Marcin Moskała
Çeviri ve Editöryal Katkı: Hasan Tunçay

Kotlin Coroutines
TÜM DETAYLARIYLA
Üçüncü Baskı

Asenkron programlamanın temellerinden
ileri seviye kullanım senaryolarına
uzanan kapsamlı bir kaynak

Kotlin Coroutines

Tüm Detaylarıyla

Marcin Moskała ve Hasan Tunçay

Bu kitap https://leanpub.com/kotlincoroutinestrke adresinde satılmaktadır

Bu versiyon 2025-05-20 tarihinde yayınlandı

Bu bir Leanpub kitabıdır. Leanpub, yazarlara ve yayıncılara Yalın Yayıncılık süreciyle güç kazandırır. Yalın Yayıncılık, okuyucu geri bildirimi almak için hafif araçlar ve birçok iterasyon kullanarak devam eden bir e-kitabı yayınlama, doğru kitaba ulaşana kadar yön değiştirme ve başardığınızda ivme kazanma sürecidir.

İçindekiler

Giriş **1**
Kitap kime hitap ediyor? 2
Kitabın bölümleri 2
Konuların kapsamı 2
Geliştiriciler İçin Kotlin Serisi 3
Kurallar ve Kılavuzlar 3
Kod Kuralları 4
Sürüm 5
Alıştırmalar ve çözümler 6
Teşekkürler 7

Bölüm 1: Kotlin Coroutines'i Anlamak **13**

Neden Kotlin Coroutines? **14**
Basitlik (Simplicity) 15
Performans 16
İptal Mekanizması (Cancellation) 18
Eşzamanlama (Synchronization) 19
Test Edilebilirlik (Testability) 21
Flow 22
Coroutines Çoklu Platform Desteğine Sahiptir 24
Kotlin Coroutines ile İlgili En Büyük Problem 24
Sonuç 24

Sequence Builder (Dizi Üretici) **26**
Gerçek hayattaki kullanım alanları 26
Alıştırma: Faktöriyel Dizisi 26
Alıştırma: Asal Sayılar Dizisi 26

Askıya Alma (Suspension) Mekanizması Nasıl Çalışır? **27**
Oyun Metaforu Üzerinden Thread ve Coroutine Karşılaştırması 27
Askıya Alınabilen (Suspending) Fonksiyonlar 27
İlk Askıya Alma (Suspension) İşlemi 27
Devam Nesnesinde (`Continuation`) Neler Saklanır? 27
Coroutine'i Geciktirme 27
Bir Değer ile Devam Etme 27
İstisna (Exception) ile Devam Ettirme (Resume) 28
Askıya alınan Coroutine'dir , fonksiyon değil 28
Özet 28
Alıştırma: Geri Çağırma (Callback) Fonksiyon Sarmalayıcıları 28
Alıştırma: Devam (Continuation) Saklama 28

Coroutine Mekanizmasının İç Yapısı **29**
Devam Nesnesi Geçişli Programlama (Continuation-Passing Style) . . 29
Çok Basit Bir Fonksiyon 29
Durum Bilgisine Sahip Bir Fonksiyon 29
Bir Değer ile Devam Ettirilen Fonksiyon 29
Çağrı Yığını (Call Stack) 29
Farklı Bağlamlarda Askıya Alınan Fonksiyonlar 29
Gerçek Kod 30
Askıya Alınan Fonksiyonların Performansı 30
Özet 30
Alıştırma: Bir Devam Nesnesi (Continuation) Neleri Saklar? 30

Coroutine'ler: Yerleşik Destek(built-in support) vs Kütüphane(library) **31**

Bölüm 2: Kotlin Coroutines Kütüphanesi . 32

Eşzamanlı İş Parçacıkları(Coroutines) Nasıl Başlatılır? **33**

Asenkron Eşzamanlı İş Parçacığı Başlatıcıları **34**
Engelleyici Eşzamanlı İş Parçacığı Başlatıcıları (Blocking Coroutine Builders) 34
Yapısal Eşzamanlılık (Structured Concurrency) 34

Coroutine Kapsamı ve Coroutine Başlatıcıları Arasındaki Farklar **35**
Özet 35
Alıştırma: `UserDetailsRepository` 35

Alıştırma: `BestStudentUseCase` 35
Alıştırma: CommentService 35
Alıştırma: mapAsync 35

