

# Spring Coroutine in action

2024.05.25 | **무비한행제를** | 최진영

# 발표자 소개



우아한형제들 광고프로덕트팀  
2년차 백엔드 엔지니어

코틀린으로 커리어 시작

지식 공유를 좋아해서 다양한 활동과 게으른 개발자 컨퍼런스 운영진도 겸하는 중

# 시작하기 전에

## 발표 대상

- spring과 kotlin을 사용하고 있지만 아직 coroutine을 적용하지 않은 분
- coroutine을 적용하였지만 잘 사용하고 있는지 고민중인 분
- 아직 kotlin을 사용하지 않지만 나중을 위해 미리 듣는 분

## 선요약

- coroutine이 등장하게된 배경과 사용방법
- spring mvc에서의 적용과정

# 목차

## 1. Coroutine

- Structured Concurrency
  - CoroutineScope, CoroutineContext, CoroutineBuilder
- CPS (Continuation Pasing Style)
  - suspend

## 2. Spring Coroutine

**coroutine is for async ?**

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay( timeMillis: 1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
```

```
fun main() = {
    launch { // launch a new coroutine and continue
        ...
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
```

Unresolved reference: launch

⋮ or 1 second (default time unit is ms)

Create function 'launch' ↕ More actions... ↕

# CoroutineScope

# Structured Concurrency

- Coroutines follow a principle of structured concurrency which means that new coroutines can only be launched in a specific CoroutineScope which delimits the lifetime of the coroutine. The above example shows that `runBlocking` establishes the corresponding `CoroutineScope` which is why the previous example waits until `World!` is printed after a second's delay.
- In a real application, you will be launching many coroutines simultaneously. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot be closed until all of its children coroutines complete. Structured concurrency also ensures that any errors in a coroutine are properly reported and are never lost.



goto

```
(0) INPUT INVENTORY FILE-A PRICE FILE-B;  
    OUTPUT PRICED-INV FILE-C UNPRICED-INV FILE-D;  
    HSP D.  
(1) COMPARE PRODUCT-NO(A) WITH PRODUCT-NO(B);  
    IF GREATER GO TO OPERATION 10;  
    IF EQUAL GO TO OPERATION 5;  
    OTHERWISE GO TO OPERATION 2.  
(2) TRANSFER A TO D.  
(3) WRITE-ITEM D.  
(4) JUMP TO OPERATION 8.  
(5) TRANSFER A TO C.  
(6) MOVE UNIT-PRICE(B) TO UNIT-PRICE(C).  
(7) WRITE-ITEM C.  
(8) READ-ITEM A; IF END OF DATA GO TO OPERATION 14.  
(9) JUMP TO OPERATION 1.  
(10) READ-ITEM B; IF END OF DATA GO TO OPERATION 12.  
(11) JUMP TO OPERATION 1.  
(12) SET OPERATION 9 TO GO TO OPERATION 2.  
(13) JUMP TO OPERATION 2.  
(14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZZZ;  
    IF EQUAL GO TO OPERATION 16;  
    OTHERWISE GO TO OPERATION 15.  
(15) REWIND B.  
(16) CLOSE-OUT FILES C, D.  
(17) STOP. (END)
```

(0) INPUT INVENTORY FILE-A PRICE FILE-B;  
OUTPUT PRICED-INV FILE-C UNPRICED-INV FILE-D;  
HSP D.

(1) COMPARE PRODUCT-NO(A) WITH PRODUCT-NO(B);  
IF GREATER GO TO OPERATION 10;  
IF EQUAL GO TO OPERATION 5;  
OTHERWISE GO TO OPERATION 2.

(2) TRANSFER A TO D.

(3) WRITE-ITEM D

(4) JUMP TO OPERATION 8.

(5) TRANSFER A TO C.

(6) MOVE UNIT-PRICE(B) TO UNIT-PRICE(C).

(7) WRITE-ITEM C.

(8) READ-ITEM A; IF END OF DATA GO TO OPERATION 14.

(9) JUMP TO OPERATION 1.

(10) READ-ITEM B; IF END OF DATA GO TO OPERATION 12.

(11) JUMP TO OPERATION 1.

(12) SET OPERATION 3 TO GO TO OPERATION 2

(13) JUMP TO OPERATION 2.

(14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZ;  
IF EQUAL GO TO OPERATION 16;  
OTHERWISE GO TO OPERATION 15.

(15) REWIND B.

(16) CLOSE-OUT FILES C, D.

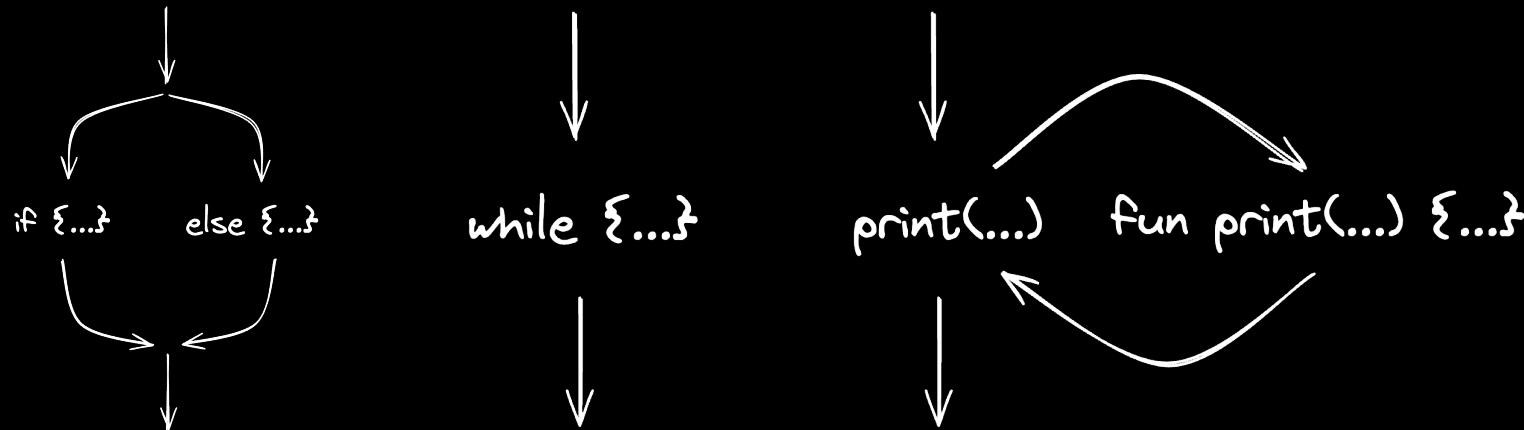
(17) STOP. (END)

# **goto statement considered harmful**

- goto와 같은 unstructured programming을 지양해야한다. Unstructured programming은 프로그램이 일련의 제어 흐름이나 논리적 단계를 따르지 않고 임의의 방향으로 점프하는 것이며 이는 코드의 가독성과 유지 보수성을 떨어뜨린다.
- 따라서 구조적이고 순차적인 프로그래밍 방식으로 프로그램을 작성하는 것이 좋으며 이를 통해 프로그램의 크기가 커져도 코드를 이해하기 쉽게 만들어주며, 개발자가 실수를 범할 가능성을 줄여준다.
- structured programming을 통해 흐름을 제어하게 되면 결국 하나의 흐름으로 돌아오기 때문에 각 호출 혹은 분기들이 black box 패턴과 연관지어진다.

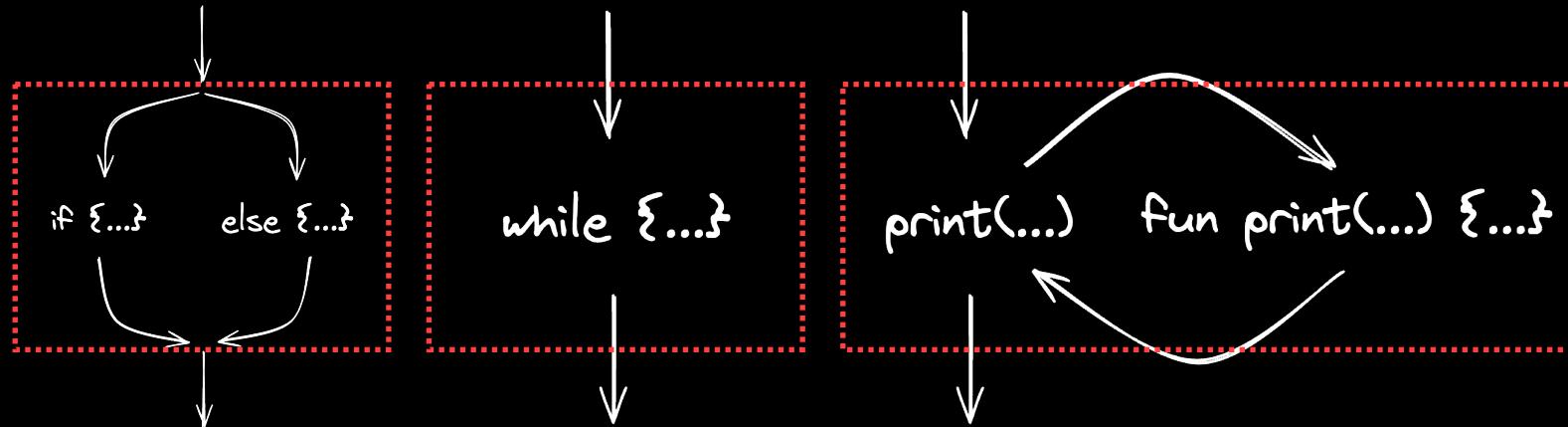
# Structured Programming

하나의 흐름이 다른 함수로 가더라도 원래의 흐름으로 돌아오는 형태



# Structured Programming

하나의 흐름이 다른 함수로 가더라도 원래의 흐름으로 돌아오는 형태



# Async is structured programming?

