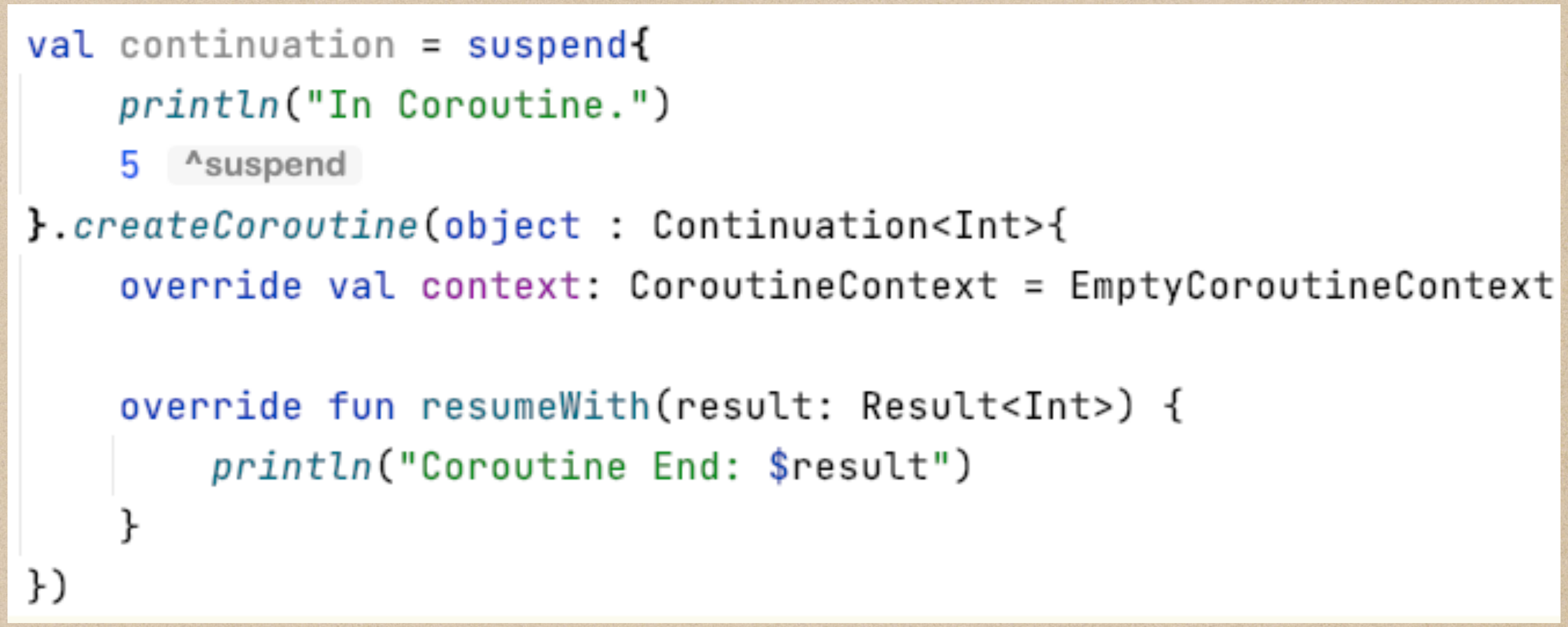
# Coroutine

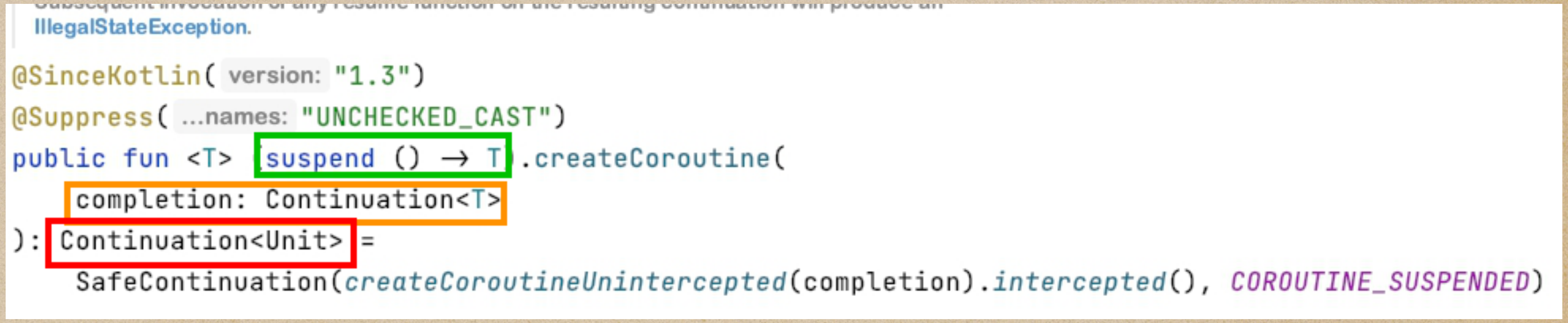
Kotlin協程的基礎設施

# 建立一個Coroutine

- Receiver是一個suspend修飾的掛起函數，稱為協程體
- completion是Coroutine的完成Callback
- 會返回一個Continuation



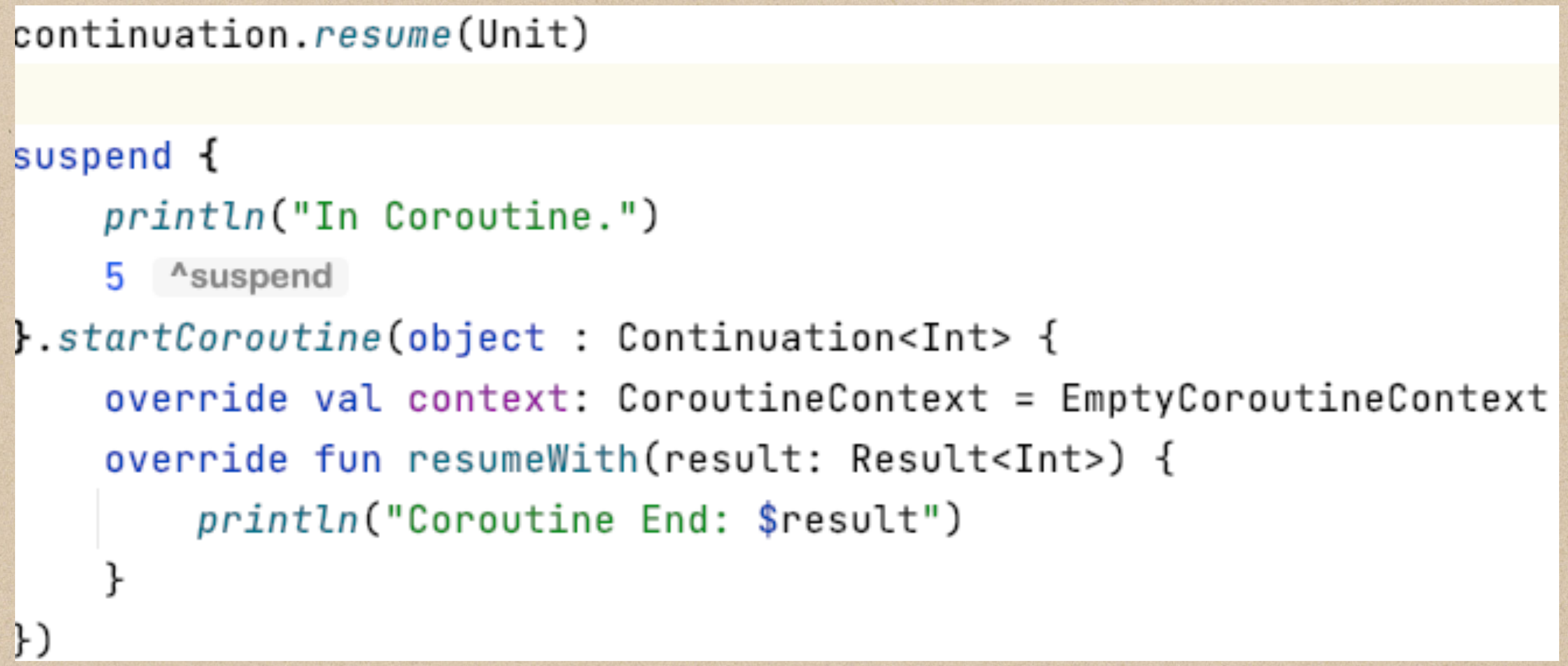
```kotlin
val continuation = suspend{
    println("In Coroutine.")
    5 ^suspend
}.createCoroutine(object : Continuation<Int>{
    override val context: CoroutineContext = EmptyCoroutineContext

    override fun resumeWith(result: Result<Int>) {
        println("Coroutine End: $result")
    }
})
```



```kotlin
    Subsequent invocation of any resume function on the resulting continuation will produce an
    IllegalStateException.
@SinceKotlin( version: "1.3")
@Suppress( ...names: "UNCHECKED_CAST")
public fun <T> (suspend () → T).createCoroutine(
    completion: Continuation<T>
): Continuation<Unit> =
    SafeContinuation(createCoroutineUnintercepted(completion).intercepted(), COROUTINE_SUSPENDED)
```

