4주차

1. Continuation

Continuation의 개념

`Continuation`은 코루틴이 일시 중단된 이후에 실행을 재개할 수 있는 지점을 나타냅니다. 코루틴이 일시 중단될 때, 현재 상태와 필요한 컨텍스트가 `Continuation` 객체에 저장됩니다. 이후에 이 객체를 통해 코루틴을 재개할 수 있습니다.

Continuation 인터페이스

`Continuation` 인터페이스는 다음과 같이 정의됩니다:

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resumeWith(result: Result<T>)
}
```

- `context`: 코루틴의 실행 컨텍스트를 나타냅니다. 이는 코루틴 디스패처와 같은 정보를 포함할 수 있습니다.
- `resumeWith(result: Result<T>)`: 코루틴을 재개하는 함수입니다. `Result` 객체를 통해 성공 또는 실패 결과를 전달합니다.

관련 함수들

코루틴과 관련된 몇 가지 중요한 함수와 개념들을 살펴보겠습니다.

suspend 함수

`suspend` 키워드는 함수가 일시 중단될 수 있음을 나타내며, 이는 코루틴 내에서만 호출될 수 있습니다. 예를 들어:

```
suspend fun fetchData(): String {
    // Some long-running operation
    return "Data"
}
```

withContext

`withContext` 함수는 다른 코루틴 컨텍스트에서 코드를 실행할 수 있게 합니다. 예를 들어:

```
import kotlinx.coroutines.*

suspend fun fetchData(): String {
    return withContext(Dispatchers.IO) {
        // Perform IO operation
        "Data"
    }
}
```

Continuation을 직접 사용하기

일반적으로 `Continuation`을 직접 사용할 필요는 없지만, 코루틴의 내부 동작을 이해하기 위해 사용할 수 있습니다. 예를 들어, `suspendCoroutine` 함수를 사용하여 `Continuation`을 직접 다룰 수 있습니다:

```
import kotlin.coroutines.*

suspend fun <T> suspendCoroutineExample(block: (Continuation<T>) -> Unit): T
=
    suspendCoroutine { continuation ->
        block(continuation)
    }
```

이 함수는 일시 중단된 후 `Continuation` 객체를 `block`에 전달합니다. `block`은 나중에 `continuation.resumeWith`을 호출하여 코루틴을 재개할 수 있습니다.

예제

다음은 `Continuation`을 사용하는 간단한 예제입니다:

```
import kotlin.coroutines.*

suspend fun main() {
    val result = suspendCoroutine<String> { continuation ->
        continuation.resumeWith(Result.success("Hello, Continuation!"))
    }
    println(result)
}
```

2. Dispatchers

코틀린(Kotlin)에서 코루틴(Coroutines)은 여러 가지 컨텍스트에서 실행될 수 있으며, 이를 관리하기 위해 `Dispatchers`가 사용됩니다. `Dispatchers`는 코루틴이 어떤 스레드 또는 스레드 풀에서 실행될지를 결정합니다. 코틀린 표준 라이브러리와 `kotlinx.coroutines` 라이브러리는 여러 가지 디스패처를 제공합니다.

주요 Dispatchers 종류

1. **Dispatchers.Default**
2. **Dispatchers.IO**
3. **Dispatchers.Main**
4. **Dispatchers.Unconfined**

각 디스패처는 특정한 용도와 최적화된 환경에서 사용됩니다.

Dispatchers.Default

`Dispatchers.Default`는 CPU 집약적인 작업을 위한 디스패처입니다. 기본적으로 코어 수에 비례하는 스레드 풀을 사용합니다. 이는 병렬 처리가 필요한 연산에 적합합니다.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch(Dispatchers.Default) {
        println("Running on Default: ${Thread.currentThread().name}")
    }
}
```