Genel Bakış **36**

Coroutine Bağlamı (Coroutine Context) **37**
`CoroutineContext` Arayüzü(`interface`) 37
`CoroutineContext` İçinde Öğe Bulma 37
Bağlam (Context) Ekleme 37
Boş İşlem Kapsamı (Empty Coroutine Context) 37
Öğeleri Çıkartma (Subtracting Elements) 37
Kapsamı Katlama (Folding Context) 37
Coroutine Bağlamı ve Başlatıcılar 38
Askıya Alınabilir Bir Fonksiyonda Bağlama (Context) Erişim 38
Coroutine Kapsam Fonksiyonları ve Bağlam Mirası 38
Askıya Alınabilir Fonksiyonlarda Bağlamı Değiştirme 38
Kendi İşlem Bağlamımızı Oluşturma 38
Eşzamanlı İşlemler ve İş Parçacığı Öğeleri 38
Özet 39
Alıştırma: Bağlam Yayılımını Anlamak 39
Alıştırma: CounterContext 39

Coroutine Yöneticileri (Dispatchers) **40**
Varsayılan Coroutine Yöneticisi (Default Dispatcher) 40
Varsayılan Dağıtıcıyı (Dispatcher) Sınırlama 41
Ana Dağıtıcı (Main Dispatcher) 42
Giriş/Çıkış (GÇ/IO) Dağıtıcısı 44
Özel Sınırlamalı Dağıtıcı 49
Sabit İş Parçacığı Havuzuna Sahip Dağıtıcı (Dispatcher) 54
Tek İş Parçacığı ile Sınırlı Dağıtıcı (Dispatcher) 55
Project Loom'dan Gelen Sanal İş Parçacıklarını Kullanma 57
Bağlı Olmayan Dağıtıcı (Unconfined Dispatcher) 60
Anında Ana İş Parçacığı Görevlendirmesi (Immediate Main Dispatching) 61
Devamlılık (Continuation) Kesici 63
Dağıtıcıların farklı görevlerdeki performansı 65
Özet 67
Alıştırma: Dağıtıcıları Kullanma 69
Alıştırma: DiscNewsRepository 71

Alıştırma: Dağıtıcılarla Deneyler 71
Dağıtıcıların Performans Testi 72

Job ve Coroutine Yaşam Döngüsü **74**
Job ve Coroutine İlişkileri 74
Coroutine Yaşam Döngüsü 74
Job'un Tamamlanmasını Bekleme 74
Job Fabrika Fonksiyonu 74
Coroutineleri Senkronize Etme 74
Özet 74

İptal Mekanizması(Cancellation) **76**
Temel İptal Mekanizması 76
`finally` Bloğu 76
invokeOnCompletion 76
Alt Coroutine'lerin İptali 76
Coroutine Kapsamında İptal İşlemi 76
Sadece Bir Çağrı Daha 76
Durdurulamaz Olanı Durdurmak 77
CancellationException'in Özel Durumu 77
CancellationException Üst Coroutine'e Yayılmaz 77
withTimeout 77
`suspendCancellableCoroutine` 77
Özet 77
Alıştırma: İptal Mekanizmasıyla İlgili Hataları Düzeltme 77

İstisna Yönetimi **79**
İstisnalar ve Yapılandırılmış Eşzamanlılık(structured concurrency) . . 79
SupervisorJob 79
supervisorScope 79
İstisnalar ve await Çağrısı 79
`CoroutineExceptionHandler` 80
Özet 80

Bir coroutine scope (iş kapsamı) oluşturma **81**
CoroutineScope fabrika fonksiyonu 81
Arka plan kapsamı (background scope) oluşturma 81
Android'de scope oluşturma 81
Özet 81
Alıştırma: NotificationSender 81

Alıştırma: BaseViewModel 81

Değiştirilebilir duruma(mutable state) erişimi eşzamanlama (synchronizing) **83**
Atomik değerlerin kullanımı (Using atomic values) 83
Synchronized bloklar (Synchronized blocks) 83
Tek iş parçacığıyla sınırlı bir dispatcher kullanımı 83
Mutex 83
Semaphore 84
Özet 84
Alıştırma: CompanyDetailsRepository 84
Alıştırma: CancellingRefresher 84
Alıştırma: TokenRepository 85
Alıştırma: Suspended lazy 85
Alıştırma: mapAsync ile eşzamanlılık sınırı (concurrency limit) 85