# 啟動一個Coroutine

- 可以使用rusume恢復
- 也有提供建立後立刻開始的語法
- Suspend Function會在編譯時轉SuspendLambda

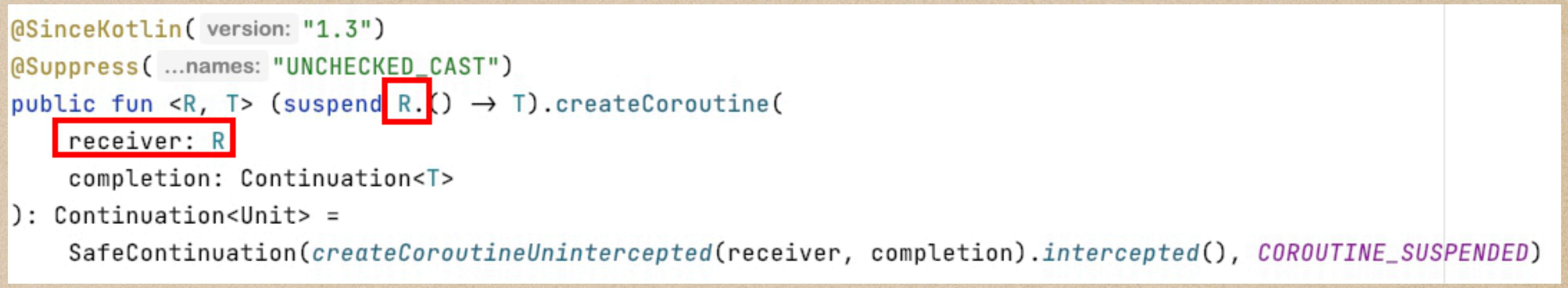


```
continuation.resume(Unit)

suspend {
    println("In Coroutine.")
    5  ^suspend
}.startCoroutine(object : Continuation<Int> {
    override val context: CoroutineContext = EmptyCoroutineContext
    override fun resumeWith(result: Result<Int>) {
        println("Coroutine End: $result")
    }
})
```

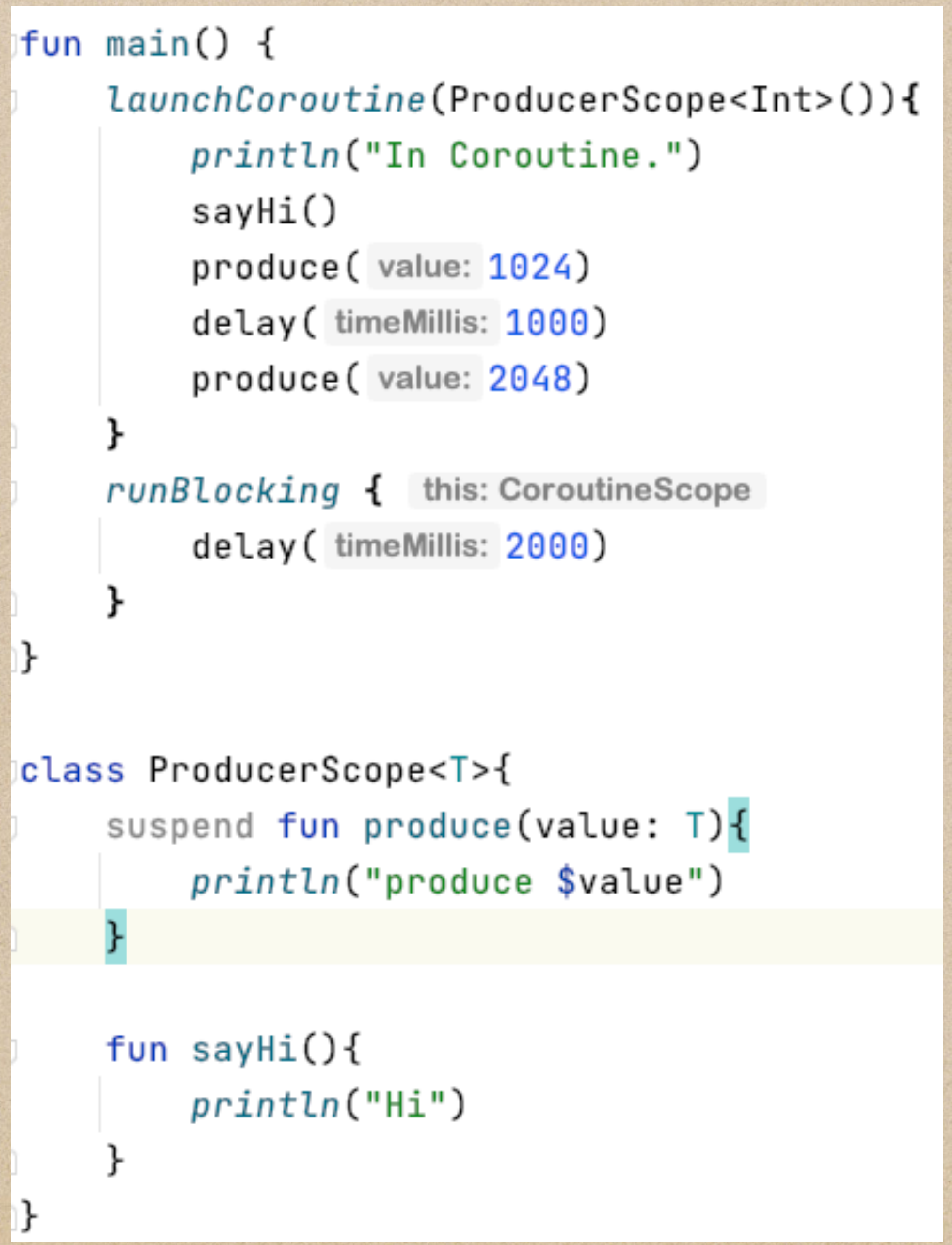
# Receiver

- 藉由Receiver，可以提供一個作用域，可以直接使用作用域內的函數或狀態
- 加上了RestrictsSuspension則能增加限制

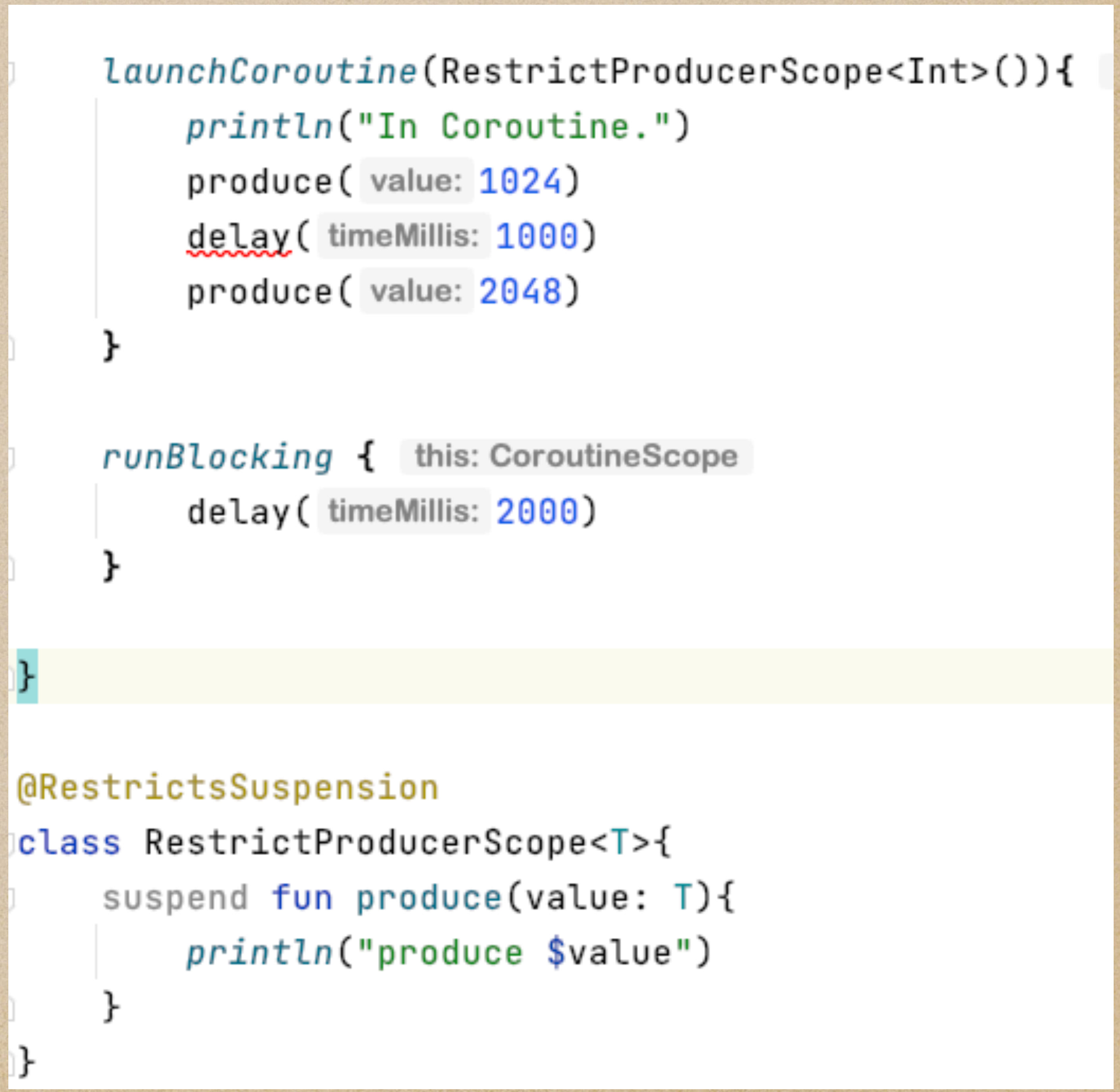
  可以用來避免無效或危險的呼叫



```kotlin
@SinceKotlin( version: "1.3")
@Suppress( ...names: "UNCHECKED_CAST")
public fun <R, T> (suspend R.() → T).createCoroutine(
    receiver: R
    completion: Continuation<T>
): Continuation<Unit> =
    SafeContinuation(createCoroutineUnintercepted(receiver, completion).intercepted(), COROUTINE_SUSPENDED)
```