```
fun main() {  
    println("start")  
    Thread {  
        println("new thread start")  
    }.start()  
    println("end")  
}
```

println("start")



Thread { ... }.start()



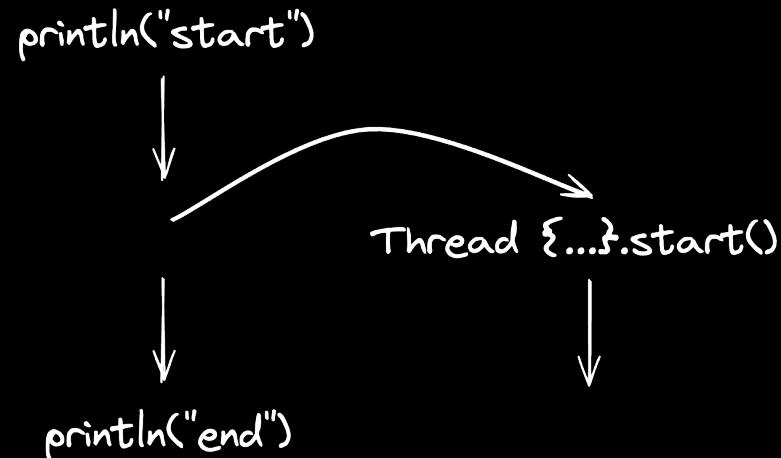
println("end")

# Async is not structured programming

새 스레드의 실행이 새로운 실행 흐름으로 넘어가므로 흐름의 제어가 되고 있지 않음

자식 스레드에서 발생한 예외는 부모 스레드의 흐름에서 확인할 수 없음