Dispatchers.IO

`Dispatchers.IO`는 IO 작업을 위한 디스패처입니다. 파일 읽기/쓰기, 네트워크 요청 등 블로킹 IO 작업에 최적화되어 있습니다. 이 디스패처는 기본적으로 많은 수의 스레드를 사용하여 블로킹 작업을 처리합니다.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch(Dispatchers.IO) {
        println("Running on IO: ${Thread.currentThread().name}")
    }
}
```

Dispatchers.Main

`Dispatchers.Main`은 안드로이드와 같은 UI 스레드에서 실행되는 작업을 위한 디스패처입니다. UI 업데이트와 같은 작업에 사용됩니다. 이 디스패처는 안드로이드나 JavaFX와 같은 플랫폼에서만 사용할 수 있습니다.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch(Dispatchers.Main) {
        println("Running on Main: ${Thread.currentThread().name}")
    }
}
```

이 예제는 안드로이드 환경에서 실행되어야 합니다.

Dispatchers.Unconfined

`Dispatchers.Unconfined`는 특정 스레드에 바인딩되지 않는 디스패처입니다. 처음에는 호출한 스레드에서 실행되지만, 일시 중단 후에는 일시 중단된 함수가 재개되는 스레드에서 실행됩니다. 주로 테스트나 특정한 컨텍스트 전환을 피하고 싶을 때 사용됩니다.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch(Dispatchers.Unconfined) {
        println("Running on Unconfined: ${Thread.currentThread().name}")
    }
}
```

Dispatchers 사용 예제

다음은 다양한 디스패처를 사용하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch(Dispatchers.Default) {
        println("Default: Running on ${Thread.currentThread().name}")
    }

    launch(Dispatchers.IO) {
        println("IO: Running on ${Thread.currentThread().name}")
    }
```

```
    launch(Dispatchers.Unconfined) {
        println("Unconfined: Running on ${Thread.currentThread().name}")
        delay(100)
        println("Unconfined after delay: Running on
${Thread.currentThread().name}")
    }

    // Dispatchers.Main은 안드로이드 환경에서만 사용 가능
    // launch(Dispatchers.Main) {
    //     println("Main: Running on ${Thread.currentThread().name}")
    // }
}
```

3. Async

코틀린(Kotlin)에서 `async`는 비동기적으로 작업을 수행하고 결과를 반환하는 코루틴 빌더입니다. `async`는 `launch`와 비슷하지만, `launch`는 결과를 반환하지 않는 반면, `async`는 `Deferred` 객체를 반환하여 나중에 결과를 얻을 수 있습니다. `Deferred`는 `Job`의 하위 클래스이며, 비동기 작업의 결과를 나타냅니다.

async의 기본 사용법

`async`는 비동기적으로 작업을 수행하고, 결과를 기다리기 위해 `await` 함수를 사용합니다. 다음은 기본적인 사용 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        // Some long-running computation
        delay(1000L)
        42
    }

    // Do some other work here if needed

    val result = deferred.await()  // Wait for the result
    println("Result: $result")
}
```

위 예제에서 `async` 블록은 백그라운드에서 실행되며, `deferred` 객체를 반환합니다. `deferred.await()`를 호출하면 결과가 준비될 때까지 기다립니다.

async와 await를 사용한 병렬 처리

`async`와 `await`를 사용하면 여러 비동기 작업을 병렬로 실행할 수 있습니다. 다음은 두 개의 비동기 작업을 병렬로 실행하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred1: Deferred<Int> = async {
        delay(1000L)
        10
    }

    val deferred2: Deferred<Int> = async {
        delay(2000L)
        20
    }

    // Wait for both results
    val result1 = deferred1.await()
    val result2 = deferred2.await()

    println("Result 1: $result1")
    println("Result 2: $result2")
}
```

위 예제에서 두 `async` 블록은 병렬로 실행되며, 각각 1초와 2초 동안 일시 중단됩니다. `await`를 사용하여 두 결과를 모두 기다린 후 출력합니다.