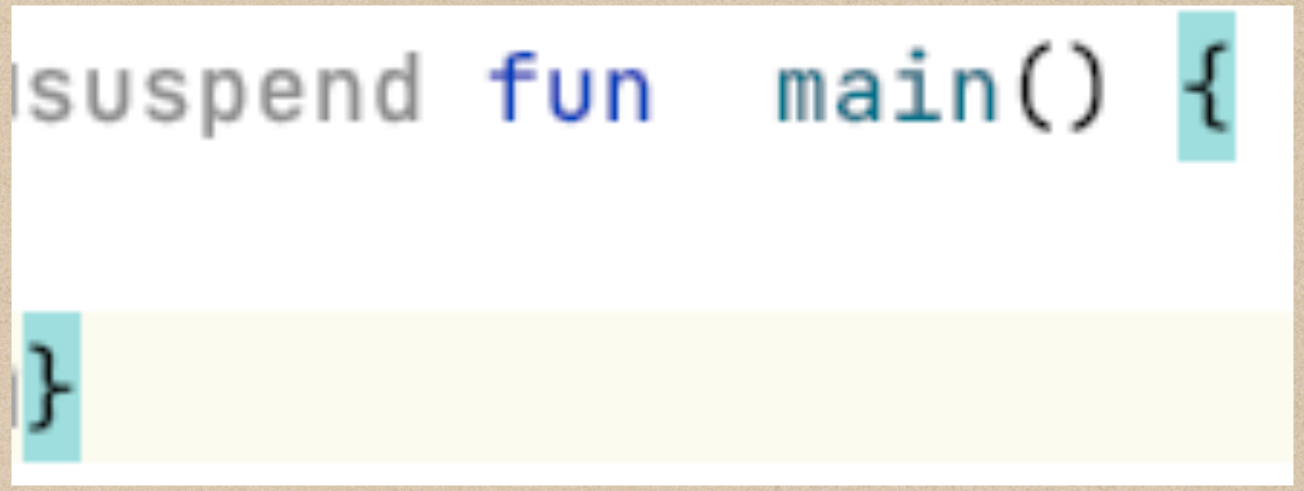
```kotlin
fun main() {
    launchCoroutine(ProducerScope<Int>()){
        println("In Coroutine.")
        sayHi()
        produce( value: 1024)
        delay( timeMillis: 1000)
        produce( value: 2048)
    }
    runBlocking {  this: CoroutineScope
        delay( timeMillis: 2000)
    }
}

class ProducerScope<T>{
    suspend fun produce(value: T){
        println("produce $value")
    }

    fun sayHi(){
        println("Hi")
    }
}
```



```kotlin
    launchCoroutine(RestrictProducerScope<Int>()){
        println("In Coroutine.")
        produce( value: 1024)
        delay( timeMillis: 1000)
        produce( value: 2048)
    }

    runBlocking {  this: CoroutineScope
        delay( timeMillis: 2000)
    }

}

@RestrictsSuspension
class RestrictProducerScope<T>{
    suspend fun produce(value: T){
        println("produce $value")
    }
}
```

# Suspend Main

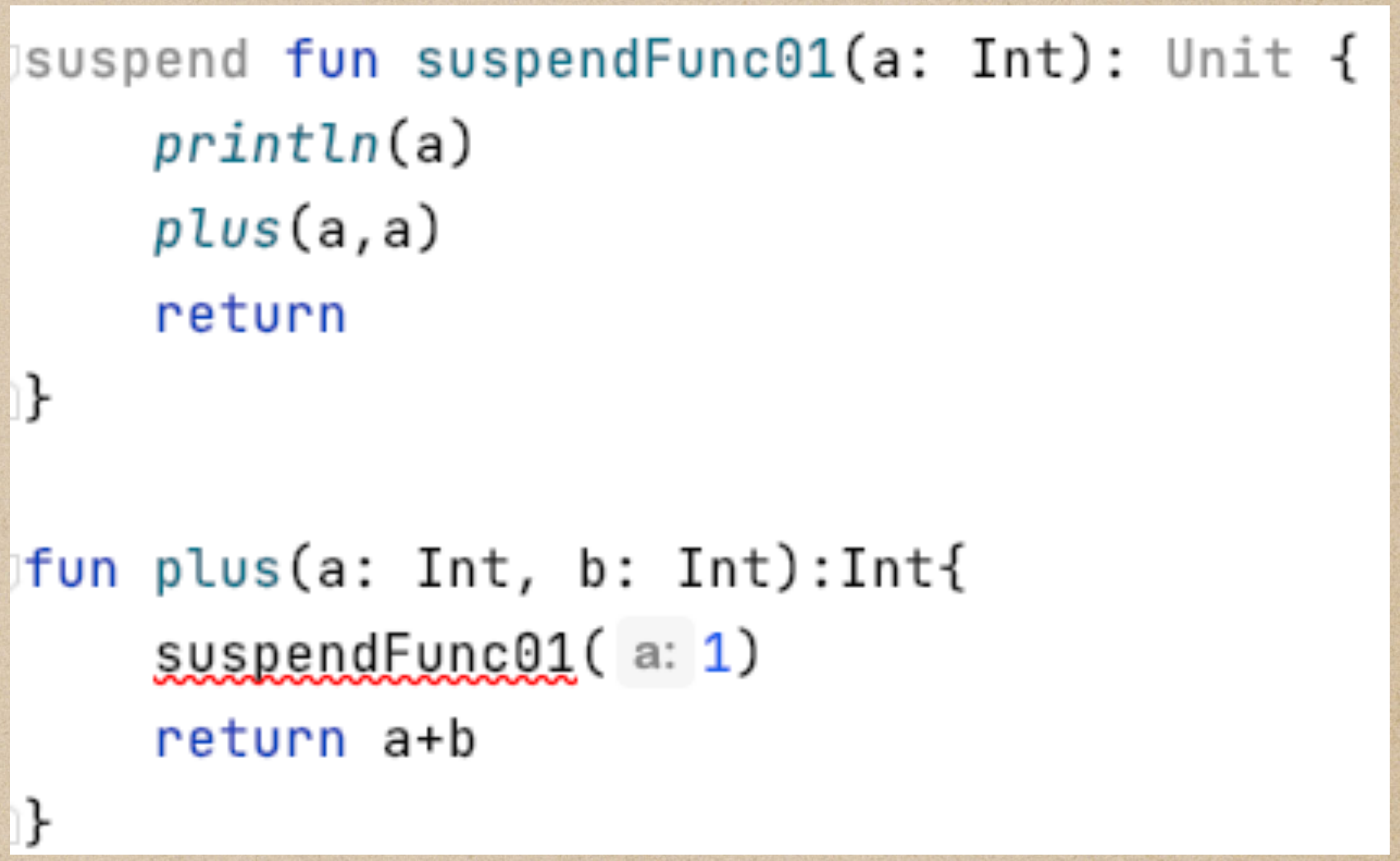
- 1.3版Kotlin後，main可以被宣告為掛起函數
- 在編譯時，Kotlin會生成一個真正的main，並且呼叫runSuspend來執行suspend main