```
fun main() {  
    println("start")  
    Thread {  
        println("new thread start")  
    }.start()  
    println("end")  
}
```

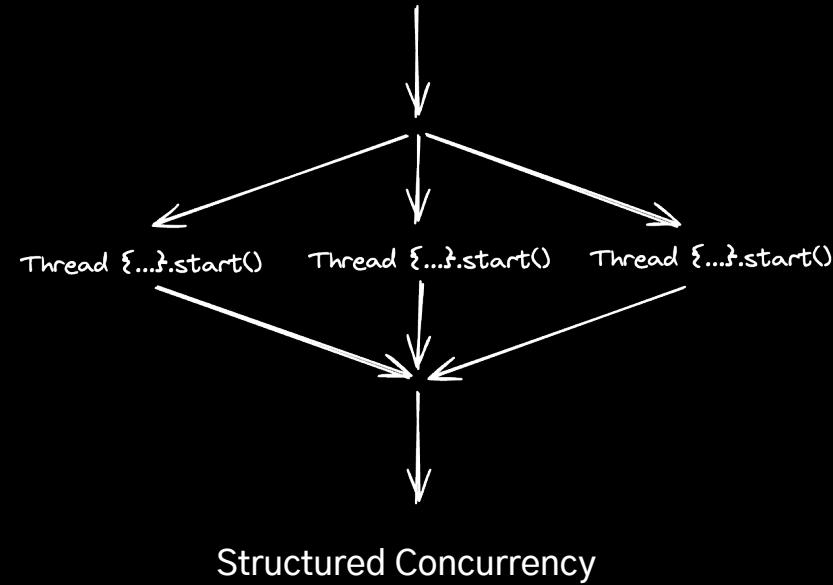
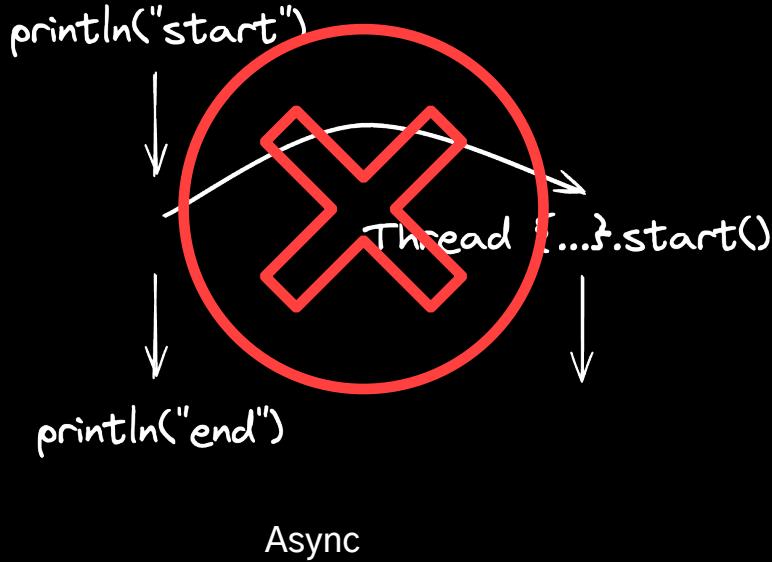


**비동기를 안쓸순 없다**

# Structured Concurrency

# Structured Concurrency

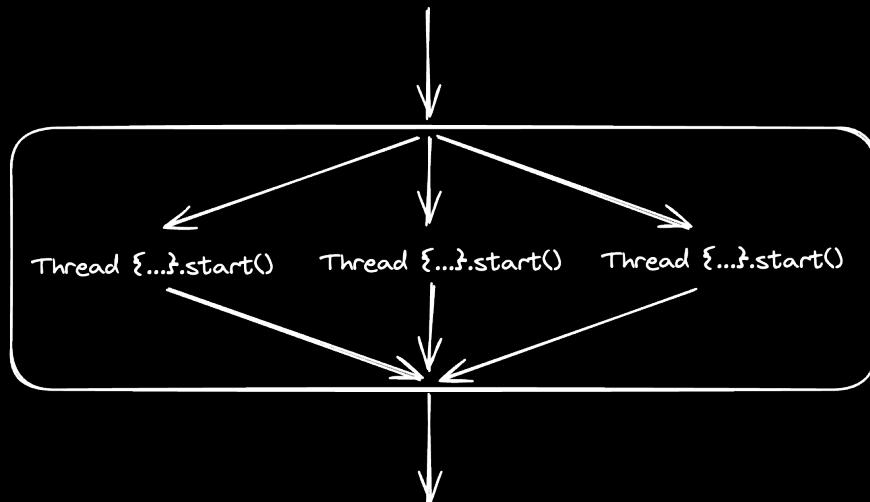
각 스레드의 실행 후 실행한 부모 스레드의 흐름으로 돌아오게끔 보장



# Structured Concurrency

하나의 block으로 감싸져있는 형태로 구성

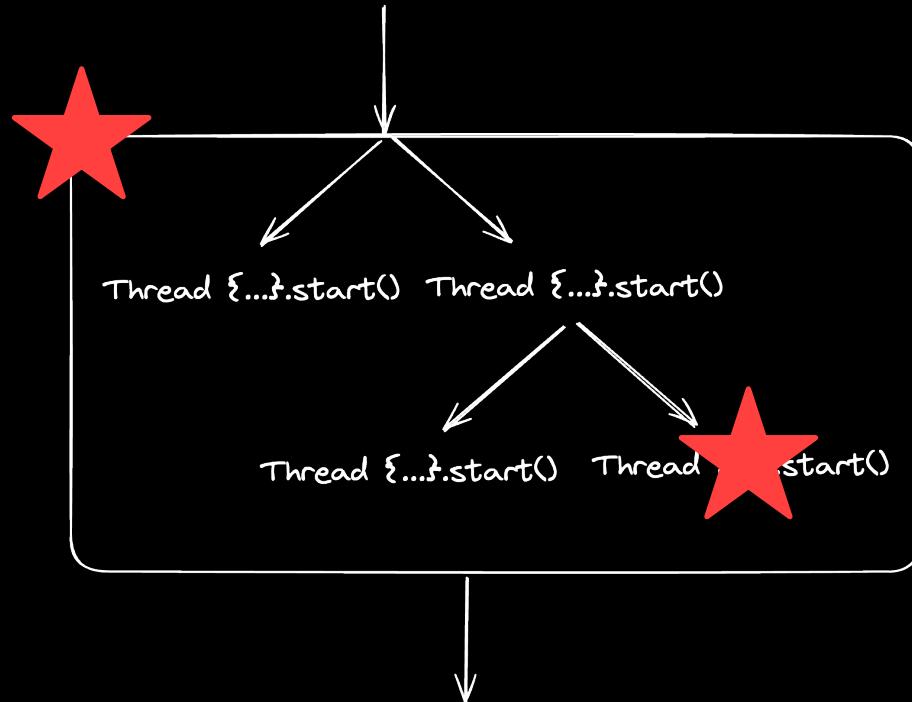
block에서 발생한 예외는 부모 스레드가 관리하는 흐름 내에 있어 컨트롤 가능



```
fun main() = runBlocking { // this: CoroutineScope
    launch {
        delay( timeMillis: 1000L)
        println("World!")
    }
    println("Hello")
}
```

# Structured Concurrency

하나의 Scope에서 자식 코루틴의 예외가 발생하더라도 부모 코루틴으로 예외가 전파됨



# Structured Concurrency

- Coroutine은 Structured Concurrency를 따르기 때문에 새로운 Coroutine의 실행은 특정 CoroutineScope에서만 실행할 수 있다.
- 실제 어플리케이션에서는 많은 Coroutine을 실행하지만 Structured Concurrency로 인해 Coroutine이 손실되거나 누수되지 않도록 보장한다.
- 따라서 발생되는 오류도 누수되지 않고 제대로 보고되고 손실되지 않도록 보장한다.

# CoroutineScope

새로운 코루틴을 실행할 수 있는 범위

```
public interface CoroutineScope {
```

The context of this scope. Context is encapsulated by the scope and used for implementation of coroutine builders that are extensions on the scope. Accessing this property in general code is not recommended for any purposes except accessing the `Job` instance for advanced usages.

By convention, should contain an instance of a `Job` to enforce structured concurrency.

```
    public val coroutineContext: CoroutineContext
```

```
}
```

# CoroutineContext

- CoroutineScope가 실행될 때 그 환경에 대한 정보를 가지고 있음
- Coroutine을 어떻게 처리할 것인지에 대한 element(정보)를 포함
  - 여러 개의 element를 Map 자료구조와 동일한 key를 기반으로 값을 저장
- CoroutineContext에 저장되는 element는 key가 중복되지 않아 값을 더하면 이후의 값으로 덮어씌워짐

# CoroutineContext

```
fun main() {  
    val coroutineContext = EmptyCoroutineContext  
    println(coroutineContext[CoroutineName]) // null  
  
    val newCoroutineContext = coroutineContext + CoroutineName(name: "new CoroutineContext")  
    println(newCoroutineContext[CoroutineName]) // CoroutineName(new CoroutineContext)  
}
```

# CoroutineContext

1. CoroutineName : 코루틴의 이름
2. CoroutineDispatcher : 코루틴이 실행할 때 할당될 스레드  
(Dispatchers.IO, Dispatchers.Default … )
3. Job : 코루틴을 컨트롤하기 위한 것으로 상태를 가지며 상태를 변화함
4. CoroutineExceptionHandler : 코루틴 예외 발생 시 예외처리

# CoroutineBuilder – launch, async

launch : 현재 스레드를 차단하지 않고 새 코루틴을 비동기로 실행

async : launch와 동일하게 새 코루틴을 비동기로 실행하며 실행된 결과(Deferred)를 전달 받음

```
fun main() = runBlocking {
    launch { println("launch 1") }
    launch { println("launch 2") }
    delay( timeMillis: 1000 )
}
```