Kotlin Coroutines Testi **86**
Zaman Bağımlılıklarının Test Edilmesi (Testing time dependencies) . . 86
`TestCoroutineScheduler` ve `StandardTestDispatcher` 86
runTest 86
Arka plan scope'u (Background scope) 86
İptal (cancellation) ve bağlam aktarımının (context passing) test edilmesi 86
`UnconfinedTestDispatcher` 86
Mock nesnelerin kullanımı (Using mocks) 87
Dispatcher değiştiren fonksiyonların test edilmesi 87
Fonksiyon çalışırken gerçekleşen davranışların test edilmesi 87
Yeni coroutine başlatan fonksiyonların test edilmesi 87
runTest içinde scope gerektiren sınıfların test edilmesi 87
Ana dispatcher'ın (Main dispatcher) değiştirilmesi 87
Coroutine başlatan Android fonksiyonlarının test edilmesi 88
Test dispatcher'ı bir rule ile ayarlamak 88
Özet 88
Alıştırma: UserDetailsRepository Sınıfını Test Etme 88
Alıştırma: mapAsync Fonksiyonunun Test Edilmesi 88
Alıştırma: NotificationSender Sınıfını Test Etme 88
Alıştırma: Bir ViewModel Sınıfını Test Etme 89

Bölüm 3: Channel ve Flow 90

Channel **91**
Kanal Türleri (Channel Types) 91
Arabellek Taşması Durumunda (On Buffer Overflow) 91
Teslim Edilemeyen Öğeler İçin İşleyici (On Undelivered Element Handler) 91
Fan-out (Yayılma Modeli) 91
Fan-in (Toplanma Modeli) 91
Pipelines (Veri İşleme Hatları) 91
Pratik Kullanım (Practical Usage) 92
Özet 92
Alıştırma: UserRefresher 92
Alıştırma: Kafeterya simülasyonu 92

Select (Seçim) **93**
Ertelenmiş (deferred) değerleri seçme 93
Channel'lardan Seçim Yapma (Selecting from channels) 93
Özet 93
Alıştırma: raceOf 93

Sıcak (Hot) ve Soğuk (Cold) Veri Kaynakları **94**

Sıcak ve Soğuk Akışlar **95**
Sıcak Channel, Soğuk Flow 95
Özet 95

Flow'a Giriş **96**
Flow'un Diğer Değer Temsil Yöntemleriyle Karşılaştırılması 96
Flow'un Özellikleri 96
Flow Terimlendirmesi (Nomenclature) 96
Gerçek Hayattaki Kullanım Durumları 96
Özet 96

`Flow`'u Anlamak **97**
`Flow`'u Anlamak 97
`Flow` İşlemenin (Processing) Nasıl Çalıştığı 104
Flow Doğası Gereği Eşzamanlıdır (Synchronous) 107
Flow ve Paylaşılan Durum (Shared State) 108
Sonuç 112

Flow Oluşturma (Flow Building) **113**
Ham Değerlerden Flow Oluşturma (Flow from raw values) 113
Dönüştürücüler (Converters) 113
Bir Fonksiyonu Flow'a Dönüştürme 113
Flow ve Reactive Streams 113
Flow Oluşturucular (Flow Builders) 113
Flow Oluşturucuyu Anlamak (Understanding flow builder) 113
channelFlow 114
callbackFlow 114
Özet 114
Alıştırma: Flow Yardımcıları (Flow utils) 114
Alıştırma: Tüm Kullanıcıları Veren Flow (All users flow) 114
Alıştırma: distinct 114

Flow Yaşam Döngüsü Fonksiyonları (Flow lifecycle functions) **115**
onEach 115
onStart 115
onCompletion 115
onEmpty 115
catch 115
Yakalanmamış İstisnalar (Uncaught exceptions) 115
`retry` 116
flowOn 116
launchIn 116
Özet 116
Alıştırma: TemperatureService 116
Alıştırma: NewsViewModel 116

Akış (Flow) İşleme **117**
`map` 117
`filter` 117
`take` and `drop` 117
Koleksiyon İşleme (Collection Processing) Nasıl Çalışır? 117
`merge`, `zip` ve `combine` 117
`fold` ve `scan` 117
`flatMapConcat`, `flatMapMerge` ve `flatMapLatest` 118
Değişmeyene Kadar Ayırt Etme (distinctUntilChanged) 118
`buffer` ve `conflate` 118
`debounce` 118

`sample` . . . 118
Terminal işlemler (Terminal operations) . . . 118
Özet . . . 118
Alıştırma: ProductService . . . 119
Alıştırma: Flow Kata . . . 119
Alıştırma: MessageService . . . 119

SharedFlow ve StateFlow . . . **120**
SharedFlow . . . 120
`shareIn` . . . 120
StateFlow . . . 120
`stateIn` . . . 120
Özet . . . 120
Alıştırma: ProductService'i Güncelleyin . . . 120
Alıştırma: TemperatureService'i Güncelleyin . . . 121
Alıştırma: LocationService . . . 121
Alıştırma: PriceService . . . 121
Alıştırma: stateIn kullanarak NewsViewModel . . . 121