async와 CoroutineScope

`async`는 `CoroutineScope` 내에서 호출되어야 합니다. `runBlocking`, `launch`, `async` 등은 모두 `CoroutineScope`를 생성합니다. 다음은 `CoroutineScope`를 명시적으로 사용하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val scope = CoroutineScope(Dispatchers.Default)

    val deferred = scope.async {
        delay(1000L)
        "Hello, World!"
    }

    val result = deferred.await()
```

```
    println(result)
}
```

예외 처리

`async` 블록 내에서 발생한 예외는 `await`를 호출할 때 전파됩니다. 따라서 `await`를 호출할 때 예외 처리를 할 수 있습니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred = async {
        throw Exception("Something went wrong")
    }

    try {
        deferred.await()
    } catch (e: Exception) {
        println("Caught exception: ${e.message}")
    }
}
```

async와 구조적 동시성

코루틴은 구조적 동시성을 지원하며, 이는 부모 코루틴이 완료될 때 자식 코루틴도 함께 취소됨을 의미합니다. `async`도 이 원칙을 따릅니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        val deferred = async {
            delay(1000L)
            42
        }
        println("Result: ${deferred.await()}")
    }
    delay(500L)
    job.cancel()  // Cancels the parent job and its children
}
```

위 예제에서 `job.cancel()`을 호출하면 `launch` 블록과 그 안의 `async` 블록이 모두 취소됩니다.

4. Launch

코틀린(Kotlin)에서 `launch`는 코루틴을 시작하는 가장 기본적인 방법 중 하나입니다. `launch`는 `Job` 객체를 반환하며, 비동기적으로 작업을 수행하지만 결과를 반환하지 않습니다. 이는 주로 실행할 작업이 결과를 반환할 필요가 없고, 단순히 비동기적으로 수행되어야 할 때 사용됩니다.

launch의 기본 사용법

`launch`는 `CoroutineScope` 내에서 호출되어야 합니다. 가장 일반적인 방법은 `runBlocking`을 사용하여 최상위 코루틴을 시작하는 것입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        // 코루틴 블록 내에서 실행될 코드
        delay(1000L)
        println("Hello from Coroutine!")
    }
    println("Hello from Main!")
}
```

위 예제에서 `launch` 블록은 비동기적으로 실행되며, `delay(1000L)`로 1초 동안 일시 중단됩니다. 그 동안 `println("Hello from Main!")`은 즉시 실행됩니다.

launch와 CoroutineScope

`launch`는 `CoroutineScope` 내에서 호출되어야 합니다. `runBlocking`, `launch`, `async` 등은 모두 `CoroutineScope`를 생성합니다. 다음은 `CoroutineScope`를 명시적으로 사용하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val scope = CoroutineScope(Dispatchers.Default)

    scope.launch {
        delay(1000L)
        println("Hello from Coroutine!")
    }

    println("Hello from Main!")
}
```

launch와 Dispatchers

`launch`는 `Dispatchers`를 사용하여 코루틴이 실행될 스레드 또는 스레드 풀을 지정할 수 있습니다. 다음은 다양한 `Dispatchers`를 사용하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch(Dispatchers.Default) {
        println("Running on Default: ${Thread.currentThread().name}")
    }

    launch(Dispatchers.IO) {
        println("Running on IO: ${Thread.currentThread().name}")
    }

    launch(Dispatchers.Unconfined) {
        println("Running on Unconfined: ${Thread.currentThread().name}")
    }

    // Dispatchers.Main은 안드로이드 환경에서만 사용 가능
    // launch(Dispatchers.Main) {
    //     println("Running on Main: ${Thread.currentThread().name}")
    // }
}
```

Job 객체

`launch`는 `Job` 객체를 반환합니다. `Job` 객체를 사용하여 코루틴을 제어할 수 있습니다. 예를 들어, 코루틴을 취소하거나 완료 여부를 확인할 수 있습니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job: Job = launch {
        delay(1000L)
        println("Hello from Coroutine!")
    }

    println("Job is active: ${job.isActive}")
    job.join()  // Wait for the coroutine to complete
    println("Job is completed: ${job.isCompleted}")
}
```

예외 처리

`launch` 블록 내에서 발생한 예외는 부모 코루틴에 전파됩니다. 예외 처리를 위해 `try-catch` 블록을 사용할 수 있습니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        try {
            throw Exception("Something went wrong")
        } catch (e: Exception) {
            println("Caught exception: ${e.message}")
        }
    }
    job.join()
}
```