```
fun main() = runBlocking {
    val a = async { findA() }.await()
    val b = async { findB() }.await()
    val c = async { findC() }

    println(a + b + c.await())
}
```

# CoroutineBuilder – runBlocking

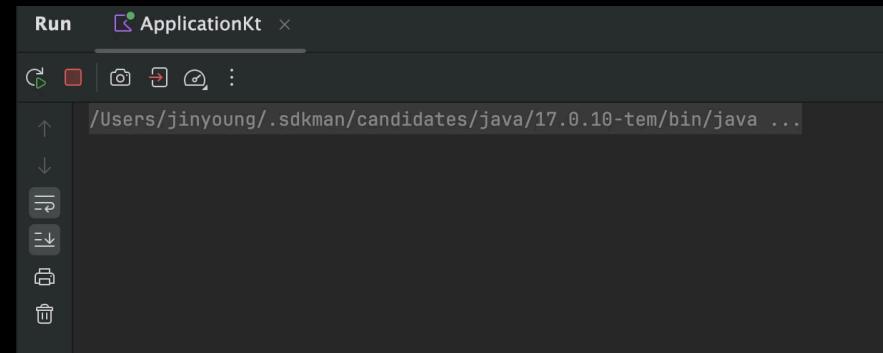
새 코루틴을 실행하고 완료될 때까지 현재 스레드를 차단하는 Scope

```
@Throws(InterruptedException::class)
public actual fun <T> runBlocking(context: CoroutineContext, block: suspend CoroutineScope.() -> T): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    val currentThread = Thread.currentThread()
    val contextInterceptor = context[ContinuationInterceptor]
    val eventLoop: EventLoop?
    val newContext: CoroutineContext
    if (contextInterceptor == null) {
        // create or use private event loop if no dispatcher is specified
    }
}
```

# CoroutineBuilder – runBlocking

runBlocking 내부의 동작이 비동기로 모두 이루어져도 main은 runBlocking의 모든 작업이 종료될 때까지 대기

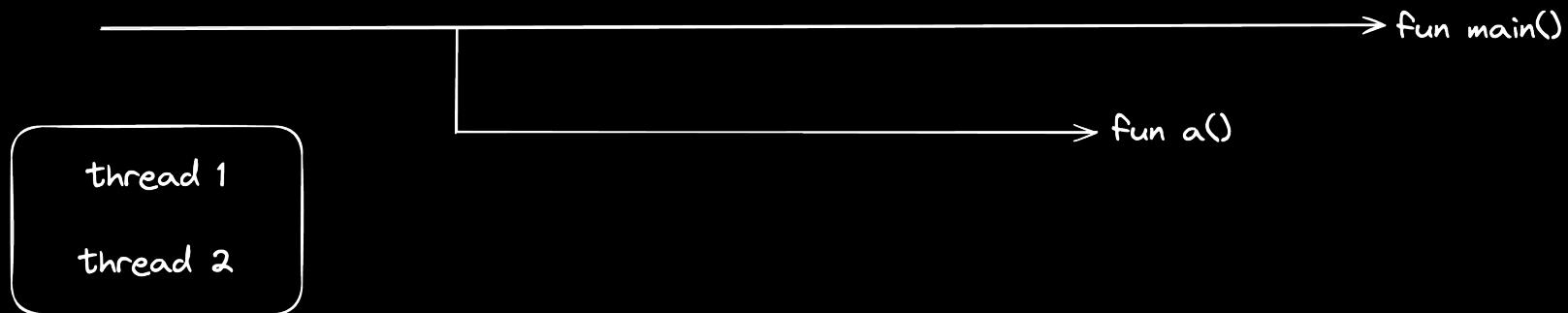
```
fun main() {  
    println("start") // main  
    runBlocking(Dispatchers.IO) {  
        launch { println("launch 1") } // DefaultDispatcher-worker-1  
        launch { println("launch 2") } // DefaultDispatcher-worker-2  
        delay( timeMillis: 1000)  
    }  
    println("end") // main  
}
```



# CoroutineScope

```
fun main() {  
    runBlocking { Parent Scope  
        launch { Child Scope  
            delay( timeMillis: 1000L)  
            println("World!")  
        }  
        println("Hello,")  
    }  
}
```

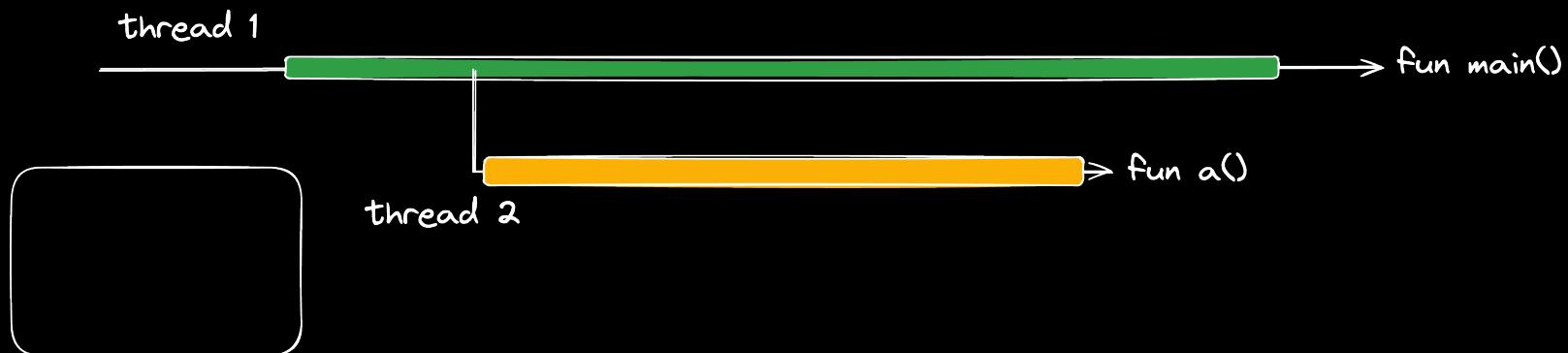
# Multithread programming



# Multithread programming



# Multithread programming



# Multithread programming

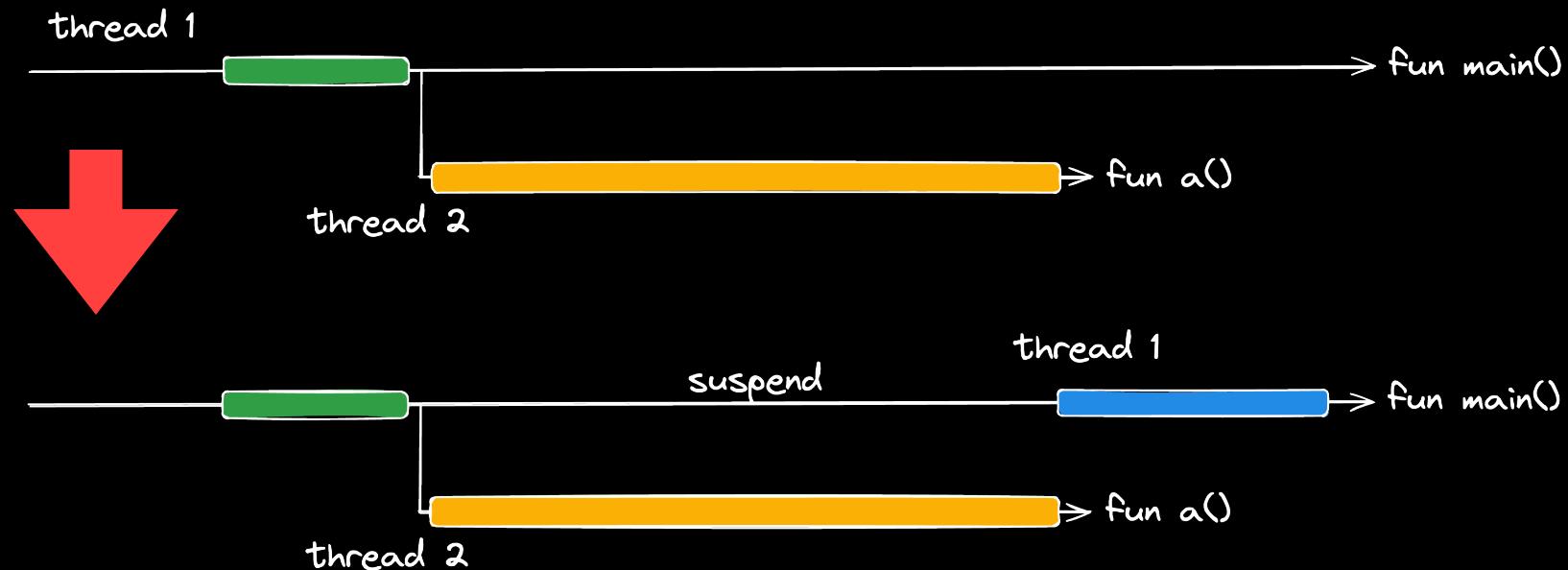
thread1은 함수 a가 종료될 때까지 실행의 종료를 기다려야함



# Coroutine

thread 2의 실행 시간 동안 기다리는 것이 아닌

일시 정지 후 필요한 지점에서 스레드를 할당하여 재실행



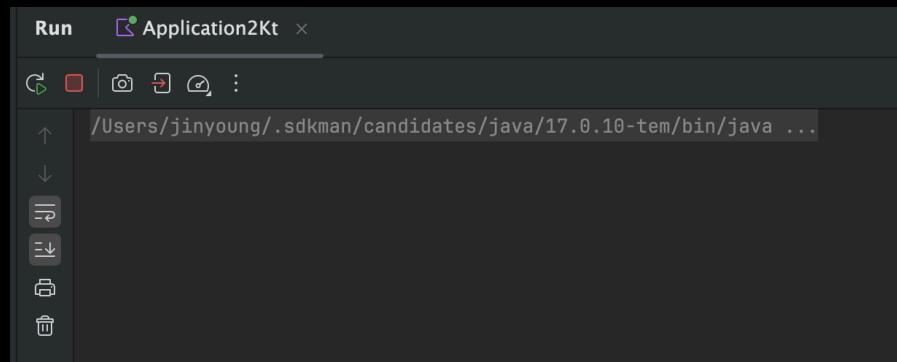
# suspend

유예하다, 중지하다

# Suspend 예시 - delay

Thread.sleep처럼 thread 2의 실행 시간 동안 기다리는 것이 아닌  
일시 정지 후 필요한 지점에서 스레드를 할당하여 재실행

