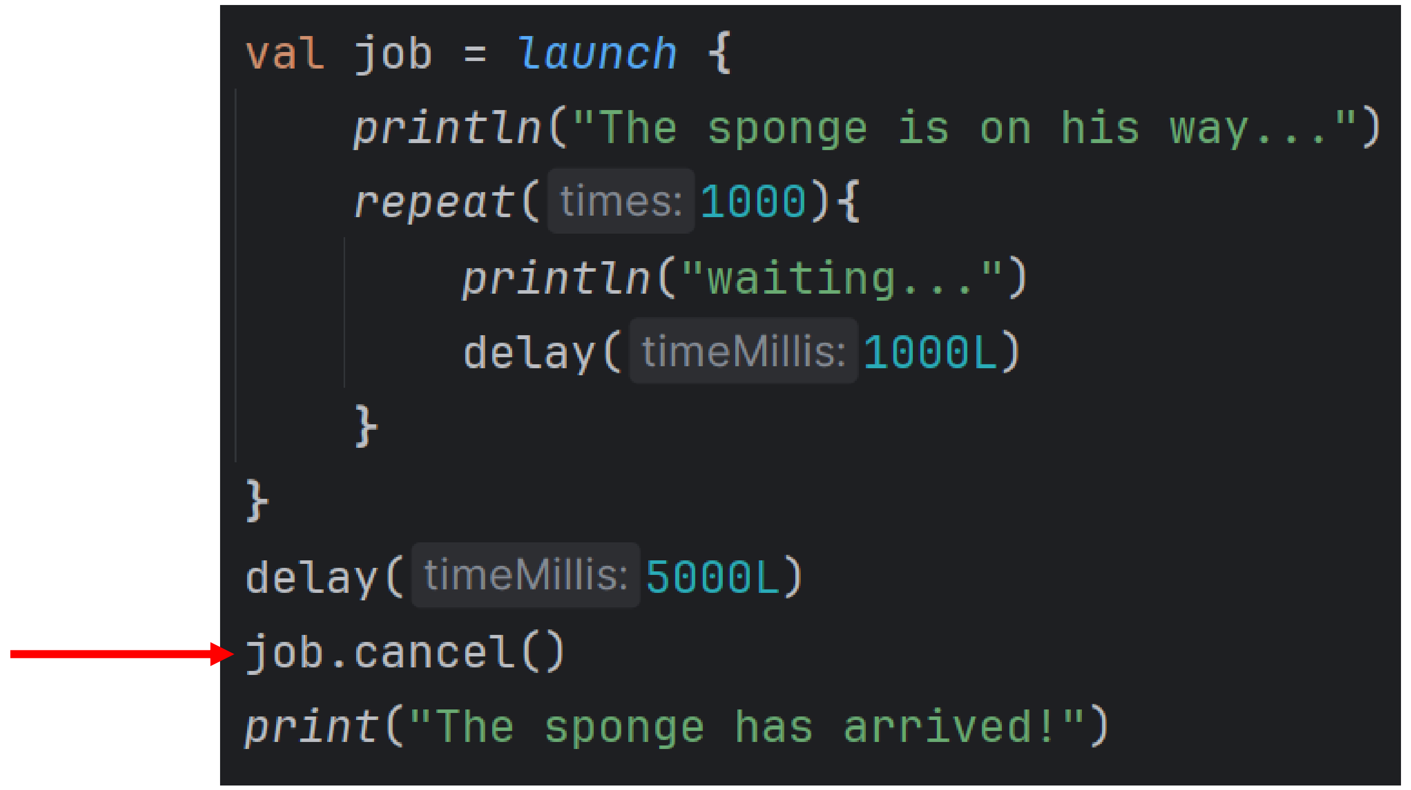


```
runBlocking { this: CoroutineScope

    val deferredJob = async { this: CoroutineScope
        delay( timeMillis: 1000L)
        " is" ^async
    }

    print("The sponge")
    print(deferredJob.await())
    println(" back")
}
```

- 在此示例中，*deferredJob* 返回一个 *Deferred* 类型为 *String*

Kotlin Coroutines (cont.)

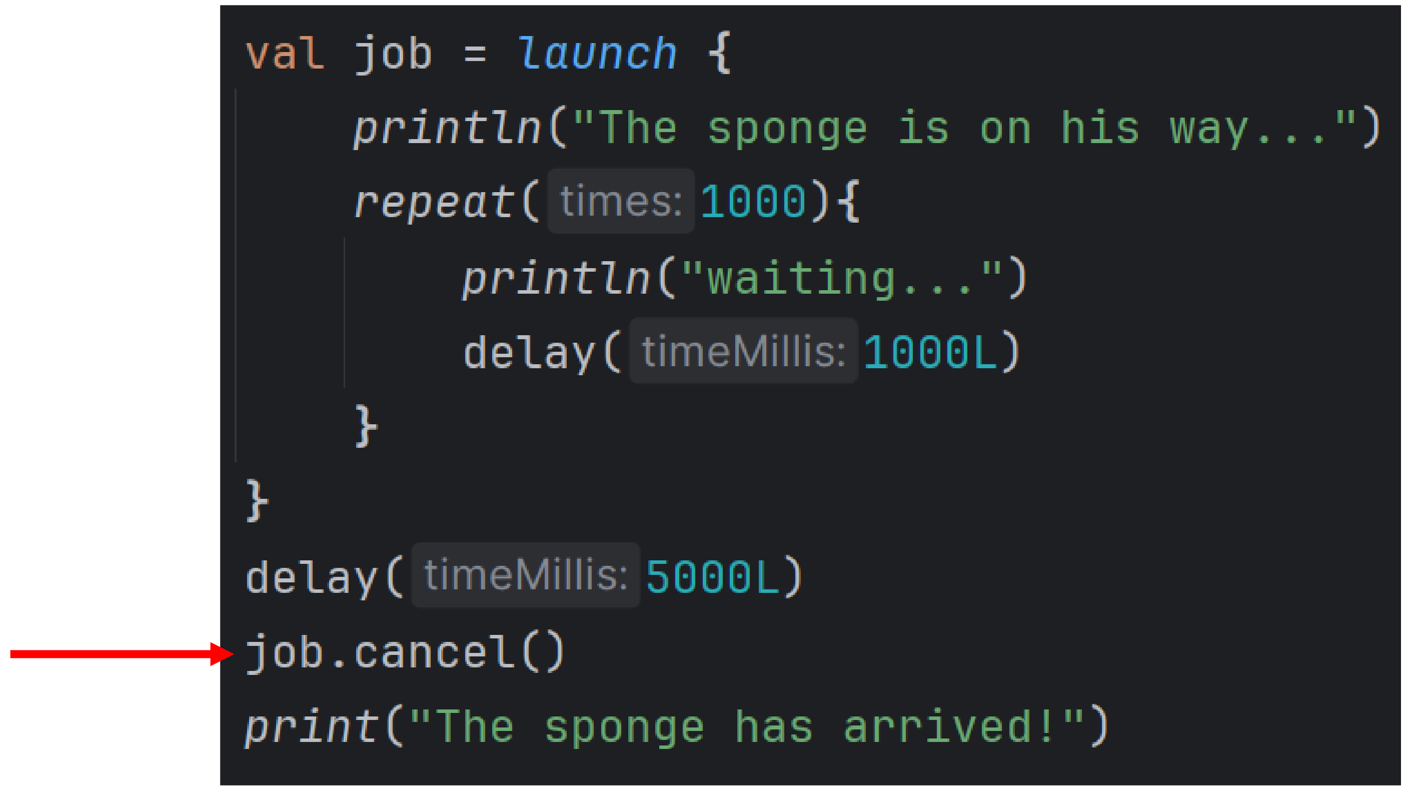
- launch and async can also be cancelled



```kotlin
val job = launch {
    println("The sponge is on his way...")
    repeat( times: 1000){
        println("waiting...")
        delay( timeMillis: 1000L)
    }
}
delay( timeMillis: 5000L)
job.cancel()
print("The sponge has arrived!")
```

Kotlin 协程（续）

- launch 和 async 也可以被取消



```kotlin
val job = launch {
    println("The sponge is on his way...")
    repeat( times: 1000){
        println("waiting...")
        delay( timeMillis: 1000L)
    }
}
delay( timeMillis: 5000L)
job.cancel()
print("The sponge has arrived!")
```

HTTP Requests

- A common asynchronous task is to make an HTTP request
  - E.g., CONNECT, GET, POST, etc. to an API
- An HTTP request can take time, especially if the network is poor
  - We don't want to freeze our app (aka. block the main thread)
- There are many ways to make HTTP requests
  - Use the java standard library
    - *HttpURLConnection*
  - Use a third-party library such as Ktor

HTTP 请求

- 一个常见的异步任务是发起 HTTP 请求
  - 例如，向 API 发送 CONNECT、GET、POST 等请求
- HTTP 请求可能需要一定时间，尤其是在网络状况不佳时
  - 我们不希望应用因此卡住（即阻塞主线程）
- 发起 HTTP 请求的方式有很多
  - 使用 Java 标准库
    - HttpURLConnection
  - 使用第三方库，例如 Ktor

Serialization and deserialization

- Serialization
  - Converting your data into a format that can be stored or transmitted
    - E.g., Across a network
- Deserialization
  - Converting that format back into a data structure
- Serializing data into JSON format is popular and simple
  - JSON: Text-based format that uses key-value pairs and arrays

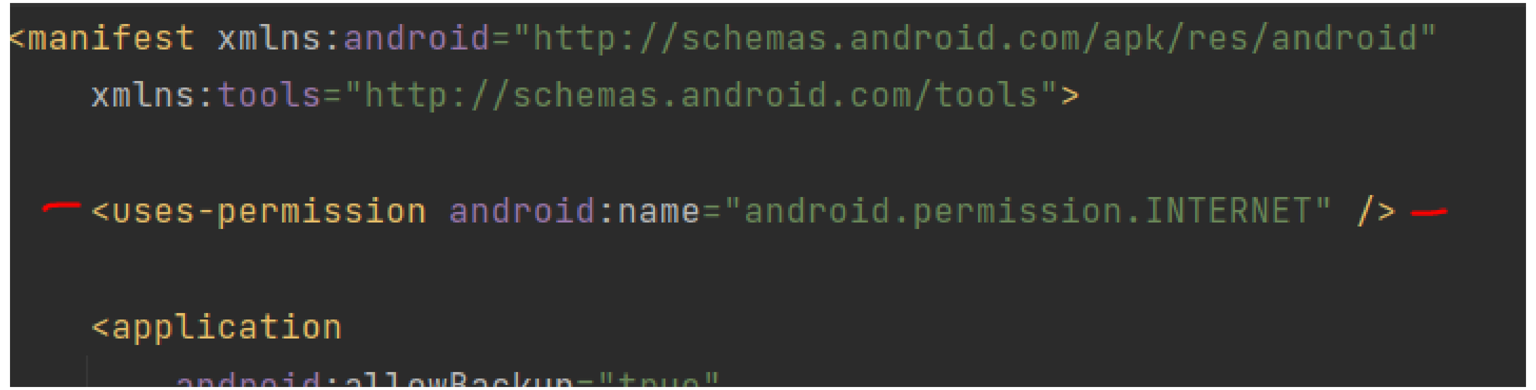
序列化与反序列化

- 序列化
  - 将数据转换为可存储或传输的格式
    - 例如，通过网络传输
- 反序列化
  - 将该格式重新转换回数据结构
- 将数据序列化为 JSON 格式既流行又简单
  - JSON：一种基于文本的格式，使用键值对和数组