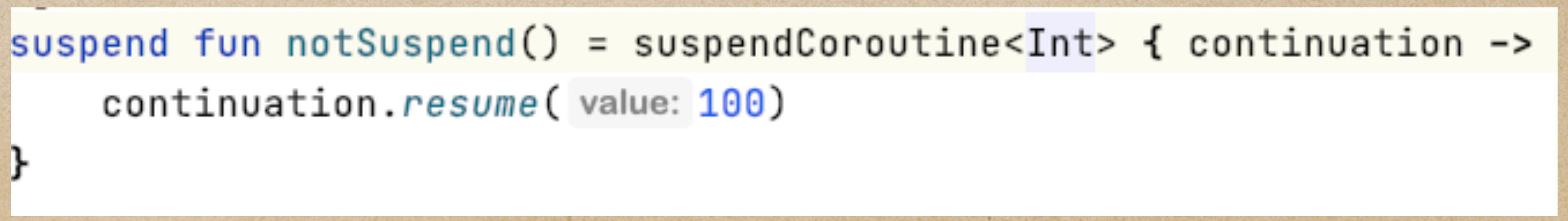
```
suspend fun  main() {

}
```

# Suspend Function

- Suspend Function可以呼叫任何Function
  一般Function只能呼叫一般Function
- 所謂的Coroutine掛起，指執行流程發生異步調用時，當前流程進入等待狀態
- 執行suspend function不一定會suspend



```
suspend fun suspendFunc01(a: Int): Unit {
    println(a)
    plus(a,a)
    return
}

fun plus(a: Int, b: Int):Int{
    suspendFunc01( a: 1)
    return a+b
}
```



```
suspend fun notSuspend() = suspendCoroutine<Int> { continuation ->
    continuation.resume( value: 100)
}
```

## 掛起點

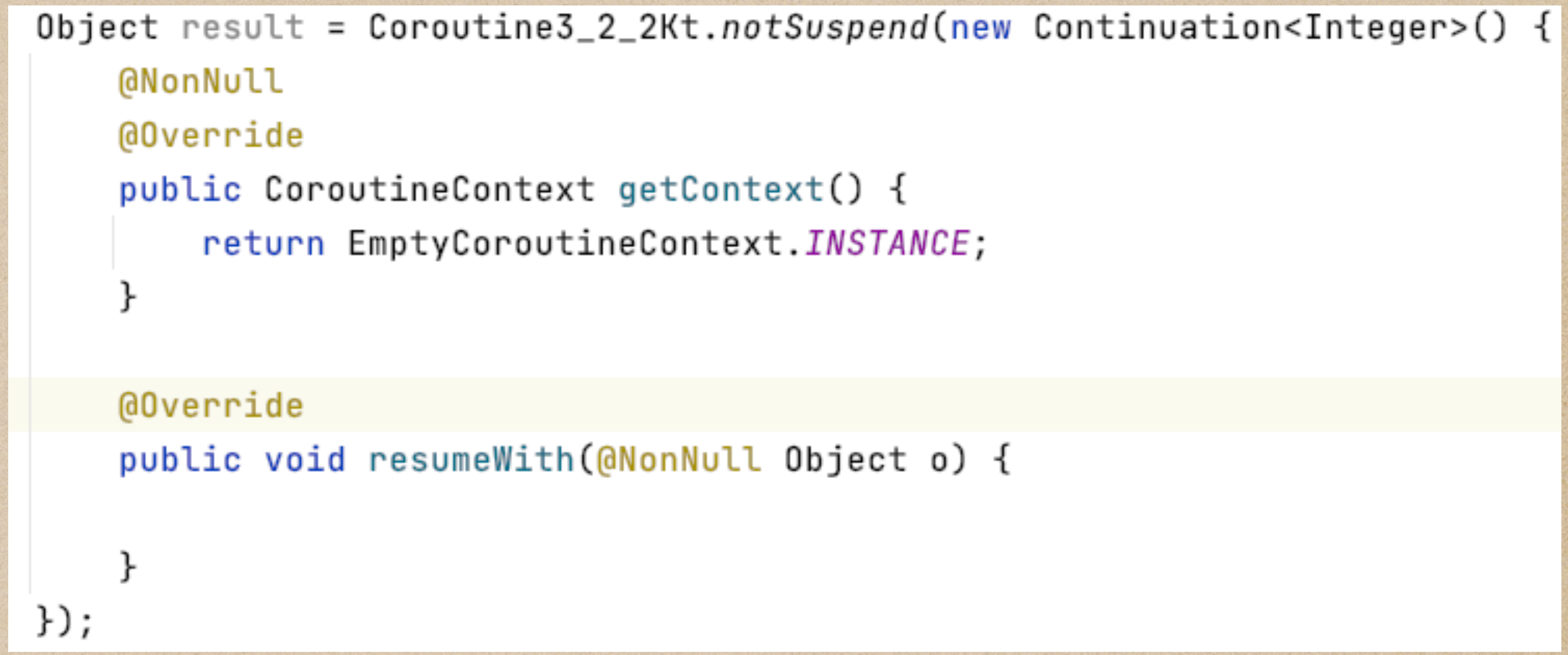
- 需要一個Continuation才能夠掛起
- 出現異步調用時，就會掛起，直到對應的Continuation的resume被呼叫
- 是否發生異步調用，取決於resume函數與suspend function是否在相同的調用棧上

## CPS變換

- CPS變換，藉由傳遞Continuation來控制異步調用流程
- 掛起最重要的是保存掛起的狀態
- Kotlin把掛起點的訊息保存到了Continuation中，而要恢復只要執行其恢復
- Continuation所佔用記憶體很小

## Continuation

- Unit的Suspend Function到Java返回Object了
- 根據情況返回
  同步返回：直接返回suspend function返回
  異步返回：掛起，返回suspend標記



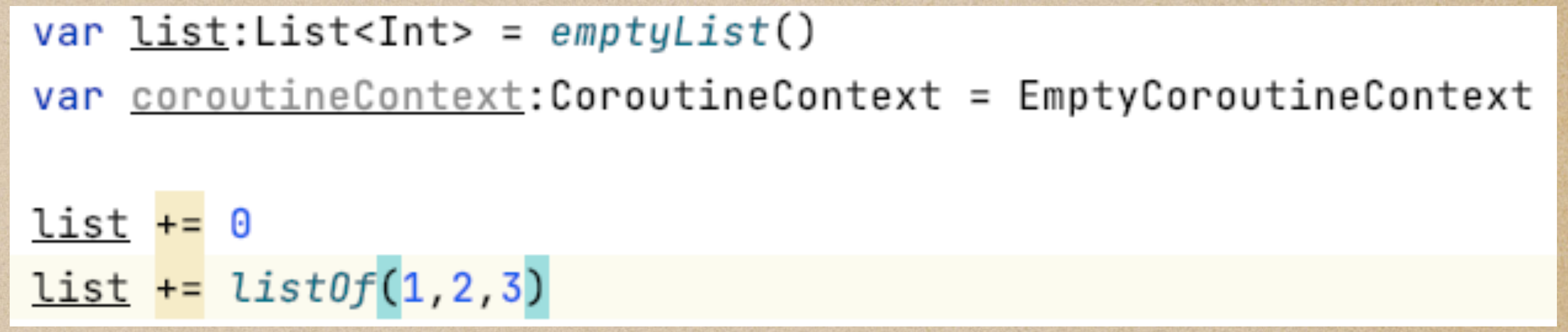
```
Object result = Coroutine3_2_2Kt.notSuspend(new Continuation<Integer>() {
    @NonNull
    @Override
    public CoroutineContext getContext() {
        return EmptyCoroutineContext.INSTANCE;
    }

    @Override
    public void resumeWith(@NonNull Object o) {

    }
});
```

## Continuation

- suspend function就是一般的function加上一個Continuation
- 因此一般的function無法跟suspend function混用