```
▶ fun main() = runBlocking {
    println("start")
    launch {
        delay( timeMillis: 1000)
        println("launch end")
    }
    println("end")
}
```



**어떻게 일시 정지 후 재실행하는 포인트를 찾을 수 있는가**

# Continuation

일시 중단된 이후 연속성 정보를 표현하기 위한 인터페이스

실행 환경(CoroutineContext)와 재실행 시 호출될 메소드(resumeWith)

```
Interface representing a continuation after a suspension point that returns a value of type T.

@SinceKotlin(version: "1.3")
public interface Continuation<in T> {
    /**
     * The context of the coroutine that corresponds to this continuation.
     */

    public val context: CoroutineContext

    /**
     * Resumes the execution of the corresponding coroutine passing a successful or failed result as
     * the return value of the last suspension point.
     */

    public fun resumeWith(result: Result<T>)
}
```

# Continuation

function에 suspend keyword를 달면 kotlin컴파일 시 Continuation을 파라미터로 받을 수 있게 변환

```
suspend fun hello() {  
    println("hello")  
}
```



```
@Nullable  
public static final Object hello(@NotNull Continuation<? super Unit> $completion) {  
    System.out.println((Object)"hello");  
    return Unit.INSTANCE;  
}
```

# CPS(Continuation Passing Style)

# Continuation Passing Style

- 단순히 suspend keyword를 사용한다고 모든 function이 일시중단되는 것은 아니다.
- 일시 중단 상태를 표현하는 COROUTINE\_SUSPENDED를 반환할 수 있어야 일시 중단이 가능하다. 이때 일시 중단이 가능한 지점을 Suspension Point라 하고, 코틀린에서는 State Machine으로 이 지점들을 관리한다.

# Continuation Passing Style

일시 중단 시점인 suspension point를 통해 컴파일

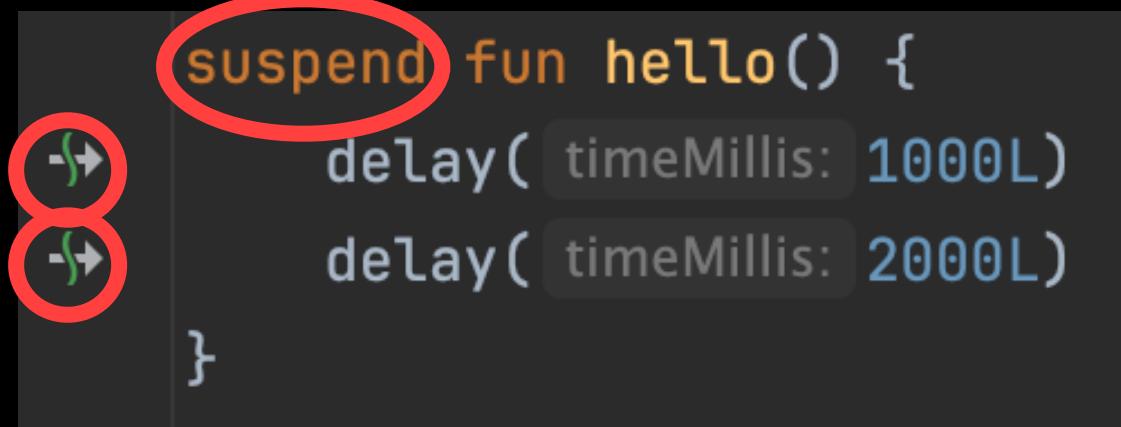


A screenshot of a code editor showing a Kotlin file. The code contains a main function that runs a blocking delay:

```
fun main() = runBlocking {
    delay(timeMillis: 1000)
}
```

The editor interface includes a play button icon and a red circle highlighting a green CPS suspension point icon (a green arrow pointing right with a green brace underneath). This icon indicates where the compiler will insert continuation passing code.

# Continuation Passing Style – compile



A screenshot of a code editor displaying a Kotlin file. The code contains a `suspend` function named `hello`. The word `suspend` is highlighted with a red oval. Two green continuation markers, represented by a green arrow pointing right with a green brace underneath, are circled in red. The function body contains two `delay` statements with time intervals of 1000L and 2000L respectively. The code ends with a closing brace.

```
suspend fun hello() {
    delay( timeMillis: 1000L)
    delay( timeMillis: 2000L)
}
```

# Continuation Passing Style – compile

1. suspend keyword를 붙이면 파라미터에 Continuation을 넣어준다.