구조적 동시성

코루틴은 구조적 동시성을 지원하며, 이는 부모 코루틴이 완료될 때 자식 코루틴도 함께 취소됨을 의미합니다. `launch` 도 이 원칙을 따릅니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val parentJob = launch {
        val childJob = launch {
            delay(1000L)
            println("Hello from Child Coroutine!")
        }
        delay(500L)
        println("Hello from Parent Coroutine!")
    }
    delay(300L)
    parentJob.cancel()  // Cancels the parent job and its children
}
```

위 예제에서 `parentJob.cancel()` 을 호출하면 부모 코루틴과 그 안의 자식 코루틴이 모두 취소됩니다.

5. Yield

코틀린(Kotlin)에서 `yield`는 코루틴의 실행을 일시 중단하고, 다른 코루틴이나 작업이 실행될 기회를 주는 함수입니다. `yield`는 협력적 멀티태스킹을 구현하는 데 사용됩니다. 이는 코루틴이 자신의 실행을 중단하고 다른 코루틴이 실행될 수 있도록 함으로써, 공정한 스케줄링을 가능하게 합니다.

yield의 기본 사용법

`yield`는 `suspend` 함수로, 코루틴 내에서만 호출될 수 있습니다. 다음은 `yield`를 사용하는 간단한 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        repeat(5) {
            println("Coroutine A - $it")
            yield()  // Yield execution to other coroutines
        }
    }

    launch {
        repeat(5) {
            println("Coroutine B - $it")
            yield()  // Yield execution to other coroutines
        }
    }
}
```

위 예제에서 두 코루틴이 `yield`를 사용하여 실행을 서로 번갈아 가며 수행합니다. `yield`는 현재 코루틴의 실행을 일시 중단하고, 다른 코루틴이 실행될 기회를 줍니다.

yield와 협력적 멀티태스킹

`yield`는 협력적 멀티태스킹을 구현하는 데 유용합니다. 협력적 멀티태스킹에서는 각 작업이 명시적으로 자신의 실행을 일시 중단하고, 다른 작업이 실행될 수 있도록 합니다. 이는 공정한 스케줄링을 가능하게 하며, 특정 코루틴이 CPU 시간을 독점하는 것을 방지합니다.

다음은 `yield`를 사용하여 협력적 멀티태스킹을 구현하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job1 = launch {
        for (i in 1..5) {
            println("Job 1 - $i")
```

```
            yield()  // Yield execution to other coroutines
        }
    }

    val job2 = launch {
        for (i in 1..5) {
            println("Job 2 - $i")
            yield()  // Yield execution to other coroutines
        }
    }

    joinAll(job1, job2)  // Wait for both jobs to complete
}
```

위 예제에서 두 개의 코루틴이 `yield`를 사용하여 번갈아 가며 실행됩니다. 각 코루틴은 자신의 실행을 일시 중단하고, 다른 코루틴이 실행될 수 있도록 합니다.

yield와 Dispatchers

`yield`는 특정 디스패처와 함께 사용될 때 더욱 유용할 수 있습니다. 예를 들어, `Dispatchers.Default`와 함께 사용하면 CPU 집약적인 작업을 공정하게 분배할 수 있습니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job1 = launch(Dispatchers.Default) {
        for (i in 1..5) {
            println("Job 1 - $i on ${Thread.currentThread().name}")
            yield()  // Yield execution to other coroutines
        }
    }

    val job2 = launch(Dispatchers.Default) {
        for (i in 1..5) {
            println("Job 2 - $i on ${Thread.currentThread().name}")
            yield()  // Yield execution to other coroutines
        }
    }

    joinAll(job1, job2)  // Wait for both jobs to complete
}
```

위 예제에서 두 개의 코루틴이 `Dispatchers.Default`에서 실행되며, `yield`를 사용하여 공정하게 CPU 시간을 분배합니다.