# Coroutine Context

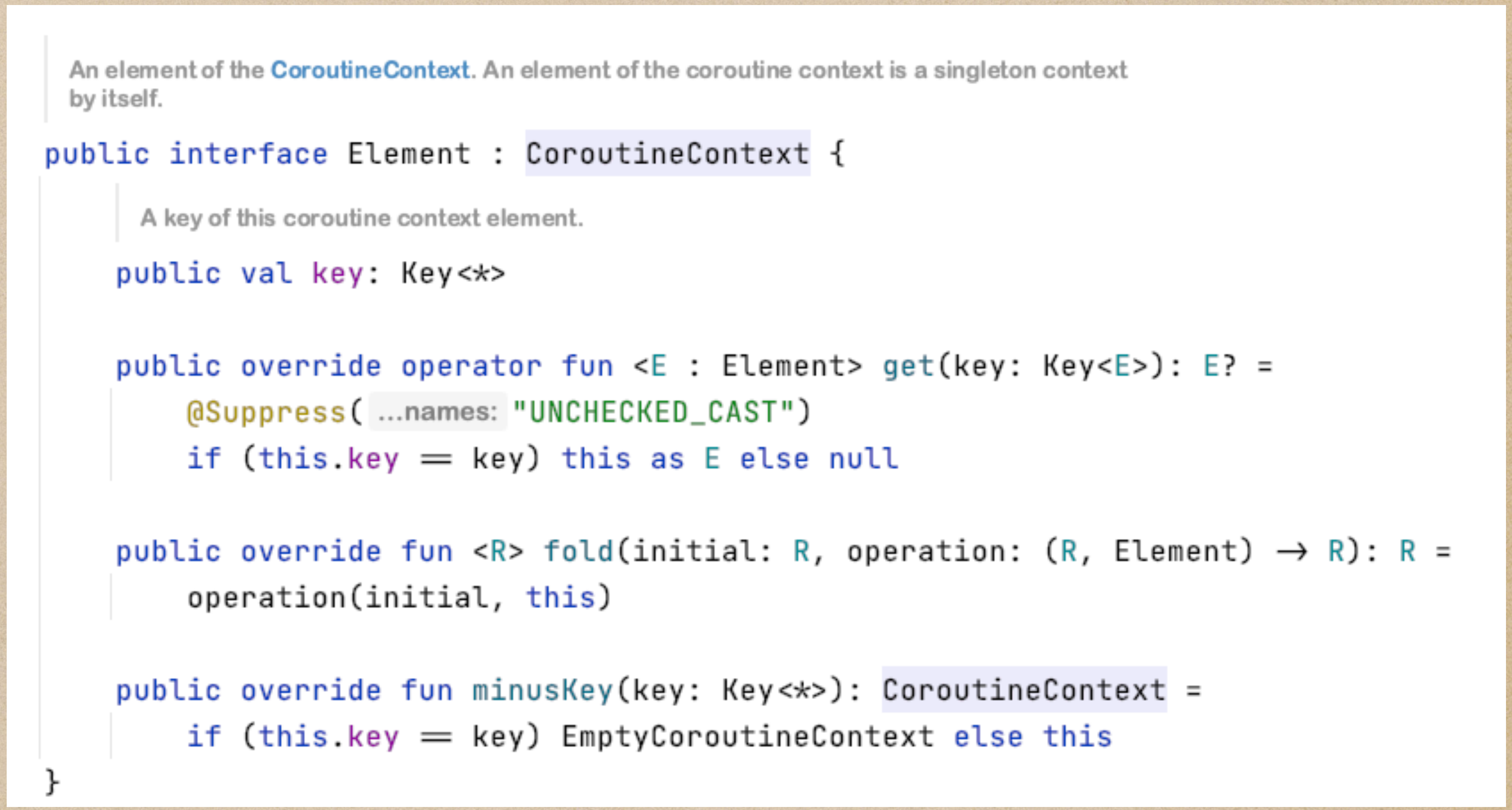
- Context乘載了資源獲取，配置管理等工作，提供執行環境相關的資源
- Coroutine的Context很類似List或是Map
- EmptyCoroutineContext是標準庫提供，表示空的Context



```kotlin
var list:List<Int> = emptyList()
var coroutineContext:CoroutineContext = EmptyCoroutineContext

list += 0
list += listOf(1,2,3)
```

## Element

- Element也實現CoroutineContext
- 藉由Key來當成索引

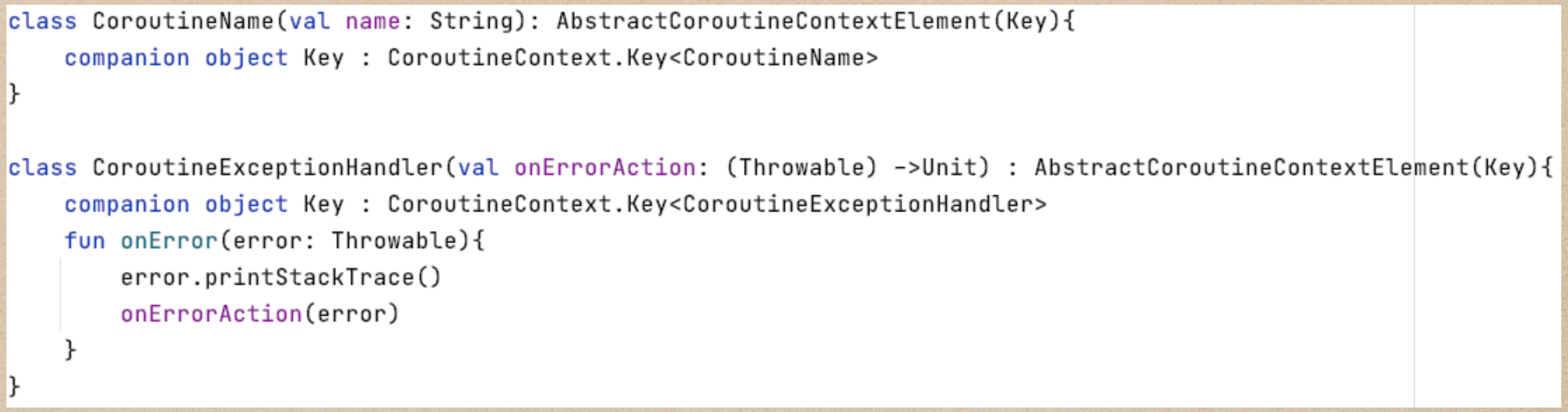


An element of the CoroutineContext. An element of the coroutine context is a singleton context by itself.

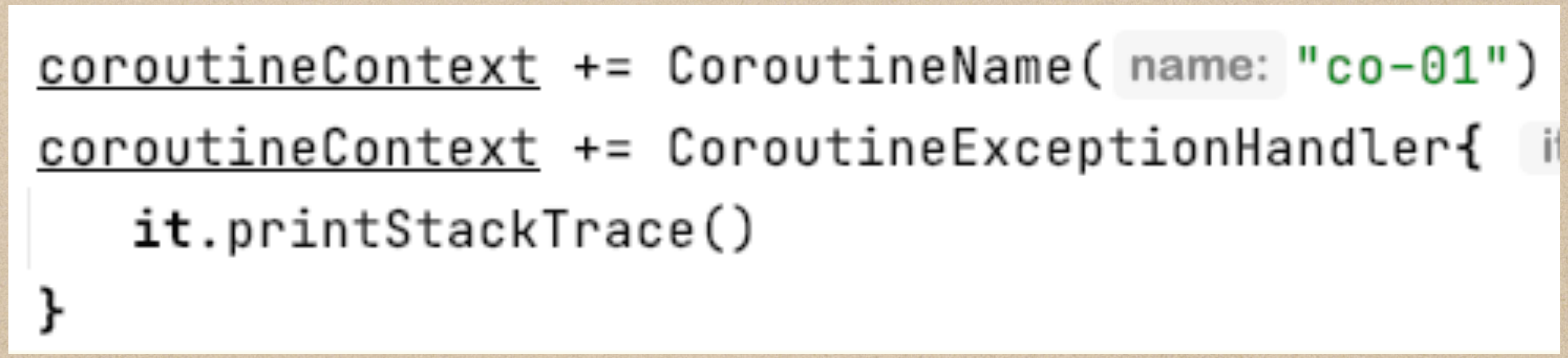
```kotlin
public interface Element : CoroutineContext {
    A key of this coroutine context element.
    public val key: Key<*>

    public override operator fun <E : Element> get(key: Key<E>): E? =
        @Suppress( ...names: "UNCHECKED_CAST")
        if (this.key == key) this as E else null

    public override fun <R> fold(initial: R, operation: (R, Element) -> R): R =
        operation(initial, this)

    public override fun minusKey(key: Key<*>): CoroutineContext =
        if (this.key == key) EmptyCoroutineContext else this
}
```