```
suspend fun hello() {  
    delay( timeMillis: 1000L)  
    delay( timeMillis: 2000L)  
}
```

kotlin

```
public Object hello(Continuation continuation) {
```

to java

# Continuation Passing Style – compile

## 2. Continuation 초기화

```
suspend fun hello() {  
    delay( timeMillis: 1000L)  
    delay( timeMillis: 2000L)  
}
```

kotlin

```
public Object hello(Continuation continuation) { 1 usage  
    Continuation currentContinuation;  
    if (continuation.equals("이미 Continuation을 초기화한 경우")) {  
        currentContinuation = continuation;  
    } else {  
        currentContinuation = new ContinuationImpl(continuation) {  
            Object result; 1 usage  
            int label; no usages  
  
            protected Object invokeSuspend(@NotNull Object result) {  
                this.result = result;  
                return hello(continuation: this);  
            }  
        };  
    }  
}
```

to java

# Continuation Passing Style – compile

2. Continuation 초기화 – 첫 호출의 경우 Continuation을 새로 생성

invokeSuspend는 resumeWith에서 사용

```
suspend fun hello() {  
    delay( timeMillis: 1000L)  
    delay( timeMillis: 2000L)  
}
```

kotlin

```
public Object hello(Continuation continuation) { 1 usage  
    Continuation currentContinuation;  
    if (continuation.equals("이미 Continuation을 초기화한 경우")) {  
        currentContinuation = continuation;  
    } else {  
        currentContinuation = new ContinuationImpl(continuation) {  
            Object result; 1 usage  
            int label; no usages  
  
            protected Object invokeSuspend(@NotNull Object result) {  
                this.result = result;  
                return hello(continuation: this);  
            }  
        };  
    }  
}
```

to java

# Continuation Passing Style – compile

3. switch문으로 suspension point마다 case 추가

```
suspend fun hello() {  
    delay( timeMillis: 1000L)  
    delay( timeMillis: 2000L)  
}
```

kotlin

```
switch (currentContinuation.label) {  
    case 0:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 1;  
        Object caseResult0 = DelayKt.delay( value: 1000L, continuation);  
        if (caseResult0 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
    case 1:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 2;  
        Object caseResult1 = DelayKt.delay( value: 2000L, continuation);  
        if (caseResult1 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
        return Unit.INSTANCE;  
    case 2:  
        throwOnFailure(currentContinuation.result);  
        return Unit.INSTANCE;  
    }  
    throw new IllegalStateException("더이상 resume될 수 없음");  
}
```

to java

# Continuation Passing Style – compile

4. switch는 suspension point인 label을 통해서 실행된다.

```
switch (currentContinuation.label) {  
    case 0:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 1;  
        Object caseResult0 = DelayKt.delay( value: 1000L, continuation);  
        if (caseResult0 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
}
```

# Continuation Passing Style – compile

5. 실행 전 Coroutine의 실패 여부를 확인한다.

```
switch (currentContinuation.label) {  
    case 0:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 1;  
        Object caseResult0 = DelayKt.delay(value: 1000L, continuation);  
        if (caseResult0 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
}
```

# Continuation Passing Style – compile

6. label값을 다음 실행할 label로 먼저 올린다.

먼저 올리지 않으면 예외 혹은 재실행 시 무한 루프가 발생할 수 있다.

```
switch (currentContinuation.label) {
    case 0:
        throwOnFailure(currentContinuation.result);
        currentContinuation.label = 1;
        Object caseResult0 = DelayKt.delay(value: 1000L, continuation);
        if (caseResult0 == "COROUTINE_SUSPENDED") {
            return "COROUTINE_SUSPENDED";
        }
}
```

# Continuation Passing Style – compile

7. suspend function 실행한다.

suspend function은 실행 후 타입과 관계 없이 항상 suspend에 대한 결과를 반환한다.

```
switch (currentContinuation.label) {  
    case 0:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 1;  
        Object caseResult0 = DelayKt.delay(value: 1000L, continuation);  
        if (caseResult0 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
}
```

# Continuation Passing Style – compile

8. suspend function 실행이 일시중단 시그널을 반환할 경우 이를 그대로 리턴한다.  
따라서 현재 함수(hello)가 일시 정지되었음을 전파한다.

```
switch (currentContinuation.label) {  
    case 0:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 1;  
        Object caseResult0 = DelayKt.delay( value: 1000L, continuation);  
        if (caseResult0 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
}
```

# Continuation Passing Style – compile

9. 만약 COROUTINE\_SUSPENDED가 아니라면 올려놓은 label 값으로 바로 다음 suspension point 인 case 1로 넘어간다.

```
switch (currentContinuation.label) {  
    case 0:  
        throwOnFailure(currentContinuation.result);  
        currentContinuation.label = 1;  
        Object caseResult0 = DelayKt.delay( value: 1000L, continuation);  
        if (caseResult0 == "COROUTINE_SUSPENDED") {  
            return "COROUTINE_SUSPENDED";  
        }  
    case 1:  
        throwOnFailure(currentContinuation.result);
```

# Continuation Passing Style – compile

10. case 1도 동일한 한 suspend function으로 현재 함수(hello)의 호출의 마지막이기 때문에 만약 일시정지되지 않는다면 그대로 Unit.INSTANCE로 void처리를 한다.

```
case 1:  
    throwOnFailure(currentContinuation.result);  
    currentContinuation.label = 2;  
    Object caseResult1 = DelayKt.delay( value: 2000L, continuation);  
    if (caseResult1 == "COROUTINE_SUSPENDED") {  
        return "COROUTINE_SUSPENDED";  
    }  
    return Unit.INSTANCE;
```

# Continuation Passing Style – compile

11. case1에서 일시중단되었다면 다시 resume되었을 때를 위해서 case2에서는 suspension하지 않은 남은 처리들을 진행한다.

```
case 2:  
    throwOnFailure(currentContinuation.result);  
    return Unit.INSTANCE;
```

# Continuation Passing Style – compile

12. switch문에서 label을 통해 모든 함수의 실행흐름을 관리하고 있다.

정상적으로 실행이 되었다면 switch문을 벗어날 수 없으므로 switch문의 아래에 예외를 둔다.

```
switch (currentContinuation.label) {...}  
throw new IllegalStateException("더이상 resume될 수 없음");
```

# Continuation Passing Style

- `suspend` 를 붙이게 되면 kotlin에서 컴파일 과정에서 자동으로 일시 중단이 가능하도록 state machine 구현체로 컴파일한다.
- `suspend` 를 붙인다고 모두 일시중단이 가능해지는 것이 아닌 suspension point로 인한 일시중단된 트리거가 있어야 일시중단이 가능해진다.
- `suspend` 는 그자체로 Coroutine이 아니다. cps를 통해 Coroutine이 일시 중단을 컨트롤할 수 있는 구현체로 컴파일해줄 뿐이다.

# why coroutine

- Structured Concurrency를 얻어 수준에서 제공하기 때문에 Coroutine이 누설되지 않고 안정적인 동시성 프로그램을 작성할 수 있다.
- 개발자는 간단하게 suspend keyword만 사용하여도 컴파일시점에 CPS(Continuation Passing Style)을 제공한다. 이로 인해 기존의 멀티 스레드보다 불필요한 스레드 생성 혹은 blocking이 없어 성능이 더 좋아진다.

# spring coroutine

# spring coroutine