Allow internet access for your app

- To give your app internet access you need to update your system permissions

- You do this in *AndroidManifest.xml* using the *<uses-permission>* element

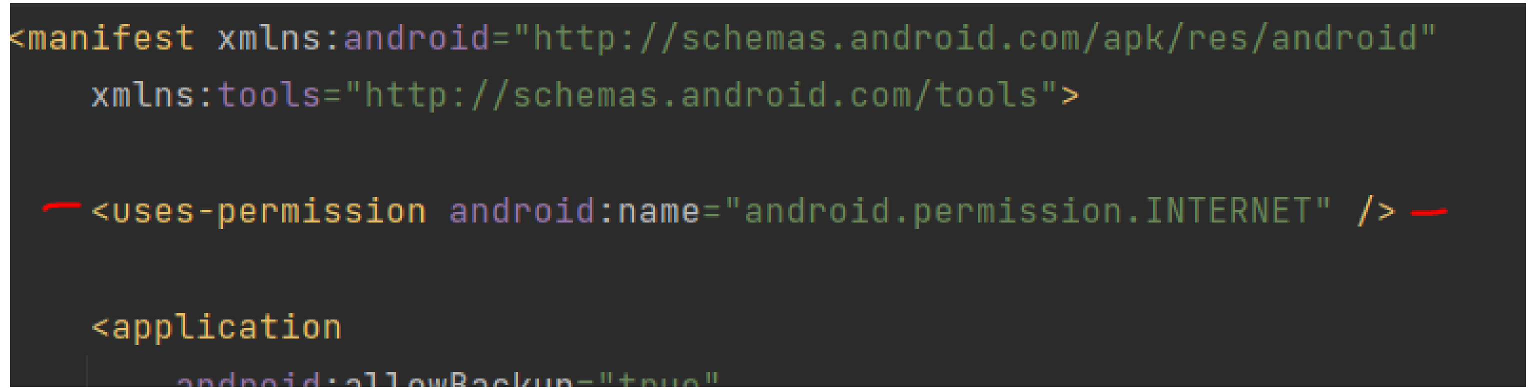


```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
```

允许应用访问互联网

- 要让应用访问互联网，您需要更新系统权限

- 您可在 AndroidManifest.xml 中使用 <uses-permission> 元素

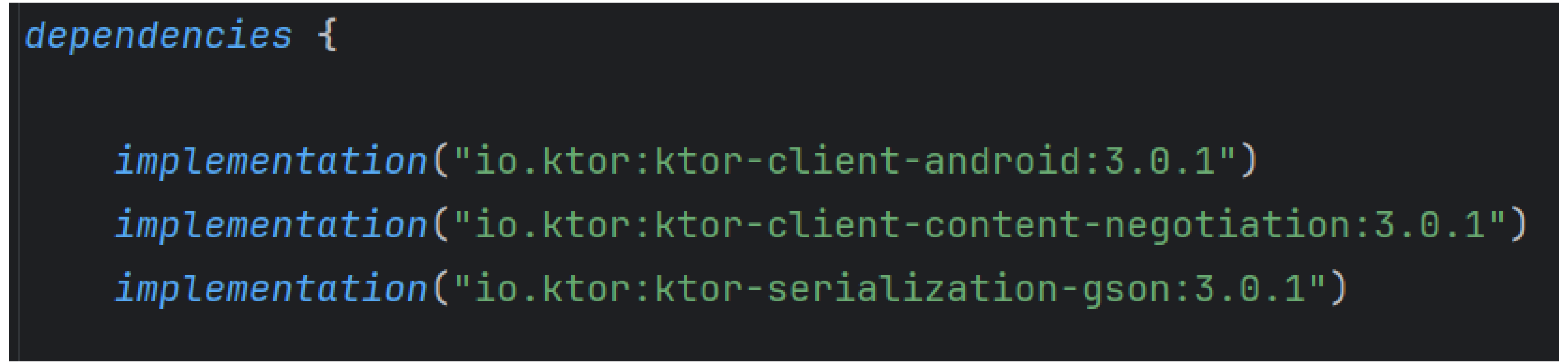


```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
```

Ktor

- Ktor is built using Kotlin Coroutines and provides us with
  - A client to make HTTP requests
  - JSON Serialization and deserialization

- To use Ktor we need to add a few dependencies

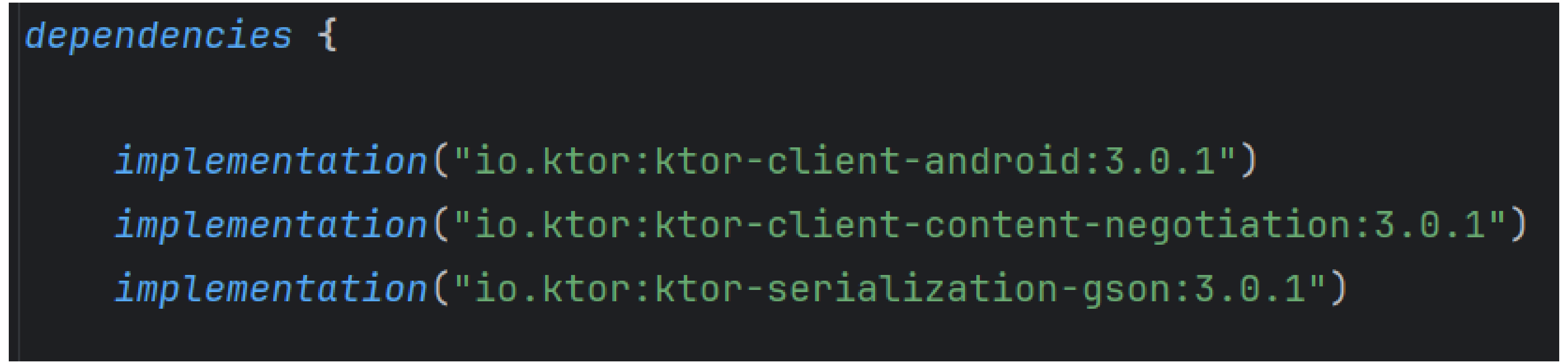


```
dependencies {

    implementation("io.ktor:ktor-client-android:3.0.1")
    implementation("io.ktor:ktor-client-content-negotiation:3.0.1")
    implementation("io.ktor:ktor-serialization-gson:3.0.1")
```

Ktor

- Ktor 基于 Kotlin 协程构建，并为我们提供了
  - 用于发起 HTTP 请求的客户端
  - JSON 序列化与反序列化

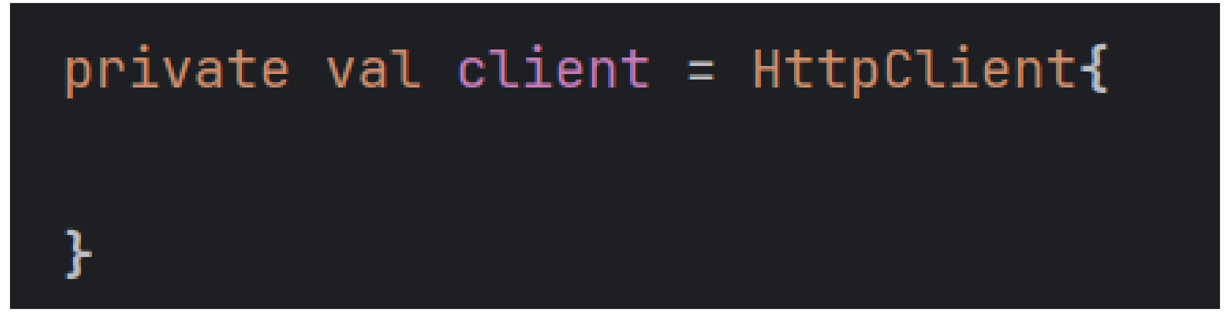
- 要使用 Ktor，我们需要添加一些依赖项



```
dependencies {

    implementation("io.ktor:ktor-client-android:3.0.1")
    implementation("io.ktor:ktor-client-content-negotiation:3.0.1")
    implementation("io.ktor:ktor-serialization-gson:3.0.1")
```

Ktor (cont.)

- Ktor's *HttpClient* is built using Coroutines

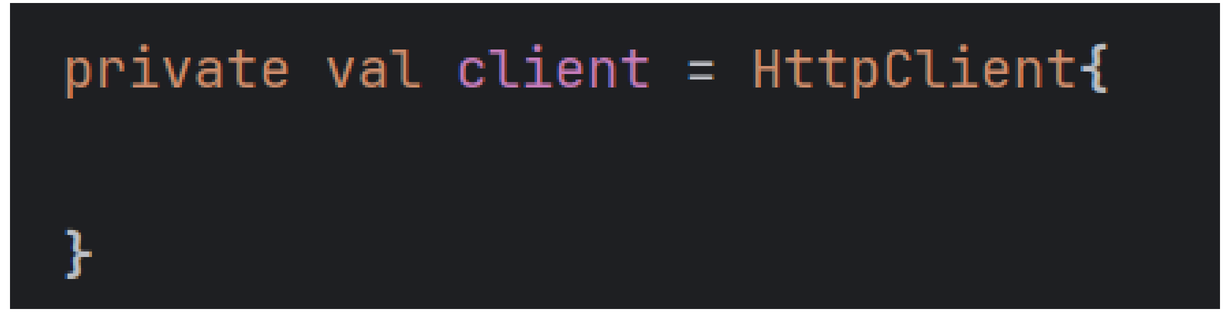


```
private val client = HttpClient{

}
```

- *HttpClient* is the entry point for creating HTTP requests

Ktor（续）

- Ktor 的 *HttpClient* 基于协程构建

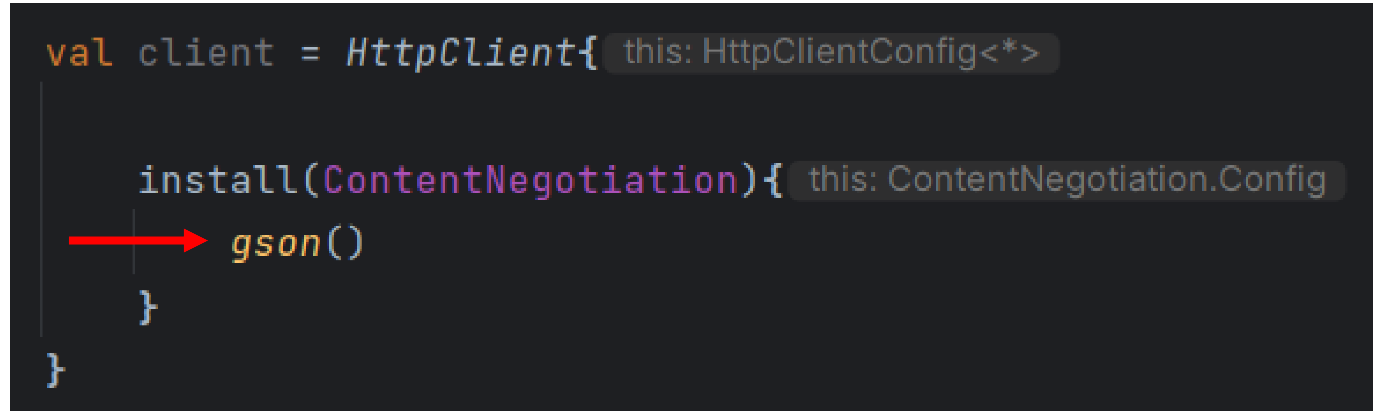


```
private val client = HttpClient{

}
```

- *HttpClient* 是创建 HTTP 请求的入口点

Ktor (cont.)