# 自訂Element

- 給定名字即可實現
- 跟List一樣使用+=
- 透過官方的coroutineContext可以直接抓到當前的Context



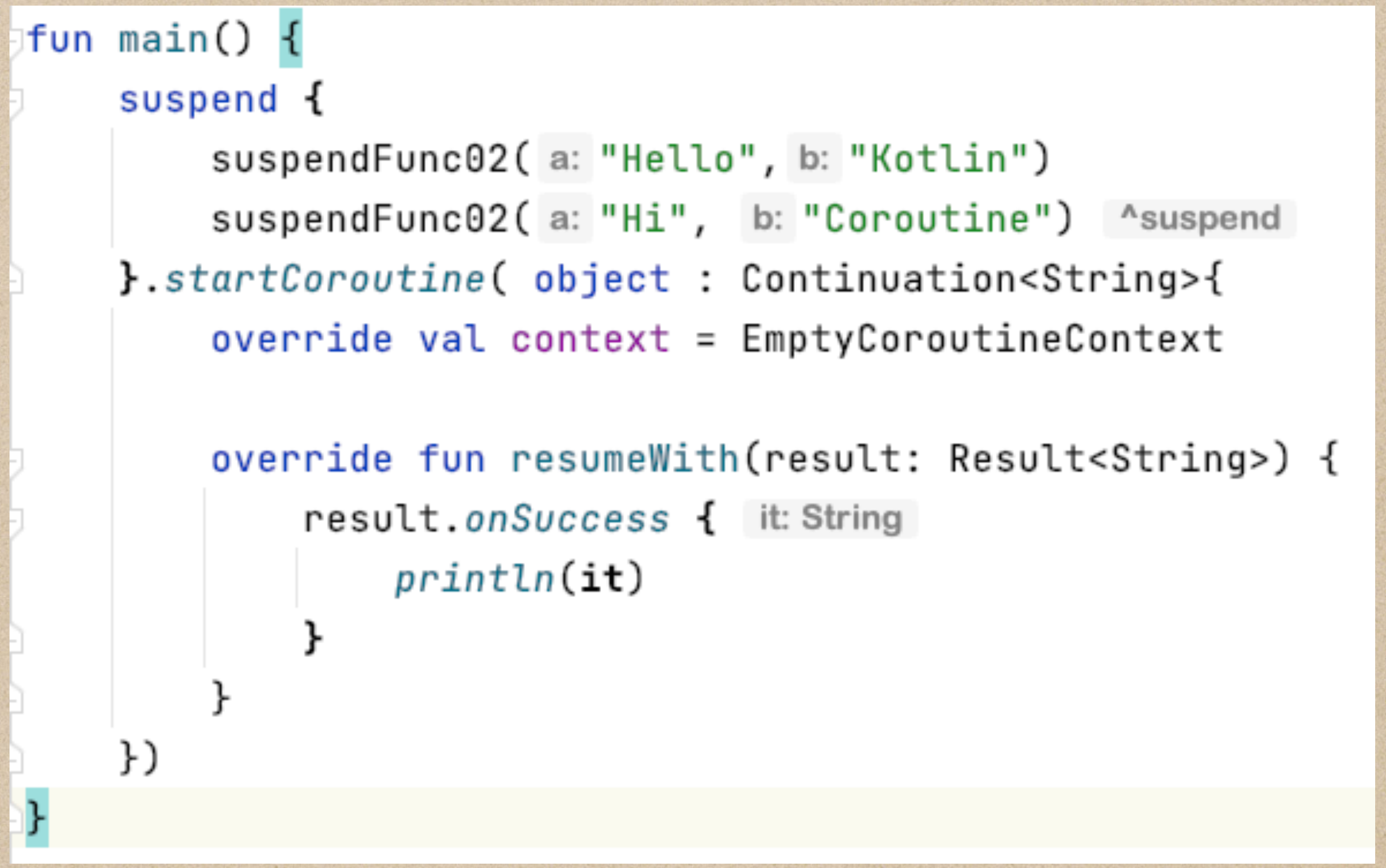
```kotlin
class CoroutineName(val name: String): AbstractCoroutineContextElement(Key){
    companion object Key : CoroutineContext.Key<CoroutineName>
}

class CoroutineExceptionHandler(val onErrorAction: (Throwable) ->Unit) : AbstractCoroutineContextElement(Key){
    companion object Key : CoroutineContext.Key<CoroutineExceptionHandler>
    fun onError(error: Throwable){
        error.printStackTrace()
        onErrorAction(error)
    }
}
```



```kotlin
coroutineContext += CoroutineName( name: "co-01")
coroutineContext += CoroutineExceptionHandler{ i
    it.printStackTrace()
}
```

# 攔截器

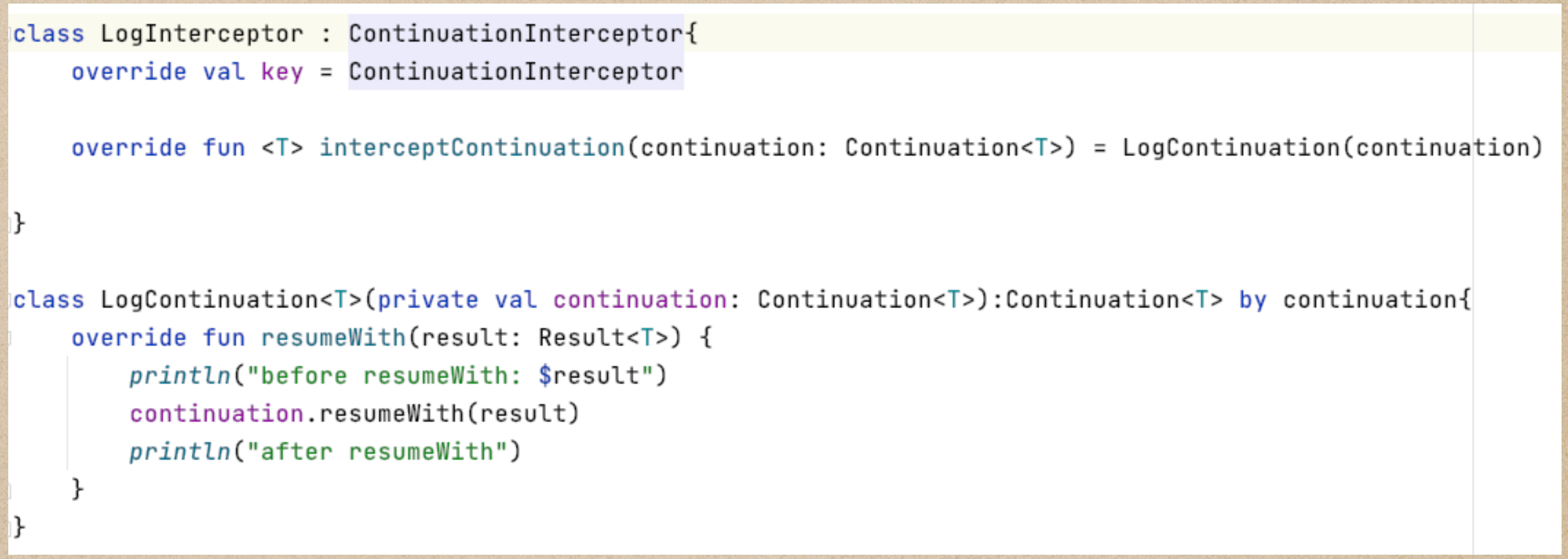
- 最常用來處理Thread切換
- 可以拿來攔截恢復調用
- 恢復調用會是n+1次



```
fun main() {
    suspend {
        suspendFunc02( a: "Hello", b: "Kotlin")
        suspendFunc02( a: "Hi",  b: "Coroutine") ^suspend
    }.startCoroutine( object : Continuation<String>{
        override val context = EmptyCoroutineContext

        override fun resumeWith(result: Result<String>) {
            result.onSuccess {  it: String
                println(it)
            }
        }
    })
}
```

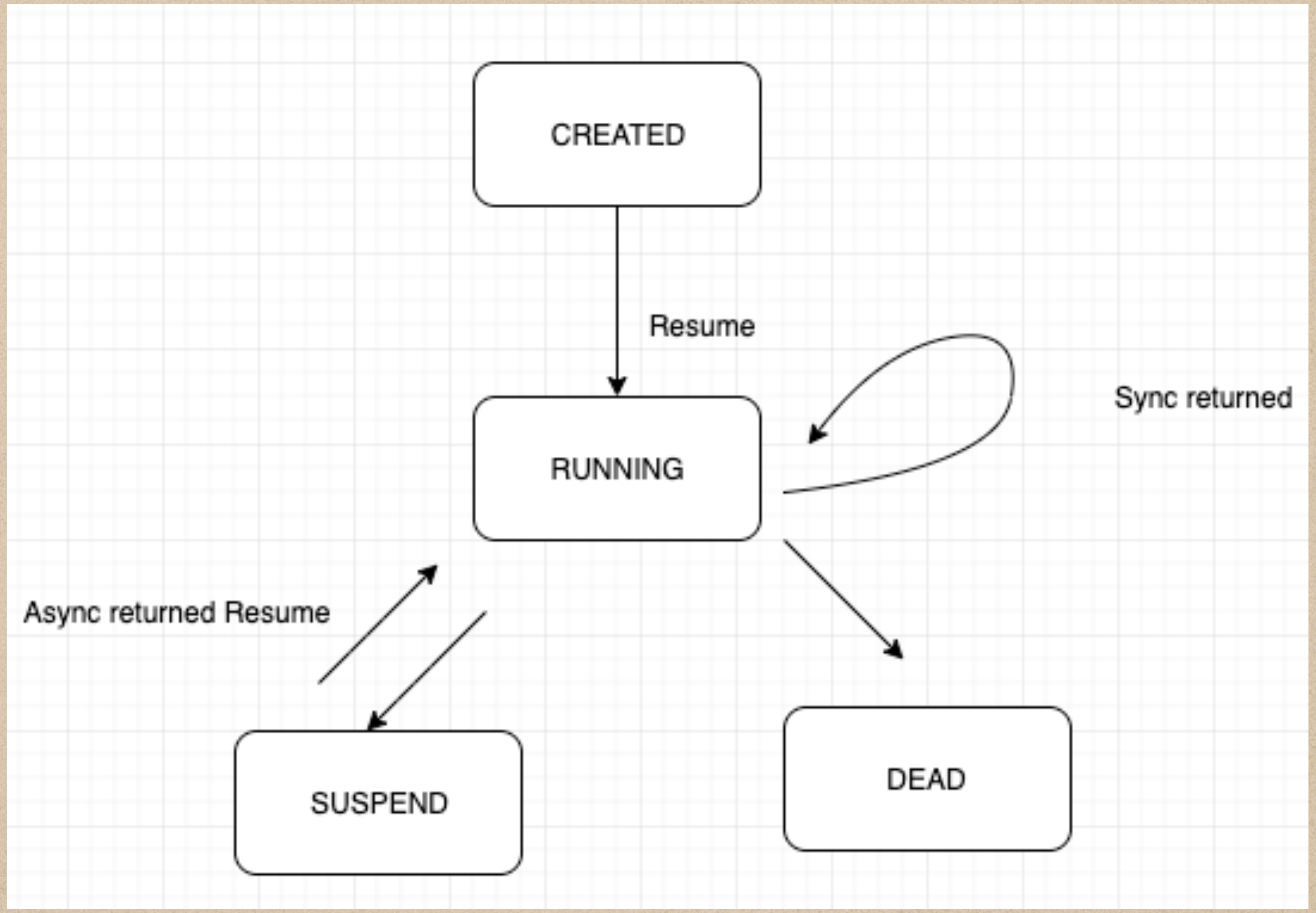
## 使用

- 實作ContinuationInterceptor
- 也是繼承Element
- Key固定為ContinuationInterceptor
- 可以拿來攔截恢復調用
- 恢復調用會是n+1次



```
class LogInterceptor : ContinuationInterceptor{
    override val key = ContinuationInterceptor

    override fun <T> interceptContinuation(continuation: Continuation<T>) = LogContinuation(continuation)

}

class LogContinuation<T>(private val continuation: Continuation<T>):Continuation<T> by continuation{
    override fun resumeWith(result: Result<T>) {
        println("before resumeWith: $result")
        continuation.resumeWith(result)
        println("after resumeWith")
    }
}
```