```
fun getArticleById(id: Long): ArticleResponse {  
    val article = articleQueryService.findById(id)  
    val comments = commentQueryService.findByArticleId(id)  
  
    return ArticleResponse.of(article, comments)  
}
```



```
fun getArticleById(id: Long): ArticleResponse = runBlocking {  
    val article = async { articleQueryService.findById(id) }  
    val comments = async { commentQueryService.findByArticleId(id) }  
  
    return@runBlocking ArticleResponse.of(article.await(), comments.await())  
}
```

```
fun getArticleById(id: Long): ArticleResponse = runBlocking {
    log.info("request")
    val article = async {
        log.info("articleQueryService")
        articleQueryService.findById(id)
    }
    val comments = async {
        log.info("commentQueryService")
        commentQueryService.findByArticleId(id)
    }

    return@runBlocking ArticleResponse.of(article.await(), comments.await())
        .also { log.info("response") }
}
```

22:33:19.492	[http-nio-8080-exec-1 @coroutine#1]	INFO - request
22:33:19.496	[http-nio-8080-exec-1 @coroutine#2]	INFO - articleQueryService
22:33:19.512	[http-nio-8080-exec-1 @coroutine#3]	INFO - commentQueryService
22:33:19.580	[http-nio-8080-exec-1 @coroutine#1]	INFO - response

```
fun getArticleById(id: Long): ArticleResponse = runBlocking {
    log.info("request")
    val article = async {
        log.info("articleQueryService")
        articleQueryService.findById(id)
    }
    val comments = async {
        log.info("commentQueryService")
        commentQueryService.findByArticleId(id)
    }

    return@runBlocking ArticleResponse.of(article.await(), comments.await())
        .also { log.info("response") }
}
```

suspend fun

suspend fun

```
22:33:19.492 [http-nio-8080-exec-1 @coroutine#1]  INFO - request
22:33:19.496 [http-nio-8080-exec-1 @coroutine#2]  INFO - articleQueryService
22:33:19.512 [http-nio-8080-exec-1 @coroutine#3]  INFO - commentQueryService
22:33:19.580 [http-nio-8080-exec-1 @coroutine#1]  INFO - response
```

# spring coroutine

```
fun getArticleById(id: Long): ArticleResponse = runBlocking(Dispatchers.IO) {  
    log.info("request")  
    val article = async {  
        log.info("articleQueryService")  
        articleQueryService.findById(id)  
    }  
    val comments = async {  
        log.info("commentQueryService")  
        commentQueryService.findByArticleId(id)  
    }  
  
    return@runBlocking ArticleResponse.of(article.await(), comments.await())  
        .also { log.info("response") }  
}
```

# spring coroutine

```
fun getArticleById(id: Long): ArticleResponse = runBlocking {
    log.info("request")
    val article = async(Dispatchers.IO) {
        log.info("articleQueryService")
        articleQueryService.findById(id)
    }
    val comments = async(Dispatchers.IO) {
        log.info("findByArticleId")
        commentQueryService.findByArticleId(id)
    }

    return@runBlocking ArticleResponse.of(article.await(), comments.await())
        .also { log.info("response") }
}
```

# spring coroutine

withContext는 상위 CoroutineContext에서 필요한 Context 정보를 덮어씌어 사용할 수 있음  
하지만 withContext를 호출한 상위 Scope를 일시정지 시킴

```
@Transactional(readOnly = true)
suspend fun findById(id: Long): Article = withContext(Dispatchers.IO) {
    articleRepository.findById(id).orElseThrow()
}
```

# spring coroutine

article async CoroutineScope은 withContext로 인해 일시 정지

```
fun getArticleById(id: Long): ArticleResponse = runBlocking {
    log.info("request")
    val article = async {
        log.info("articleQueryService")
        articleQueryService.findById(id) // suspend scope
        // 현재 scopeOf findById가 완료될 때까지 일시 정지된다.
    }
    val comments = async {
        log.info("commentQueryService")
        commentQueryService.findByArticleId(id)
    }
}
```

# spring coroutine

```
fun getArticleById(id: Long): ArticleResponse = runBlocking(dispatchers.I0) {
    log.info("request")
    val article = async {
        log.info("articleQueryService")
        articleQueryService.findById(id)
    }
    val comments = async {
        log.info("findByArticleId")
        commentQueryService.findByArticleId(id)
    }

    return@runBlocking ArticleResponse.of(article.await(), comments.await())
        .also { log.info("response") }
}
```



coroutineDispatchers

# spring coroutine – MDCContext

CoroutineScope마다 실행되는 worker 스레드가 달라 MDC 정보가 공유되지 않을 수 있음

```
fun getArticleById(id: Long): ArticleResponse = runBlocking(Dispatchers.IO) {  
    MDC.put(key: "key", val: "mdc")  
  
    val article = async {  
        log.info(MDC.get("key"))  
        articleQueryService.findById(id)  
    }  
    val comments = async {  
        log.info(MDC.get("key"))  
        commentQueryService.findByArticleId(id)  
    }  
    return@runBlocking ArticleResponse.of(article.await(), comments.await())  
        .also { log.info(MDC.get("key")) } null  
}
```

# spring coroutine – MDCCContext

## MDCCContext

```
class MDCCContext(  
    val contextMap: MDCCContextMap = MDC.getCopyOfContextMap()  
) : AbstractCoroutineContextElement, ThreadContextElement<MDCCContextMap>
```

(source)

MDC context element for CoroutineContext.

Example:

```
MDC.put("kotlin", "rocks") // Put a value into the MDC context  
  
launch(MDCCContext()) {  
    logger.info { "..." } // The MDC context contains the mapping here  
}
```

Note that you cannot update MDC context from inside the coroutine simply using `MDC.put`. These updates are going to be lost on the next suspension and reinstalled to the MDC context that was captured or explicitly specified in `contextMap` when this object was created on the next resumption.

# spring coroutine – MDCCContext

MDCCContext 생성 시점에 MDC 정보들을 복사해서 저장

```
public class MDCCContext(
    |   The value of MDC context map.
    |
    @Suppress( ...names: "MemberVisibilityCanBePrivate")
    public val contextMap: MDCCContextMap = MDC.getCopyOfContextMap()
) : ThreadContextElement<MDCCContextMap>, AbstractCoroutineContextElement(Key) {
```

# spring coroutine – MDCCContext

MDCCContext 생성 시점에 MDC 정보들을 복사해서 저장

```
fun getArticleById(id: Long): ArticleResponse = runBlocking( context: Dispatchers.IO + MDCCContext() ) {
    MDC.put( key: "key",  val: "mdc" )

    val article = async {
        log.info(MDC.get("key"))
        articlequeryservice.findById(id)
    }
    val comments = async {
        log.info(MDC.get("key"))
        commentqueryService.findByIdArticleId(id)
    }
    return@runBlocking ArticleResponse.of(article.await(), comments.await())
        .also { log.info(MDC.get("key")) }
}
```

# spring coroutine – MDCContext

1. MDC 저장 시점을 MDCContext 생성 전에 두는 것을 보장

```
fun getArticleById(id: Long): ArticleResponse {  
    MDC.put(key: "key", val: "mdc")  
    return runBlocking(context: Dispatchers.IO + MDCContext()) {
```

# spring coroutine – MDCContext

2. 각 Coroutine을 실행할 때마다 필요한 위치에 MDCContext를 생성해 저장해서 쓸 수 있도록 보장