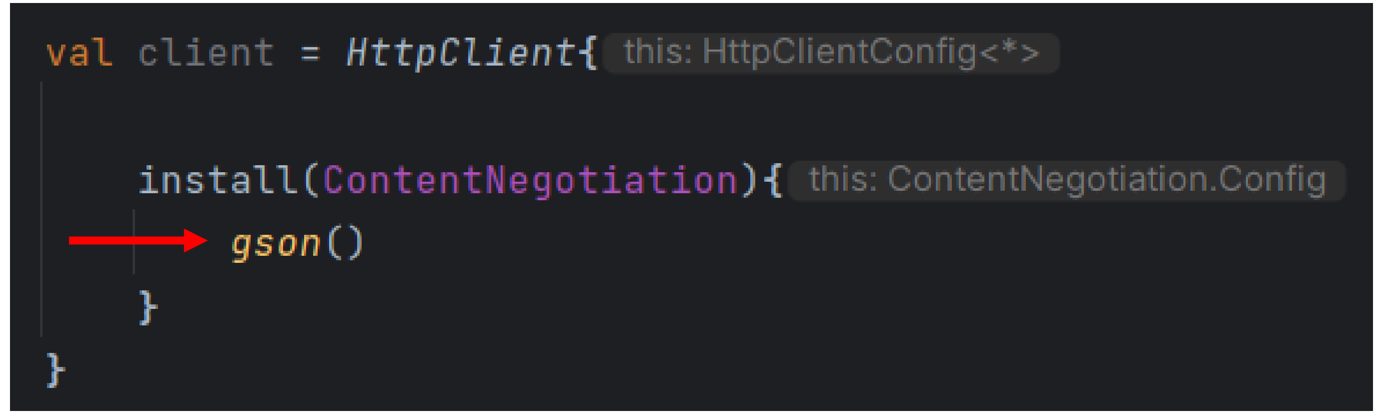
- Ktor has many JSON serializers but the one we will use is *GSON*
  - Here we use the *install* function to use the specific plugin



```
val client = HttpClient{

    install(ContentNegotiation){
        gson()
    }
}
```

Ktor（续）

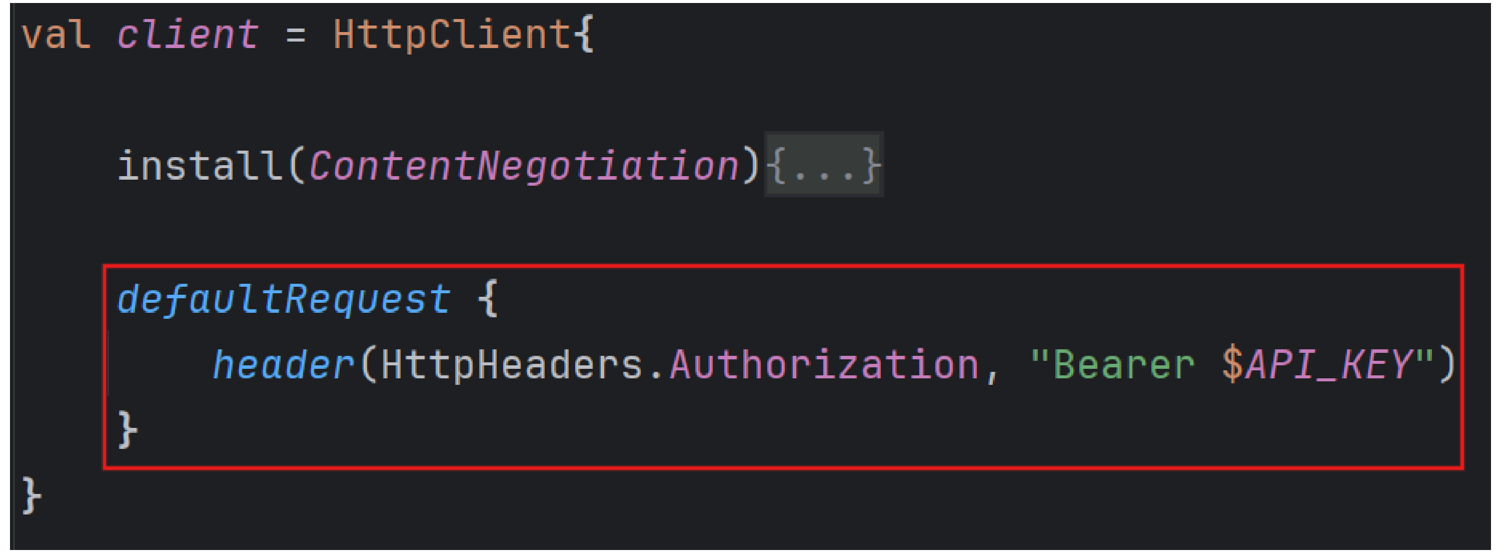
- Ktor 拥有许多 JSON 序列化器，但我们使用的将是 *GSON*
  - 这里我们使用 *install* 函数来使用特定的插件



```
val client = HttpClient{

    install(ContentNegotiation){
        gson()
    }
}
```

Ktor (cont.)

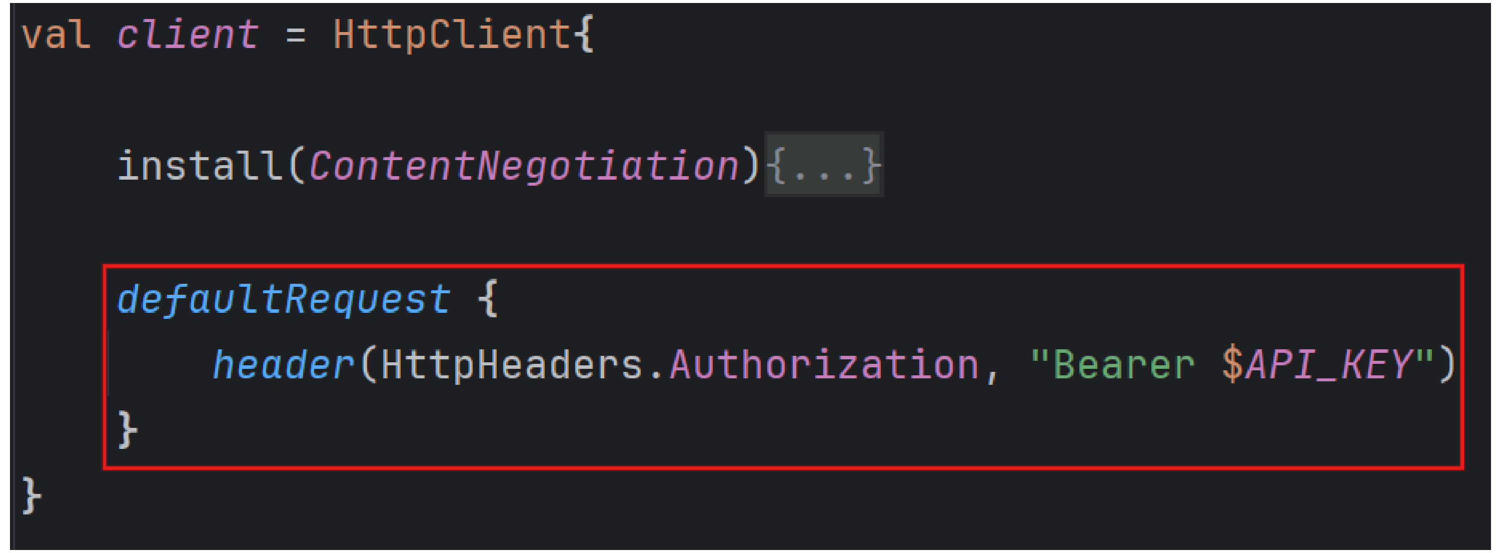
- We can add specific headers for our request here too
  - Ex. Adding an *Authorization* header and an API key in our request



```
val client = HttpClient{

    install(ContentNegotiation){...}

    defaultRequest {
        header(HttpHeaders.Authorization, "Bearer $API_KEY")
    }
}
```

Ktor（续）

- 我们也可以在这里为请求添加特定的头部信息
  - 例如，在请求中添加 *Authorization* 头部和一个 API 密钥



```
val client = HttpClient{

    install(ContentNegotiation){...}

    defaultRequest {
        header(HttpHeaders.Authorization, "Bearer $API_KEY")
    }
}
```

Consuming an API

- Application programming interface (aka. API)
  - Allows a way for two or more computers to communicate
- The API we are consuming is from the Art institute of Chicago
  - http://api.artic.edu/docs/
- Most modern APIs provide their data in JSON format
  - https://api.artic.edu/api/v1/artworks?fields=id,title,image_id

调用 API

- 应用程序编程接口（又称 API）
  - 提供了一种让两台或多台计算机相互通信的方式
- 我们正在调用的 API 来自芝加哥艺术学院
  - http://api.artic.edu/docs/
- 大多数现代 API 都以 JSON 格式提供数据
  - https://api.artic.edu/api/v1/artworks?fields=id,title,image_id

Consuming an API (cont.)