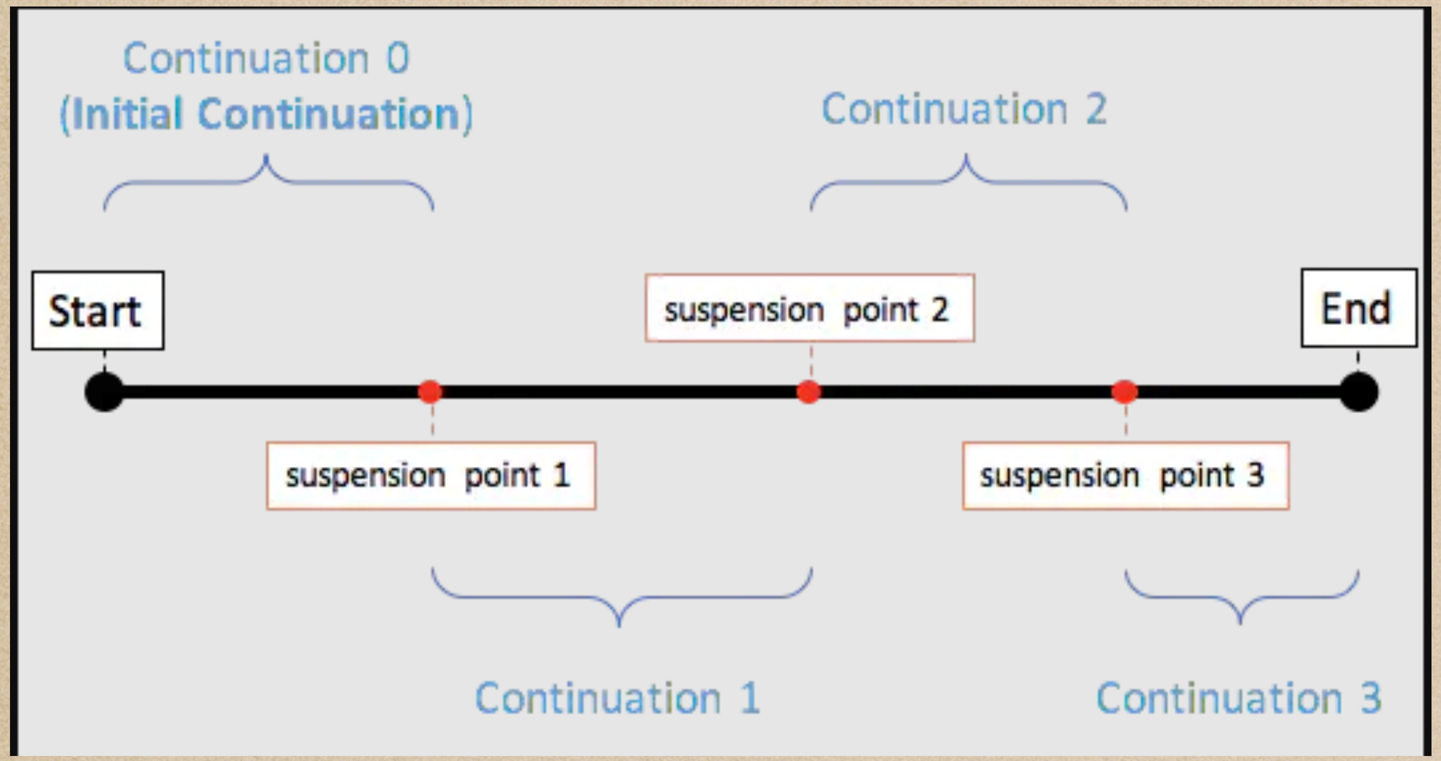
- delegate是攔截器攔截後的Continuation
- 也因此可以進行Thread的切換