```
val article = async(MDCContext()) {
    log.info(MDC.get("key"))
    articleQueryService.findById(id)
}

val comments = async(MDCContext()) {
    log.info(MDC.get("key"))
    commentQueryService.findByArticleId(id)
}

return@runBlocking withContext(MDCContext()) {
    ArticleResponse.of(article.await(), comments.await())
        .also { log.info(MDC.get("key")) }
}
```

# spring coroutine – runBlocking

Calling `runBlocking` from a suspend function is redundant.

## runBlocking

concurrent

```
expect fun <T> runBlocking(  
    context: CoroutineContext = EmptyCoroutineContext,  
    block: suspend CoroutineScope.() -> T  
) : T
```

(source)

Runs a new coroutine and **blocks** the current thread until its completion.

It is designed to bridge regular blocking code to libraries that are written in suspending style, to be used in `main` functions and in tests.

Calling `runBlocking` from a suspend function is redundant. For example, the following code is incorrect:

```
suspend fun loadConfiguration() {  
    // DO NOT DO THIS:  
    val data = runBlocking { // ← redundant and blocks the thread, do not do that  
        fetchConfigurationData() // suspending function  
    }  
}
```

Here, instead of releasing the thread on which `loadConfiguration` runs if `fetchConfigurationData` suspends, it will block, potentially leading to thread starvation issues.

# spring coroutine – runBlocking

```
fun main() {  
    val dispatchers = Executors.newFixedThreadPool( nThreads: 1)  
        .asCoroutineDispatcher()  
    runBlocking(dispatchers) {  
        runBlocking(dispatchers) {  
            println(1)  
        }  
    }  
    dispatchers.close()  
}
```

dead lock

```
fun main() {  
    val dispatchers = Executors.newFixedThreadPool( nThreads: 1)  
        .asCoroutineDispatcher()  
    runBlocking(dispatchers) {  
        runBlocking {  
            println(1)  
        }  
    }  
    dispatchers.close()  
}
```

success

# spring coroutine – runBlocking

runBlocking에서 CoroutineContext를 추가하지 않는 경우 내부적으로 eventLoop가 실행

```
if (contextInterceptor == null) {
    // create or use private event loop if no dispatcher is specified
    eventLoop = ThreadLocalEventLoop.eventLoop
    newContext = GlobalScope.newCoroutineContext(context: context + eventLoop)
} else {
    // See if context's interceptor is an event loop that we shall use (to support TestContext)
    // or take an existing thread-local event loop if present to avoid blocking it (but don't create one)
    eventLoop = (contextInterceptor as? EventLoop)?.takeIf { it.shouldBeProcessedFromContext() }
        ?: ThreadLocalEventLoop.currentOrNull()
    newContext = GlobalScope.newCoroutineContext(context)
}
```

# spring coroutine – runBlocking

runBlocking에서 CoroutineContext를 추가하지 않는 경우 내부적으로 eventLoop가 실행

```
fun main() {
    val dispatchers = Executors.newFixedThreadPool( nThreads: 1 )
        .asCoroutineDispatcher()
    runBlocking(dispatchers) {
        println("$coroutineContext")
        runBlocking {
            println("$coroutineContext")
        }
    }
    dispatchers.close()
}
```

```
[CoroutineId(1), "coroutine#1":BlockingCoroutine{Active}@7183ba32, java.util.concurrent
.ThreadPoolExecutor@22226935[Running, pool size = 1, active threads = 1, queued tasks = 0, completed tasks = 0]]

[CoroutineId(2), "coroutine#2":BlockingCoroutine{Active}@32bb56dd, BlockingEventLoop@627ac08d
```

# spring coroutine – runBlocking

## runBlocking

concurrent

```
expect fun <T> runBlocking(
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> T
): T
```

(source)

Runs a new coroutine and **blocks** the current thread until its completion.

It is designed to bridge regular blocking code to libraries that are written in suspending style, to be used in `main` functions and in tests.

Calling `runBlocking` from a suspend function is redundant. For example, the following code is incorrect:

```
suspend fun loadConfiguration() {
    // DO NOT DO THIS:
    val data = runBlocking { // ← redundant and blocks the thread, do not do that
        fetchConfigurationData() // suspending function
    }
}
```

Here, instead of releasing the thread on which `loadConfiguration` runs if `fetchConfigurationData` suspends, it will block, potentially leading to thread starvation issues.

# spring coroutine – unstructured concurrency

구조적 동시성을 깬다면 예외를 컨트롤할 수 없으므로 CoroutineExceptionHandler로 보완하여야함

```
fun getArticleById(id: Long): ArticleResponse = runBlocking(Dispatchers.IO) {
    val article = async() {...}
    val comments = async() {...}
    return@runBlocking ArticleResponse.of(article.await(), comments.await())
        .also { CoroutineScope(context: Dispatchers.IO + coroutineExceptionHandler()) launch {
            log.info("logging")
        } }
}

private fun coroutineExceptionHandler() = CoroutineExceptionHandler { _, exception ->
    log.warn("exception")
}
```

# spring coroutine – meltdown

각 Blocking 작업이 Dispatchers.IO의 worker 스레드를 점유하고 있음

```
fun getArticleById(id: Long): ArticleResponse = runBlocking(Dispatchers.IO) {
    val article = async {
        log.info("articleQueryService")
        articleRepository.findById(id).orElseThrow()
    }
    val comments = async {
        log.info("findByArticleId")
        commentRepository.findAllByArticleId(id)
    }

    return@runBlocking ArticleResponse.of(article.await(), comments.await())
}
```

# spring coroutine – CoroutineCrudRepository

```
@org.springframework.data.repository.NoRepositoryBean public interface CoroutineCrudRepository<T, ID> : org.springframework.data.repository.Repository<T, ID> {
    public abstract suspend fun count(): kotlin.Long

    public abstract suspend fun delete(entity: T): kotlin.Unit

    public abstract suspend fun deleteAll(): kotlin.Unit

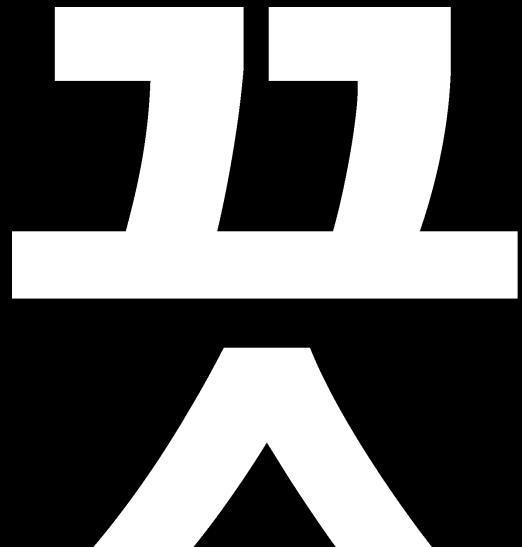
    public abstract suspend fun deleteAll(entities: kotlin.collections.Iterable<T>): kotlin.Unit

    public abstract suspend fun <S : T> deleteAll(entityStream: kotlinx.coroutines.flow.Flow<S>): kotlin.Unit

    public abstract suspend fun deleteAllById(ids: kotlin.collections.Iterable<ID>): kotlin.Unit

    public abstract suspend fun deleteById(id: ID): kotlin.Unit

    public abstract suspend fun existsById(id: ID): kotlin.Boolean
}
```



우아한형제들