- When consuming an API, it's first important to determine what data you want in your app
- Many APIs provide far more data than is needed for the program requesting the information
- Once you know the specific data, you should create the data model

调用 API（续）

- 在调用 API 时，首先需要确定你的应用程序需要哪些数据
- 许多 API 提供的数据远超程序所需请求信息
- 一旦明确了具体数据，就应创建数据模型

Consuming an API (cont.)

- There are automated tools for creating a data models such as
  - Online JSON formatters
  - JetBrains plugin
    - JSONToKotlinClass

- You can also just create the model yourself if the API is simple enough and you know how to read JSON

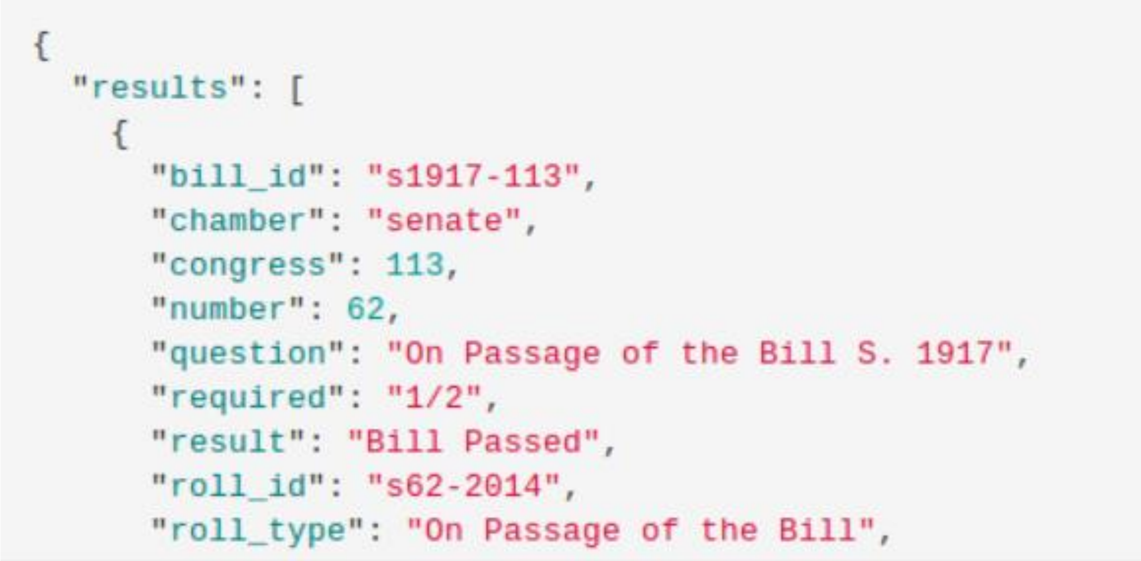
调用 API（续）

- 有一些自动生成数据模型的自动化工具，例如
  - 在线 JSON 格式化工具
  - JetBrains 插件
    - JSONToKotlinClass

- 如果 API 足够简单，并且你懂得如何阅读 JSON，也可以手动创建模型

Understanding JSON

- A JSON is made up of keys and values
  - A key is always a string
  - A value can be
    - An object which is wrapped in { }
    - An array which is wrapped in [ ]
    - A string, number or boolean



```
{
  "results": [
    {
      "bill_id": "s1917-113",
      "chamber": "senate",
      "congress": 113,
      "number": 62,
      "question": "On Passage of the Bill S. 1917",
      "required": "1/2",
      "result": "Bill Passed",
      "roll_id": "s62-2014",
      "roll_type": "On Passage of the Bill",
```

理解 JSON

- JSON 由键和值组成
  - 键始终是一个字符串
  - 值可以是
    - 用 { } 括起来的对象
    - 用 [ ] 括起来的数组
    - 字符串、数字或布尔值

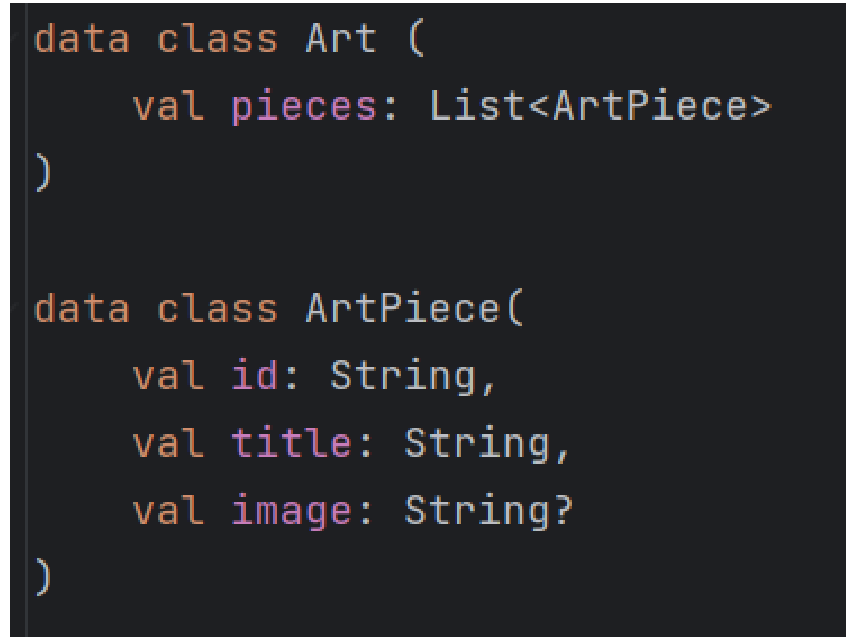
```
{
  "results": [
    {
      "bill_id": "s1917-113",
      "chamber": "senate",
      "congress": 113,
      "number": 62,
      "question": "On Passage of the Bill S. 1917",
      "required": "1/2",
      "result": "Bill Passed",
      "roll_id": "s62-2014",
      "roll_type": "On Passage of the Bill",
```

GUESS WHAT ?
BREAK TIME !
memegenerator.net

GUESS WHAT ?
BREAK TIME !
memegenerator.net

Consuming an API (cont.)

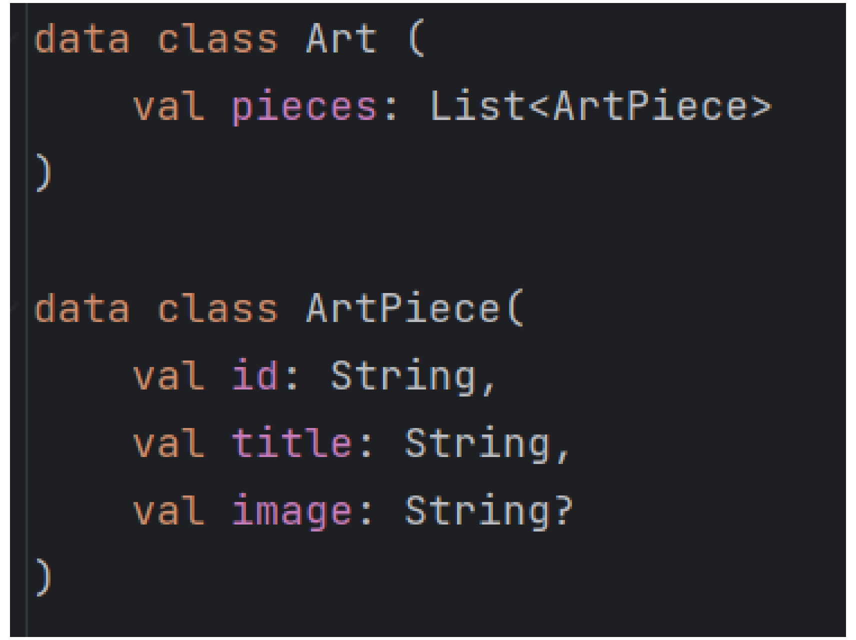
- Here is a simple data model for the JSON I want to use
  - https://api.artic.edu/api/v1/artworks?fields=id,title,image_id
  - Notice I don't need to model everything in the JSON



```
data class Art (
    val pieces: List<ArtPiece>
)

data class ArtPiece(
    val id: String,
    val title: String,
    val image: String?
)
```

调用 API（续）

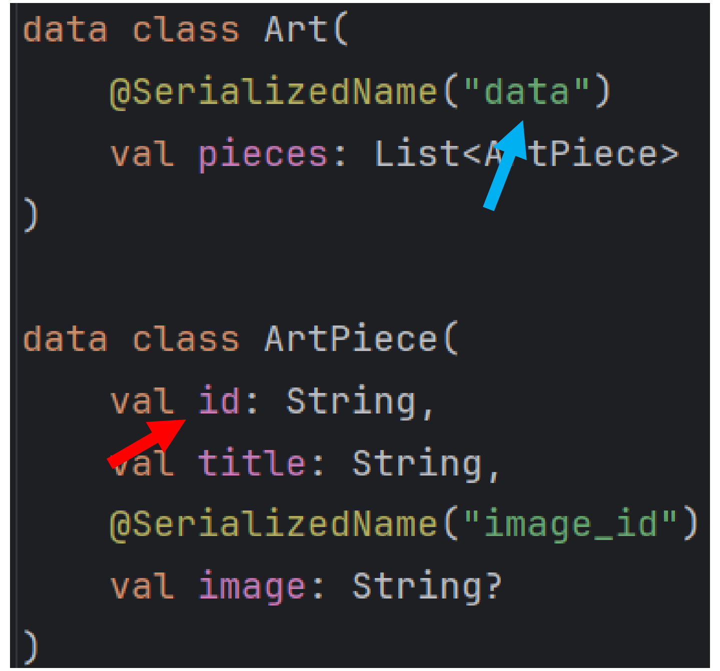
- 以下是我希望使用的 JSON 的一个简单数据模型
  - https://api.artic.edu/api/v1/artworks?fields=id,title,image_id
  - 注意，我不需要对 JSON 中的所有内容都建模



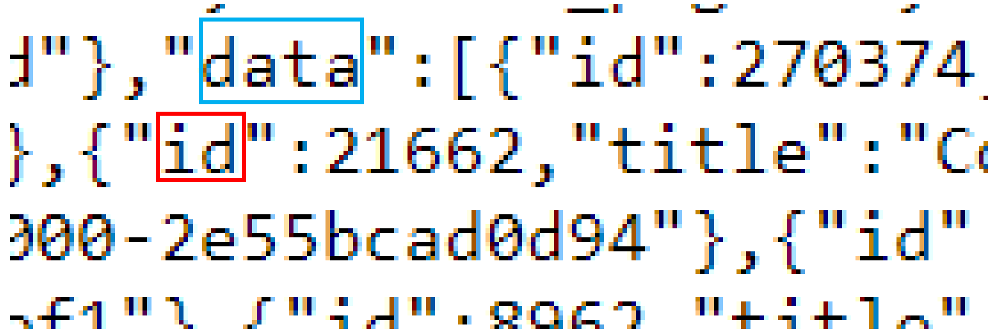
```
data class Art (
    val pieces: List<ArtPiece>
)

data class ArtPiece(
    val id: String,
    val title: String,
    val image: String?
)
```

Consuming an API (cont.)