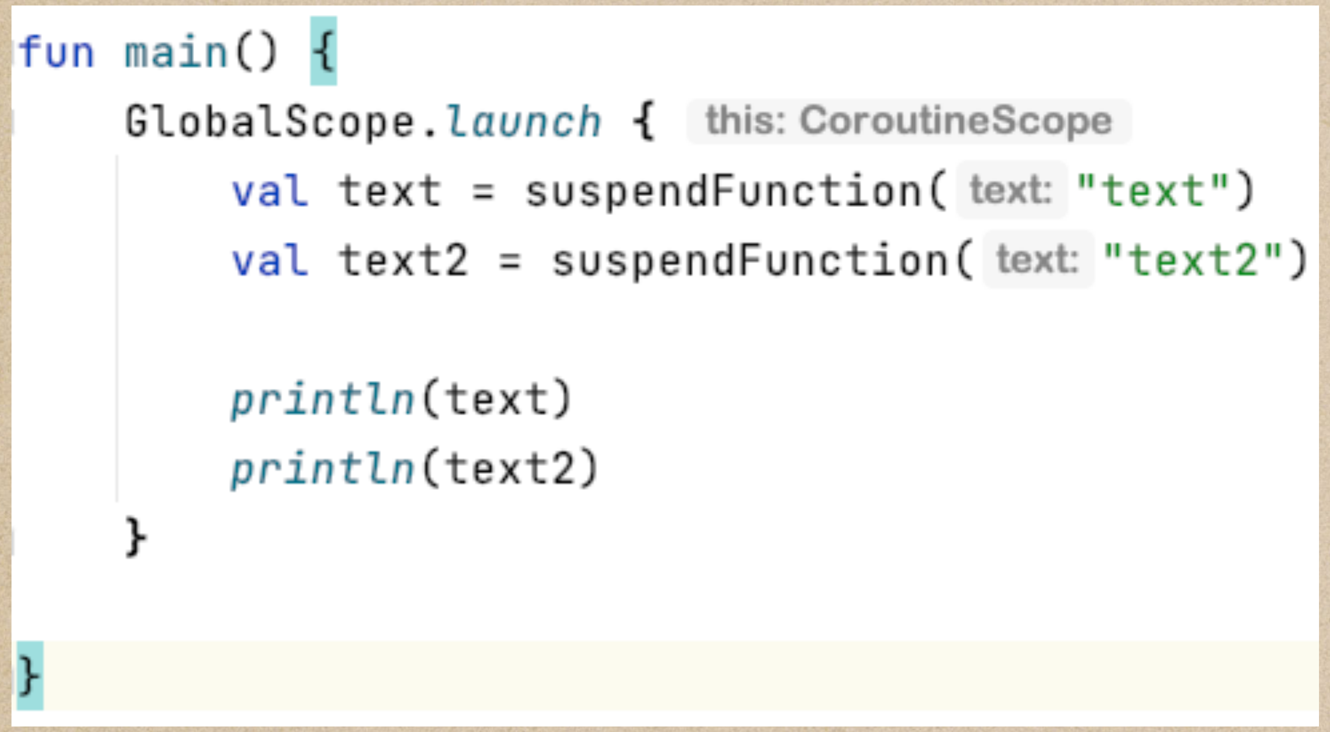


## Kotlin Coroutine的類別

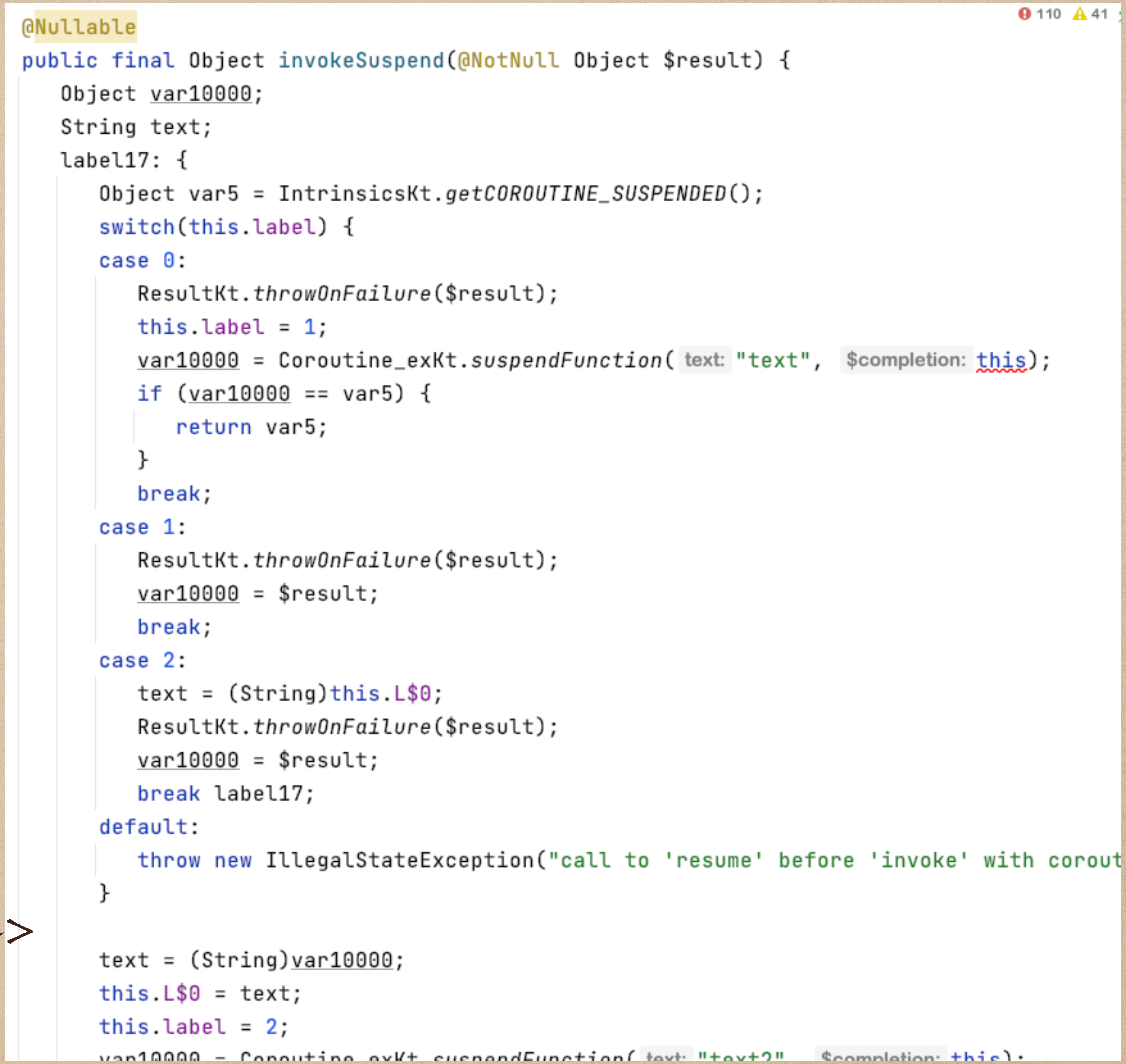
- 分類不是絕對的
- 由於不能在一般function掛起=>無棧
  但是suspend可以任意牽套=>可以當成有棧的實現
- Kotlin是非對稱調用，但是也有自己的對稱Coroutine實現(4.3.2節)

今天到此結束





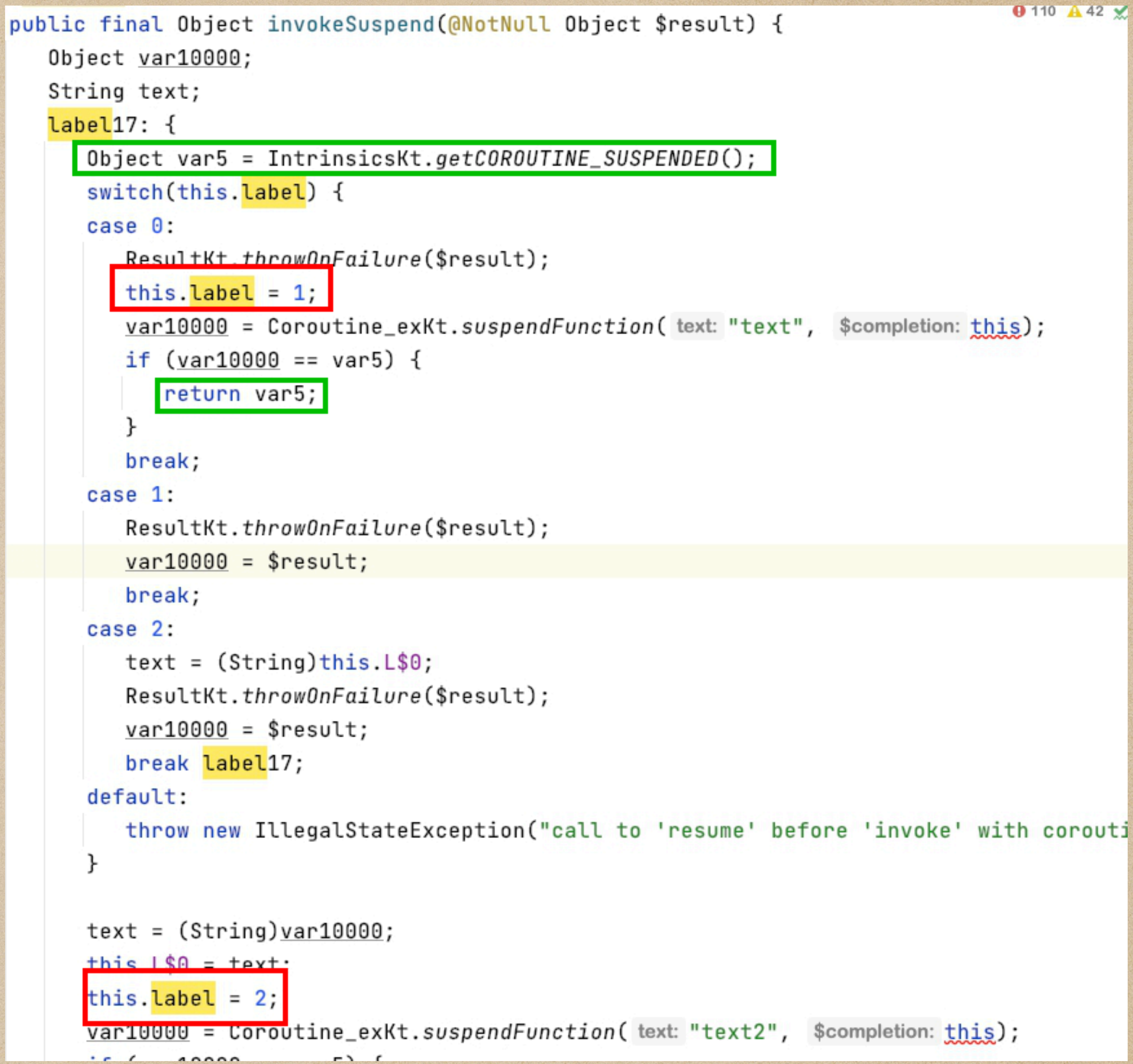
```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        val text = suspendFunction( text: "text")
        val text2 = suspendFunction( text: "text2")

        println(text)
        println(text2)
    }

}
```



```
@Nullable
public final Object invokeSuspend(@NotNull Object $result) {
   Object var10000;
   String text;
   label17: {
      Object var5 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
      switch(this.label) {
      case 0:
         ResultKt.throwOnFailure($result);
         this.label = 1;
         var10000 = Coroutine_exKt.suspendFunction( text: "text",  $completion: this);
         if (var10000 == var5) {
            return var5;
         }
         break;
      case 1:
         ResultKt.throwOnFailure($result);
         var10000 = $result;
         break;
      case 2:
         text = (String)this.L$0;
         ResultKt.throwOnFailure($result);
         var10000 = $result;
         break label17;
      default:
         throw new IllegalStateException("call to 'resume' before 'invoke' with corout
      }

      text = (String)var10000;
      this.L$0 = text;
      this.label = 2;
      var10000 = Coroutine_exKt.suspendFunction( text: "text2",  $completion: this);
```

- SuspendLambda -> ContinuationImpl -> BaseContinuationImpl -> Continuation

- label就是狀態機的狀態
- 如果工作未完成就會return COROUTINE_SUSPEND



```
public final Object invokeSuspend(@NotNull Object $result) {
   Object var10000;
   String text;
   label17: {
      Object var5 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
      switch(this.label) {
      case 0:
         ResultKt.throwOnFailure($result);
         this.label = 1;
         var10000 = Coroutine_exKt.suspendFunction( text: "text",  $completion: this);
         if (var10000 == var5) {
            return var5;
         }
         break;
      case 1:
         ResultKt.throwOnFailure($result);
         var10000 = $result;
         break;
      case 2:
         text = (String)this.L$0;
         ResultKt.throwOnFailure($result);
         var10000 = $result;
         break label17;
      default:
         throw new IllegalStateException("call to 'resume' before 'invoke' with corouti
      }

      text = (String)var10000;
      this.L$0 = text;
      this.label = 2;
      var10000 = Coroutine_exKt.suspendFunction( text: "text2",  $completion: this);
```