6. runBlocking

코틀린(Kotlin)에서 `runBlocking`은 코루틴을 시작하고, 해당 코루틴이 완료될 때까지 현재 스레드를 차단하는 함수입니다. 이는 주로 메인 함수나 테스트 코드에서 사용되며, 코루틴의 실행을 동기적으로 기다려야 할 때 유용합니다. `runBlocking`은 코루틴 빌더 중 하나로, 블로킹 방식으로 코루틴을 실행합니다.

runBlocking의 기본 사용법

`runBlocking`을 사용하여 코루틴을 시작하고, 해당 코루틴이 완료될 때까지 현재 스레드를 차단할 수 있습니다. 다음은 `runBlocking`의 기본 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Hello from Coroutine!")
    }
    println("Hello from Main!")
}
```

위 예제에서 `runBlocking` 블록 내에서 `launch`를 사용하여 코루틴을 시작합니다. `delay(1000L)`로 인해 1초 동안 일시 중단된 후 "Hello from Coroutine!" 메시지가 출력됩니다. `runBlocking`은 코루틴이 완료될 때까지 현재 스레드를 차단하므로, "Hello from Main!" 메시지가 먼저 출력되고 나서 코루틴의 메시지가 출력됩니다.

runBlocking과 CoroutineScope

`runBlocking`은 `CoroutineScope`를 생성하며, 이 스코프 내에서 코루틴을 시작할 수 있습니다. `runBlocking` 블록 내에서 여러 개의 코루틴을 시작할 수 있습니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Coroutine 1")
    }

    launch {
        delay(500L)
        println("Coroutine 2")
    }
```

```
    println("Hello from Main!")
}
```

위 예제에서 두 개의 코루틴이 `runBlocking` 블록 내에서 시작됩니다. 각 코루틴은 서로 다른 지연 시간을 가지며, "Hello from Main!" 메시지가 먼저 출력된 후 코루틴의 메시지가 출력됩니다.

runBlocking과 Dispatchers

`runBlocking`은 기본적으로 호출된 스레드에서 실행되지만, `Dispatchers`를 사용하여 다른 스레드에서 실행될 수 있습니다. 다음은 `Dispatchers`를 사용하는 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking(Dispatchers.Default) {
    launch {
        println("Running on Default: ${Thread.currentThread().name}")
    }

    launch(Dispatchers.IO) {
        println("Running on IO: ${Thread.currentThread().name}")
    }
}
```

위 예제에서 `runBlocking`은 `Dispatchers.Default`에서 실행되며, 두 개의 코루틴이 각각 `Dispatchers.Default`와 `Dispatchers.IO`에서 실행됩니다.

runBlocking과 예외 처리

`runBlocking` 블록 내에서 발생한 예외는 호출한 스레드로 전파됩니다. 예외 처리를 위해 `try-catch` 블록을 사용할 수 있습니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        launch {
            throw Exception("Something went wrong")
        }.join()
    } catch (e: Exception) {
        println("Caught exception: ${e.message}")
    }
}
```

위 예제에서 `launch` 블록 내에서 예외가 발생하면, `try-catch` 블록을 통해 예외를 처리할 수 있습니다.