- The variable name needs to match the name in the JSON



```
data class Art(
    @SerializedName("data")
    val pieces: List<ArtPiece>
)

data class ArtPiece(
    val id: String,
    val title: String,
    @SerializedName("image_id")
    val image: String?
)
```

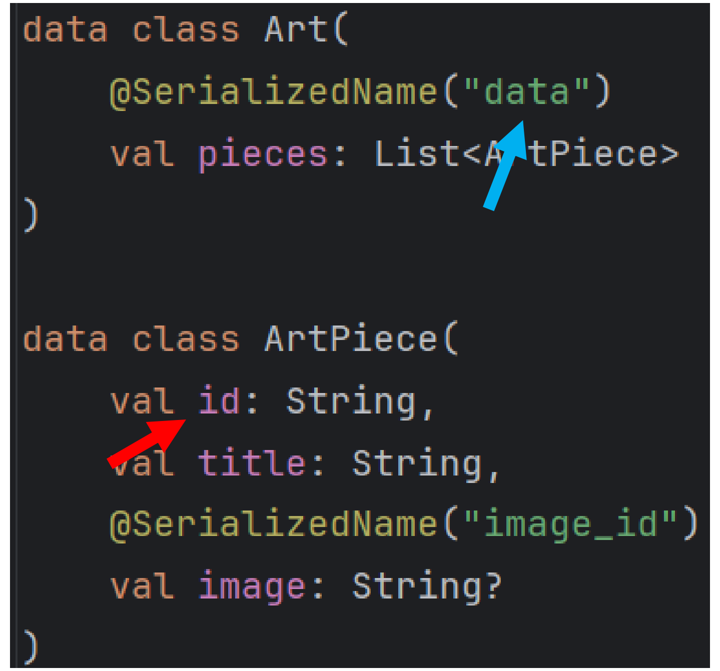


```
d"},"data":[{"id":270374,
},{"id":21662,"title":"C
900-2e55bcad0d94"},{"id"
af1"},{"id":8962,"title"
```

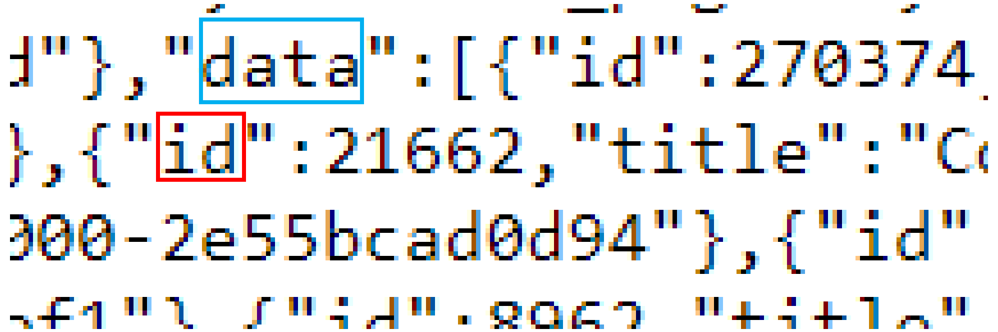
- If you want to have a different variable name, then use *@SerializedName* and provide the name that matches the JSON

调用 API（续）

- 变量名 必须与 JSON 中的名称匹配需要与 JSON 中的名称一致



```
data class Art(
    @SerializedName("data")
    val pieces: List<ArtPiece>
)

data class ArtPiece(
    val id: String,
    val title: String,
    @SerializedName("image_id")
    val image: String?
)
```



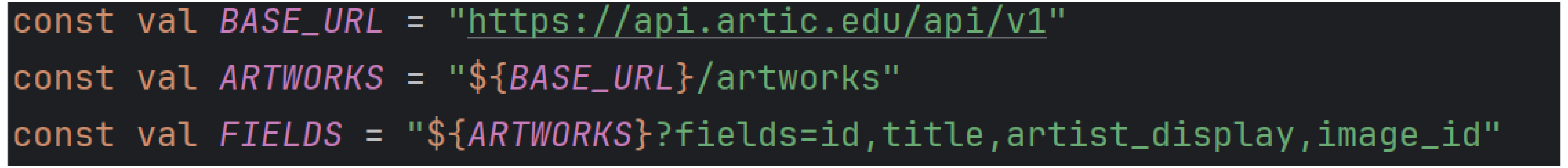
```
d"},"data":[{"id":270374,
},{"id":21662,"title":"C
900-2e55bcad0d94"},{"id"
af1"},{"id":8962,"title"
```

- 如果希望使用不同的变量名，则应使用 *@SerializedName* 并提供与 JSON 中匹配的名称

Consuming an API (cont.)

- Sometimes you have multiple endpoints
  - Endpoints are URLs that represent specific resources or actions in an API

- Its good practise to make your endpoints constants

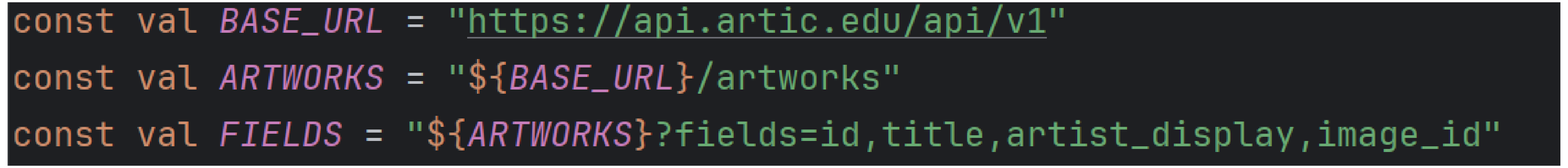


```kotlin
const val BASE_URL = "https://api.artic.edu/api/v1"
const val ARTWORKS = "${BASE_URL}/artworks"
const val FIELDS = "${ARTWORKS}?fields=id,title,artist_display,image_id"
```

调用 API（续）

- 有时你会有多个端点
  - 端点是代表 API 中特定资源或操作的 URL

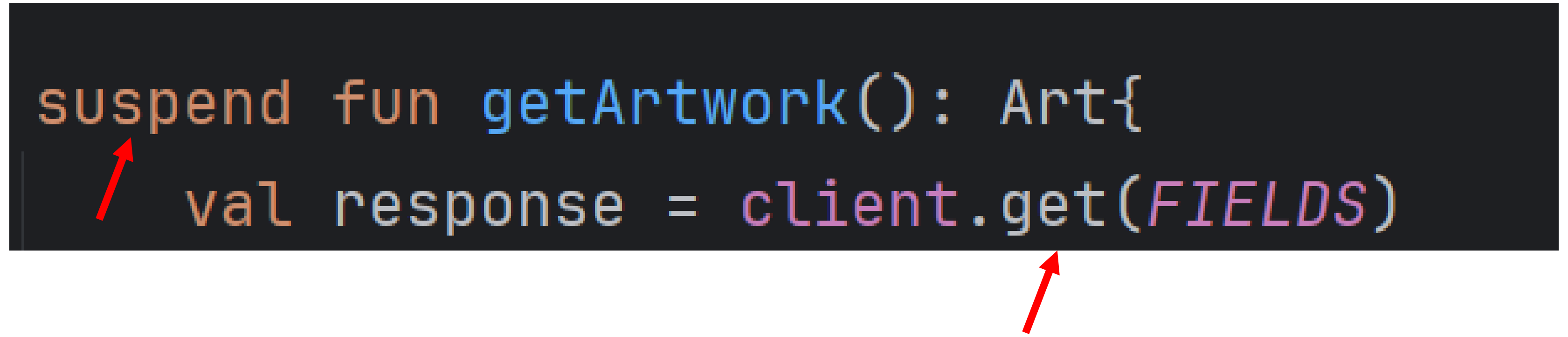
- 将端点设为常量是一种良好的实践



```kotlin
const val BASE_URL = "https://api.artic.edu/api/v1"
const val ARTWORKS = "${BASE_URL}/artworks"
const val FIELDS = "${ARTWORKS}?fields=id,title,artist_display,image_id"
```

Consuming an API (cont.)

- We can now call the HTTP request GET
  - Since this is data access logic, we should put it in a Repository

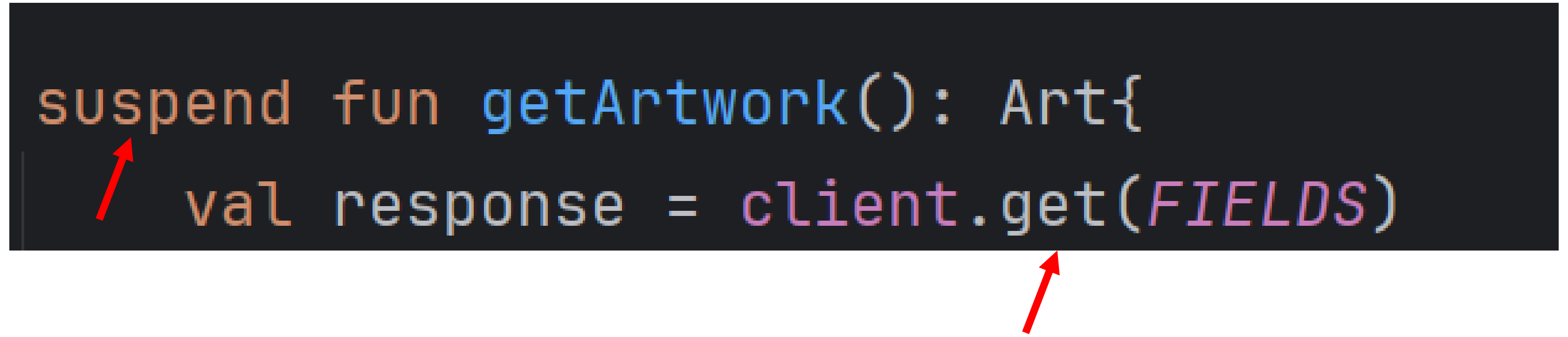


```
suspend fun getArtwork(): Art{
    val response = client.get(FIELDS)
```

- Ktor's *client.get* is a suspend function so we need to put this in another suspend function

调用 API（续）

- 我们现在可以调用 HTTP 请求 GET
  - 由于这是数据访问逻辑，我们应该将其放入仓库中

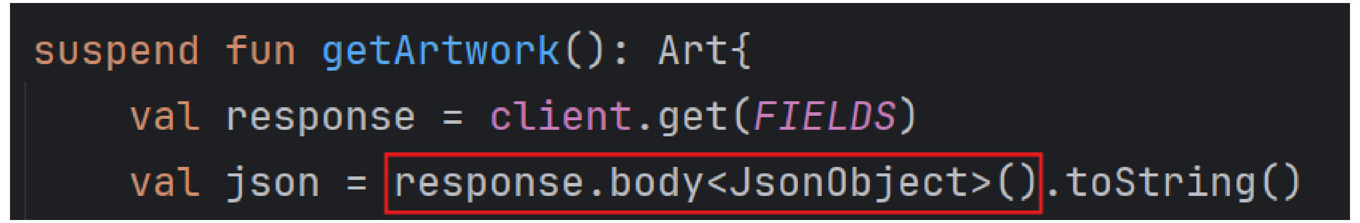


```
suspend fun getArtwork(): Art{
    val response = client.get(FIELDS)
```

- Ktor 的 *client.get* 是一个挂起函数，因此我们需要将其放在另一个挂起函数中

Consuming an API (cont.)

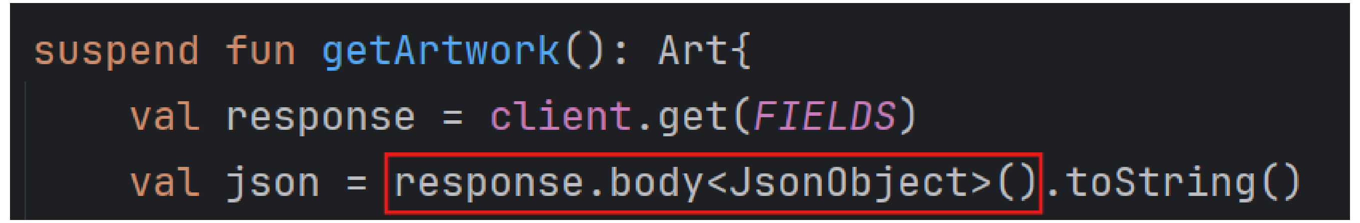
- Next, we will make use of GSON
- The *HttpResponse.body* provides us with the Json in type *JsonObject*



```
suspend fun getArtwork(): Art{
    val response = client.get(FIELDS)
    val json = response.body<JsonObject>().toString()
```

调用 API（续）

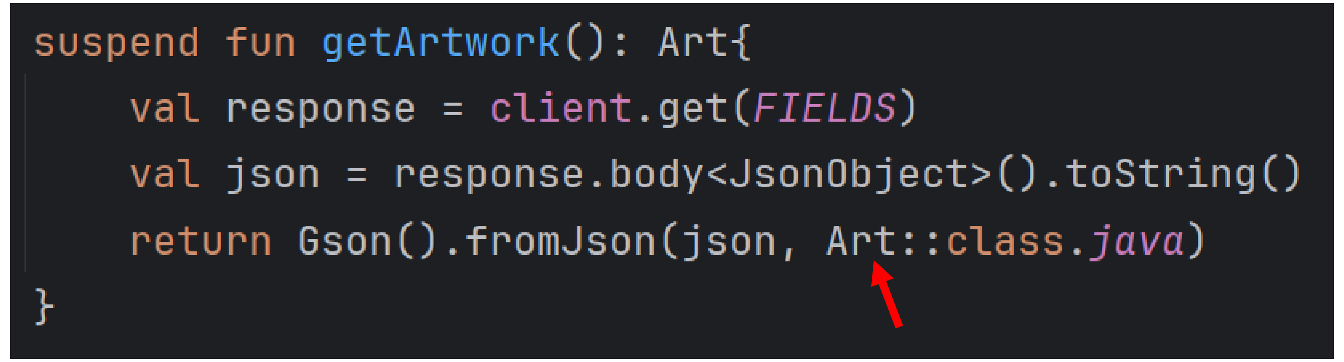
- 接下来，我们将使用 GSON
- 其中 *HttpResponse.body* 向我们提供了类型为 Json 的 *JsonObject*



```
suspend fun getArtwork(): Art{
    val response = client.get(FIELDS)
    val json = response.body<JsonObject>().toString()
```

Consuming an API (cont.)

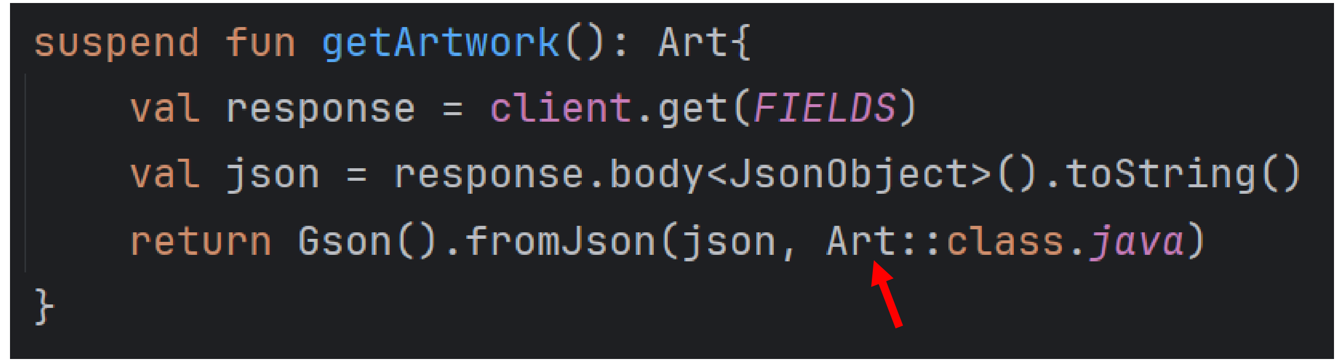
- Here we are using GSON to deserialize the Json object into a new instance of our data model



```
suspend fun getArtwork(): Art{
    val response = client.get(FIELDS)
    val json = response.body<JsonObject>().toString()
    return Gson().fromJson(json, Art::class.java)
}
```

调用 API（续）

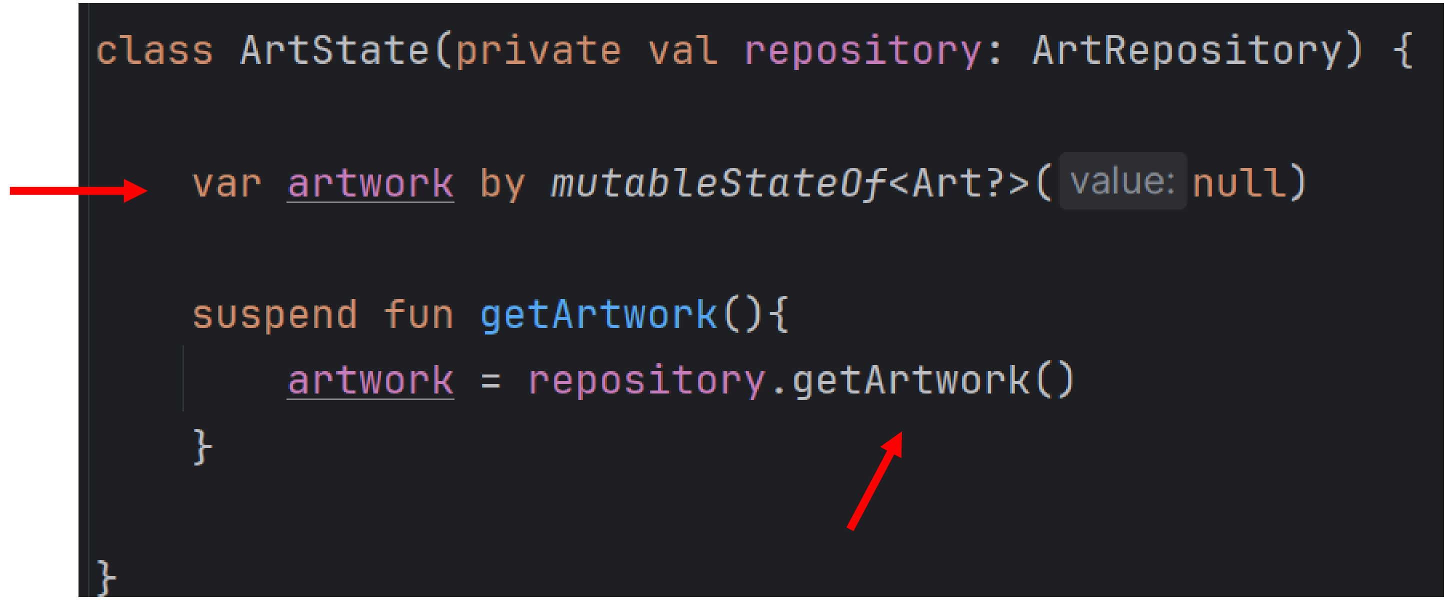
- 这里我们使用 GSON 将 Json 对象反序列化为一个新的我们的 数据模型 实例



```
suspend fun getArtwork(): Art{
    val response = client.get(FIELDS)
    val json = response.body<JsonObject>().toString()
    return Gson().fromJson(json, Art::class.java)
}
```

Consuming an API (cont.)

- Once our repository is set up, we can set up our state holder

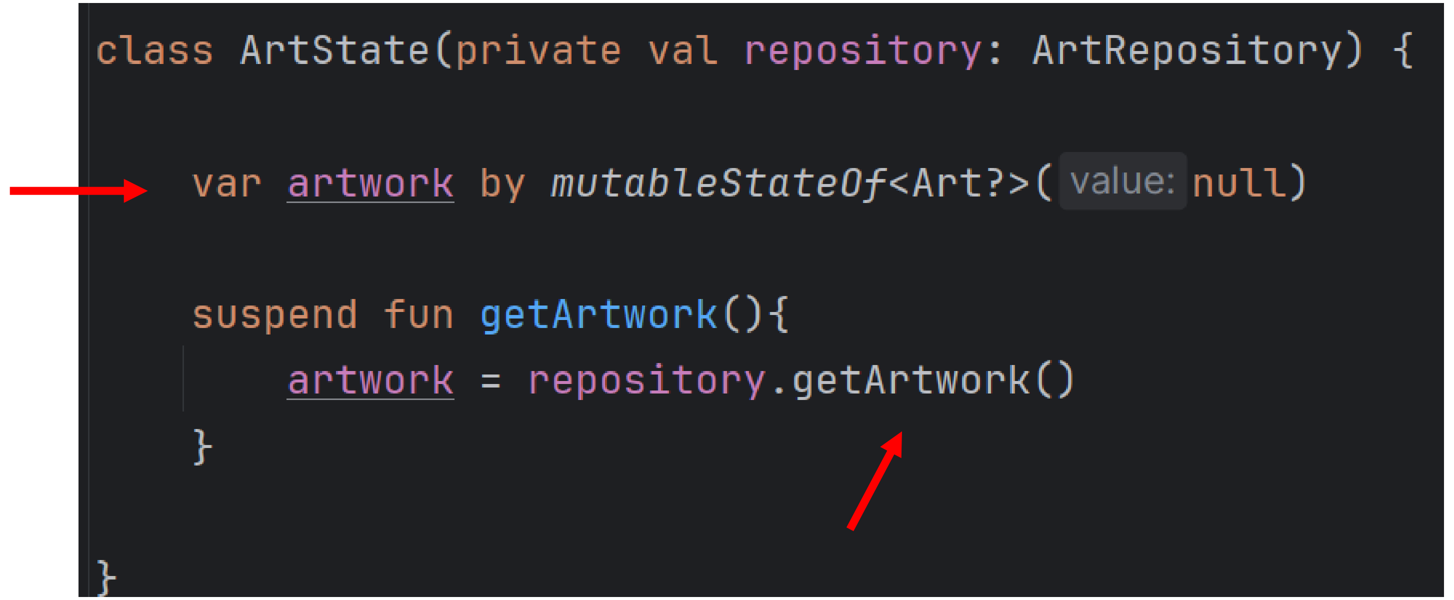


```
class ArtState(private val repository: ArtRepository) {

    var artwork by mutableStateOf<Art?>( value: null)

    suspend fun getArtwork(){
        artwork = repository.getArtwork()
    }

}
```