runBlocking과 구조적 동시성

코루틴은 구조적 동시성을 지원하며, 이는 부모 코루틴이 완료될 때 자식 코루틴도 함께 취소됨을 의미합니다. `runBlocking`도 이 원칙을 따릅니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val parentJob = launch {
        val childJob = launch {
            delay(1000L)
            println("Hello from Child Coroutine!")
        }
        delay(500L)
        println("Hello from Parent Coroutine!")
    }
    delay(300L)
    parentJob.cancel()  // Cancels the parent job and its children
}
```

위 예제에서 `parentJob.cancel()`을 호출하면 부모 코루틴과 그 안의 자식 코루틴이 모두 취소됩니다.

runBlocking의 주의사항

`runBlocking`은 주로 메인 함수나 테스트 코드에서 사용되며, 일반적인 애플리케이션 코드에서는 자주 사용되지 않습니다. 이는 `runBlocking`이 현재 스레드를 차단하기 때문에, UI 스레드를 차단하는 등의 부작용을 초래할 수 있기 때문입니다. 따라서, `runBlocking`은 주로 동기적으로 코루틴을 실행해야 하는 상황에서만 사용해야 합니다.

7. withContext

코틀린(Kotlin)에서 `withContext`는 코루틴의 컨텍스트를 일시적으로 변경하여 특정 블록을 실행하는 데 사용되는 함수입니다. 이는 주로 다른 디스패처(Dispatcher)에서 코드를 실행하고 싶을 때 사용됩니다. `withContext`는 `suspend` 함수이기 때문에 코루틴 내에서만 호출될 수 있으며, 코루틴의 일시 중단과 재개를 처리합니다.

withContext의 기본 사용법

`withContext`를 사용하면 코드 블록을 지정된 디스패처에서 실행할 수 있습니다. 예를 들어, IO 작업을 `Dispatchers.IO`에서 실행하고, CPU 집약적인 작업을 `Dispatchers.Default`에서 실행할 수 있습니다.

다음은 `withContext`의 기본 예제입니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    println("Running in context: ${coroutineContext}")

    withContext(Dispatchers.Default) {
        println("Running in context: ${coroutineContext}")
    }

    withContext(Dispatchers.IO) {
        println("Running in context: ${coroutineContext}")
    }
}
```

위 예제에서 `runBlocking` 블록은 기본 컨텍스트에서 실행됩니다. `withContext(Dispatchers.Default)`와 `withContext(Dispatchers.IO)`를 사용하여 각각 다른 디스패처에서 코드 블록을 실행합니다.

withContext와 비동기 작업

`withContext`는 비동기 작업을 다른 디스패처에서 실행하고, 결과를 반환받을 때 유용합니다. 예를 들어, 네트워크 요청이나 파일 읽기/쓰기와 같은 IO 작업을 `Dispatchers.IO`에서 실행할 수 있습니다:

```
import kotlinx.coroutines.*

suspend fun fetchData(): String = withContext(Dispatchers.IO) {
    // Simulate a long-running IO operation
    delay(1000L)
    "Data from network"
}

fun main() = runBlocking {
    println("Fetching data...")
    val data = fetchData()
    println("Received data: $data")
}
```

위 예제에서 `fetchData` 함수는 `Dispatchers.IO`에서 실행되며, 1초 동안 일시 중단된 후 데이터를 반환합니다. `runBlocking` 블록은 `fetchData` 함수의 결과를 기다렸다가 출력합니다.

withContext와 예외 처리

`withContext` 블록 내에서 발생한 예외는 호출한 코루틴으로 전파됩니다. 예외 처리를 위해 `try-catch` 블록을 사용할 수 있습니다:

```
import kotlinx.coroutines.*

suspend fun riskyOperation(): String = withContext(Dispatchers.Default) {
    // Simulate an operation that might throw an exception
    if (Math.random() < 0.5) {
        throw Exception("Something went wrong")
    }
    "Successful operation"
}

fun main() = runBlocking {
    try {
        val result = riskyOperation()
        println("Operation result: $result")
    } catch (e: Exception) {
        println("Caught exception: ${e.message}")
    }
}
```

위 예제에서 `riskyOperation` 함수는 `Dispatchers.Default`에서 실행되며, 예외가 발생할 수 있습니다. `runBlocking` 블록 내에서 예외를 처리하여 안전하게 실행할 수 있습니다.

withContext와 구조적 동시성

`withContext`는 구조적 동시성을 지원합니다. 이는 부모 코루틴이 완료될 때 자식 코루틴도 함께 취소됨을 의미합니다. `withContext` 블록은 부모 코루틴의 구조적 동시성을 유지합니다:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        withContext(Dispatchers.Default) {
            delay(1000L)
            println("Hello from withContext!")
        }
    }
    delay(500L)
    job.cancel()  // Cancels the parent job and its children
    println("Job cancelled")
}
```

위 예제에서 `job.cancel()`을 호출하면 `withContext` 블록 내의 작업도 함께 취소됩니다.