- *getArtwork* is suspending so it needs to go in another suspend function

调用 API（续）

- 一旦我们的仓库设置完成，就可以设置状态持有者

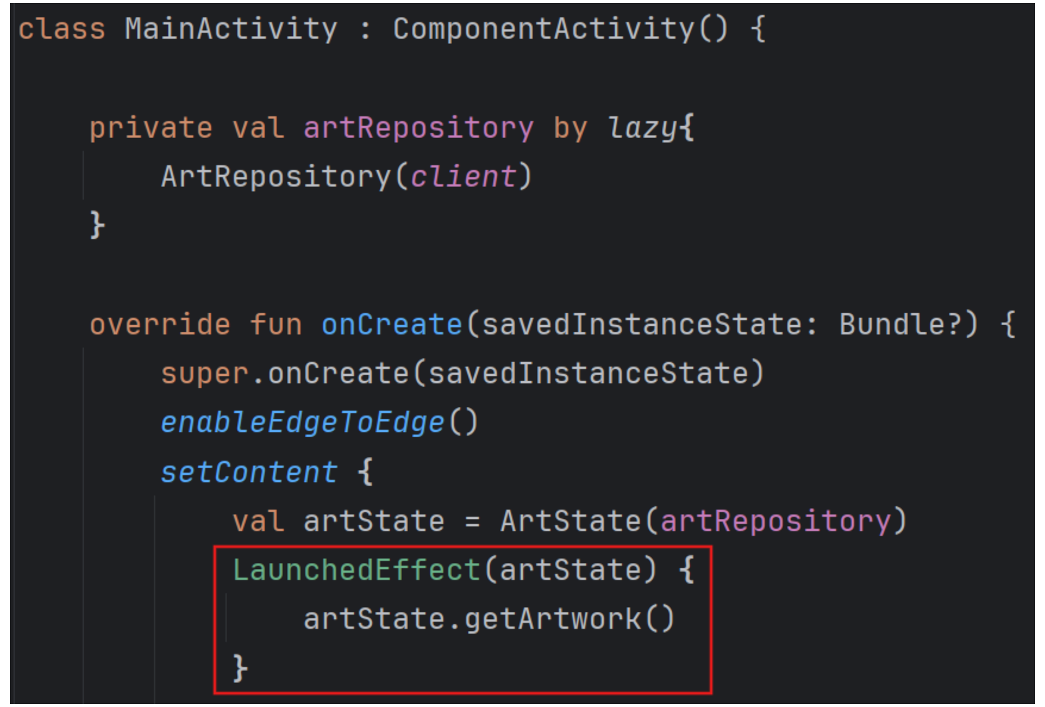


```
class ArtState(private val repository: ArtRepository) {

    var artwork by mutableStateOf<Art?>( value: null)

    suspend fun getArtwork(){
        artwork = repository.getArtwork()
    }

}
```

- *getArtwork* 是 挂起的，因此它需要放在另一个挂起函数中

Consuming an API (cont.)

- LaunchedEffect is a composable AND a coroutine
  - Useful for running suspend functions in the scope of a composable



```
class MainActivity : ComponentActivity() {

    private val artRepository by lazy{
        ArtRepository(client)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            val artState = ArtState(artRepository)
            LaunchedEffect(artState) {
                artState.getArtwork()
            }
```

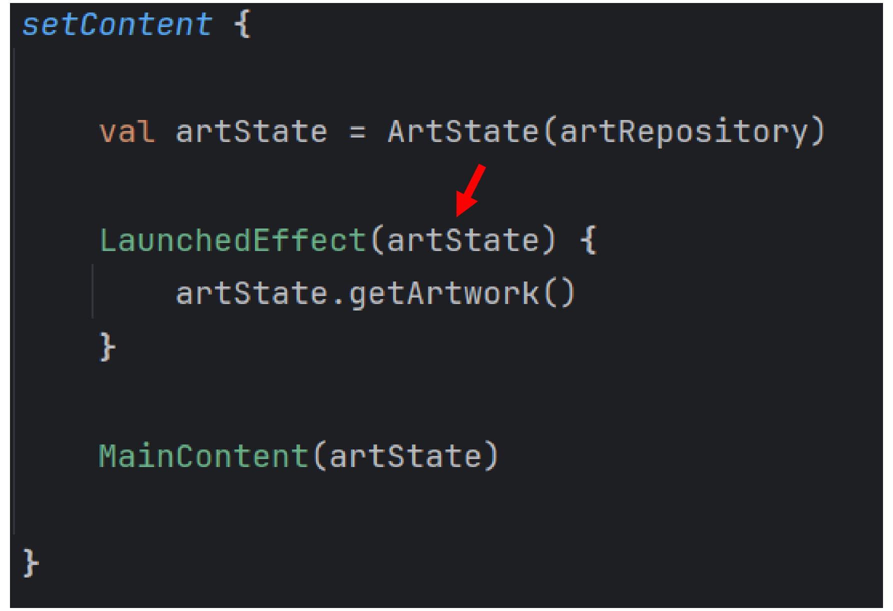
调用 API（续）

- LaunchedEffect 既是可组合函数，也是一个协程
  - 适用于在可组合函数的作用域内运行挂起函数

```
class MainActivity : ComponentActivity() {

    private val artRepository by lazy{
        ArtRepository(client)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            val artState = ArtState(artRepository)
            LaunchedEffect(artState) {
                artState.getArtwork()
            }
```

Consuming an API (cont.)

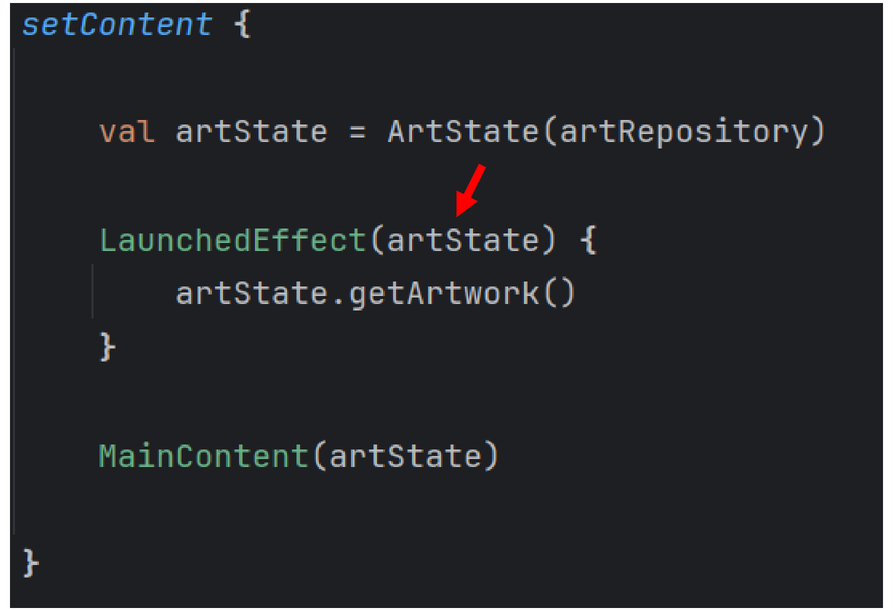
- *LaunchedEffect* will re-launch if its key changes
- *LaunchedEffect* will not re-launch if recomposed and the key doesn’t change



```
setContent {

    val artState = ArtState(artRepository)

    LaunchedEffect(artState) {
        artState.getArtwork()
    }

    MainContent(artState)

}
```

调用 API（续）

- *LaunchedEffect* 将在 key 发生变化时重新启动
- *LaunchedEffect* 在重组时不会重新启动，如果 key 没有变化

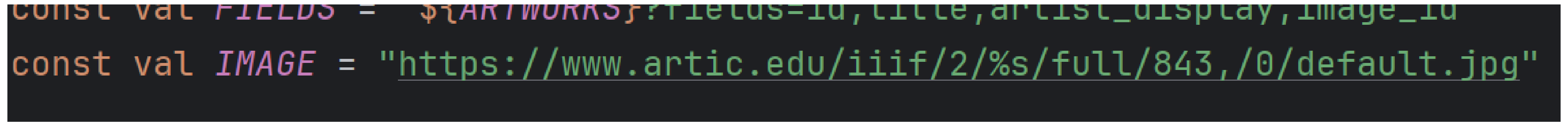


```
setContent {

    val artState = ArtState(artRepository)

    LaunchedEffect(artState) {
        artState.getArtwork()
    }

    MainContent(artState)

}
```

Consuming an API (cont.)

- If your API provides images, you first need to obtain the correct URL

- For the Chicago Art institute API, the format is below
  - Docs: https://api.artic.edu/docs/#images

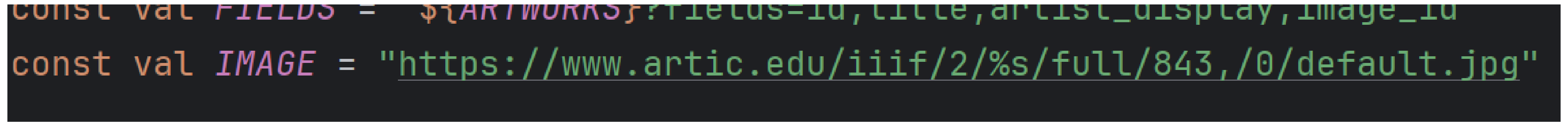


```
const val FIELDS = "${ARTWORKS}?fields=id,title,artist_display,image_id
const val IMAGE = "https://www.artic.edu/iiif/2/%s/full/843,/0/default.jpg"
```

调用 API（续）

- 如果您的 API 提供图像，您首先需要获取正确的 URL

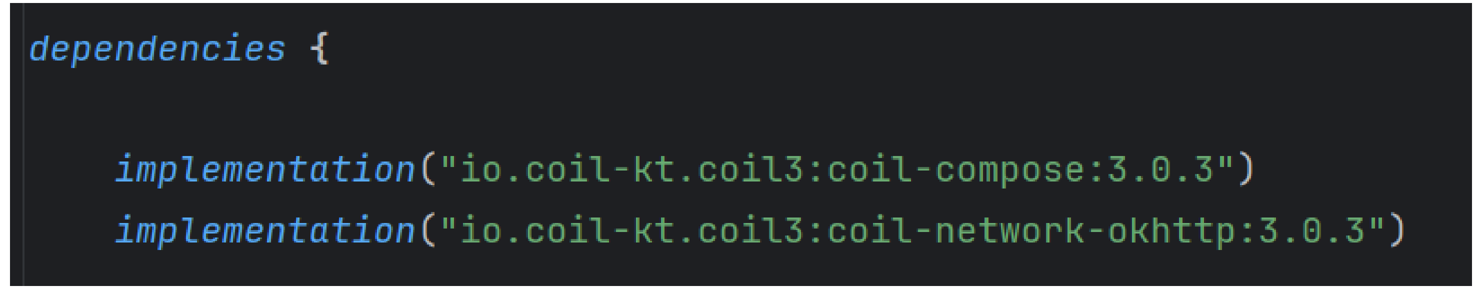
- 对于芝加哥艺术学院的 API，格式如下
  - 文档：https://api.artic.edu/docs/#images



```
const val FIELDS = "${ARTWORKS}?fields=id,title,artist_display,image_id
const val IMAGE = "https://www.artic.edu/iiif/2/%s/full/843,/0/default.jpg"
```

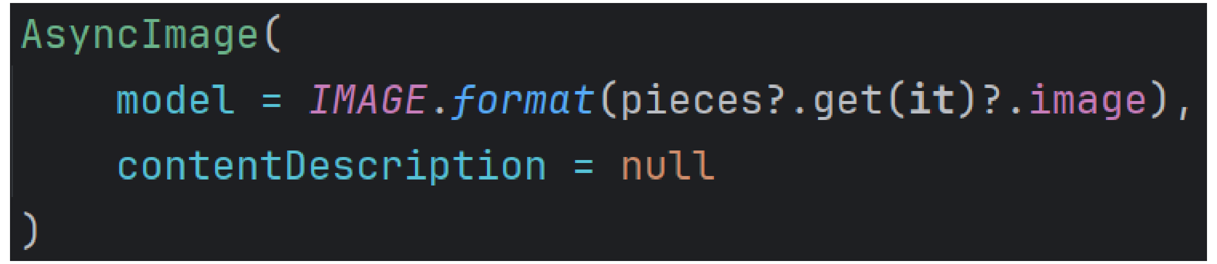
Coil

- Coil is an image loader library for jetpack compose



```
dependencies {

    implementation("io.coil-kt.coil3:coil-compose:3.0.3")
    implementation("io.coil-kt.coil3:coil-network-okhttp:3.0.3")
```

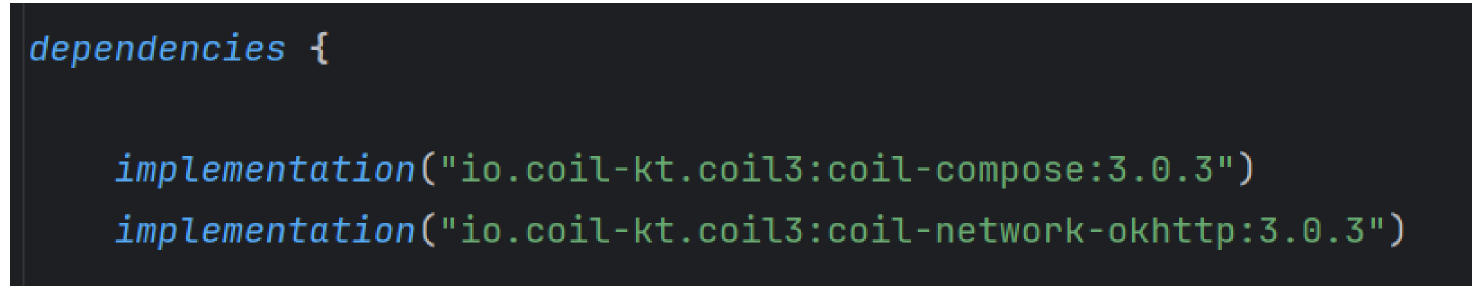
- It makes our life easy



```
AsyncImage(
    model = IMAGE.format(pieces?.get(it)?.image),
    contentDescription = null
)
```

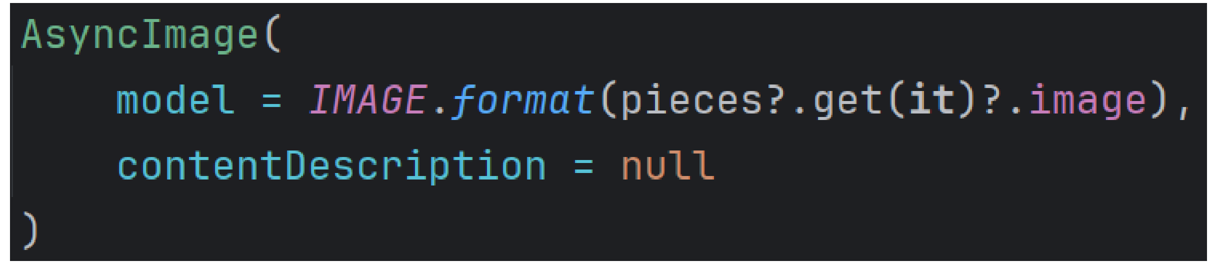
Coil

- Coil 是一个用于 Jetpack Compose 的图像加载库



```
dependencies {

    implementation("io.coil-kt.coil3:coil-compose:3.0.3")
    implementation("io.coil-kt.coil3:coil-network-okhttp:3.0.3")
```

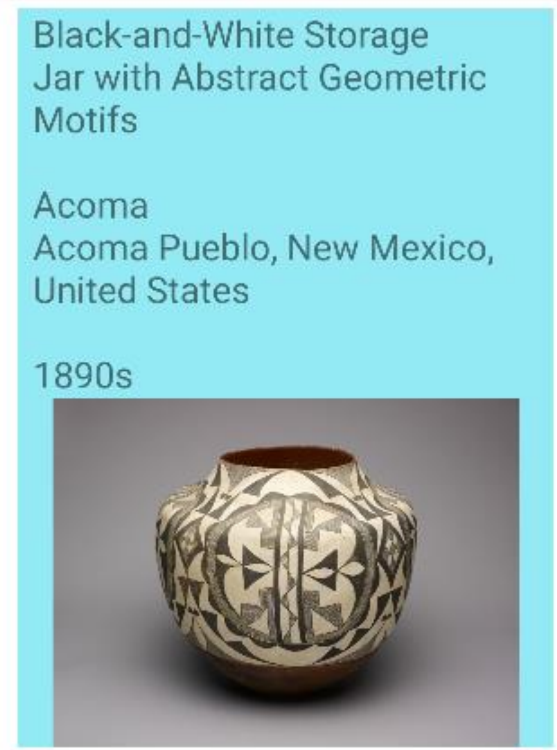
- 它让我们的生活变得更加轻松



```
AsyncImage(
    model = IMAGE.format(pieces?.get(it)?.image),
    contentDescription = null
)
```

Consuming an API (cont.)

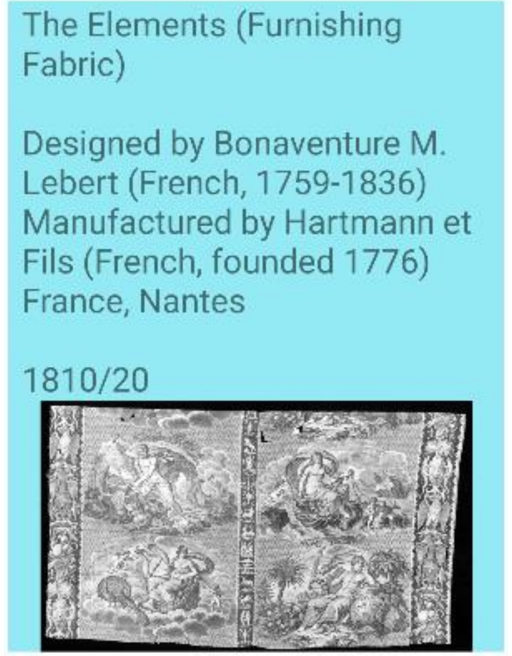
- Display your data however you want



Black-and-White Storage Jar with Abstract Geometric Motifs

Acoma
Acoma Pueblo, New Mexico, United States

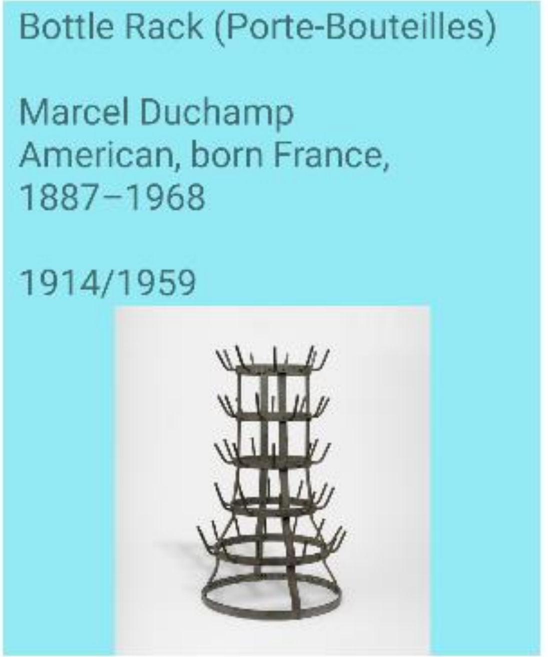
1890s



The Elements (Furnishing Fabric)

Designed by Bonaventure M. Lebert (French, 1759-1836)
Manufactured by Hartmann et Fils (French, founded 1776)
France, Nantes

1810/20



Bottle Rack (Porte-Bouteilles)

Marcel Duchamp
American, born France, 1887–1968

1914/1959

调用 API（续）

- 以任意方式展示您的数据

Black-and-White Storage Jar with Abstract Geometric Motifs

Acoma
Acoma Pueblo, New Mexico, United States

1890s

The Elements (Furnishing Fabric)

Designed by Bonaventure M. Lebert (French, 1759-1836)
Manufactured by Hartmann et Fils (French, founded 1776)
France, Nantes

1810/20

Bottle Rack (Porte-Bouteilles)

Marcel Duchamp
American, born France, 1887–1968

1914/1959