Flow Yapılarının Test Edilmesi . . . **122**
Dönüştürücü (Transformation) Fonksiyonlar . . . 122
Sonsuz Flow'ların Test Edilmesi . . . 122
Kaç bağlantının açıldığını belirlemek . . . 122
ViewModel'ların Test Edilmesi . . . 122
Özet . . . 122
Alıştırma: Flow Testi . . . 122

Bölüm 4: Uygulamada Kotlin Coroutines . 124

Yaygın Kullanım Senaryoları . . . **125**
Veri/Adaptör Katmanı(Data/Adapters Layer) . . . 125
Alan Katmanı(Domain Layer) . . . 125
Sunum/API/UI katmanı (Presentation/API/UI layer) . . . 126
Özet . . . 126

Diğer Dillerden Coroutine Kullanımı . . . **127**
Farklı Platformlarda Thread Kullanımı . . . 127
Askıya Alınan(suspending) Fonksiyonları Askıya Alınmayan(non-suspending) Fonksiyonlara Dönüştürmek . . . 127

Diğer Dillerden Askıya Alınan (Suspending) Fonksiyonları Çağırmak . . 127
Flow ve Tepkisel-Reaktif Akışlar (Reactive Streams) 128
Özet 128

Coroutine'lerin Başlatılması ile Askıya Alınan Fonksiyonların Karşılaştırılması 129

En İyi Uygulamalar 130

Sonuç 133

Alıştırma çözümleri 134

Giriş

Neden Kotlin Coroutines öğrenmek istiyorsunuz? Çoğu zaman kurs katılımcılarıma bu soruyu sorarım. "Havalı olduğu için" ya da "Herkes ondan bahsettiği için" yaygın cevaplar arasında yer alır. Daha derinlemesine incelediğimde ise "Daha hafif iş parçacıkları (threads) oldukları için", "RxJava'dan daha kolay oldukları için" ya da "Eşzamanlılığı (concurrency[1]) sürdürürken imperatif bir kod yazımını mümkün kılıyorlar" gibi yanıtlar alırım. Ancak, coroutines (coroutines[2]) bunlardan çok daha fazlasıdır. Eşzamanlılığın kutsal kasesidir. Bir kavram olarak, bilgisayar bilimlerinde 1960'lı yıllardan beri biliniyorlar, ancak ana akım programlamada yalnızca sınırlı bir biçimde (async/await gibi) kullanılmıştır. Bu durum, çok daha genel amaçlı coroutines'leri (coroutines) tanıtan Golang ile değişti. Kotlin ise bu fikrin üzerine inşa ederek, şu anda en güçlü ve pratik uygulamalardan biri olduğunu düşündüğüm bir yapı oluşturdu.

Eşzamanlılık (concurrency) günümüzde yazılım geliştirme süreçlerinde giderek daha büyük bir önem kazanıyor. Ancak, geleneksel yöntemler artan performans ve verimlilik gereksinimlerini karşılamakta yetersiz kalıyor. Bu nedenle, daha modern ve etkili eşzamanlılık çözümlerine olan ihtiyaç her geçen gün artıyor.. Mevcut eğilimler, sektörümüzün açıkça coroutines (coroutines) yönüne doğru ilerlediğini gösteriyor ve Kotlin Coroutines bu alanda önemli bir adımı temsil ediyor. Size coroutines'i (coroutines) tanıtmak istiyorum ve yaygın

[1]Concurrency kelimesi Latince kökenlidir. "Con-" öneki "birlikte" anlamına gelir, "currere" ise "koşmak" anlamındadır. Bu iki öge birleşerek "concurrere" fiilini oluşturur, yani "birlikte koşmak" anlamına gelir. Zamanla bu kelime, eşzamanlı olayların gerçekleşmesini ifade eden concurrency kavramına evrilmiştir. Programlama açısından concurrency, bir programın aynı anda birden fazla görevi yönetebilme yeteneğini ifade eder. Görevler sıralı olarak çalışabilir ancak birden fazla görev başlatılabilir ve paylaşılabilir kaynakları kullanabilir.

[2]Coroutines kelimesi "Co-" (birlikte) ve "routine" (düzenli iş) sözcüklerinden türemiştir. Coroutines, program akışını durdurup daha sonra kaldığı yerden devam ettirme yeteneği sunar ve async/await yapılarına benzer. Concurrency genel anlamda birden fazla görevi aynı anda yürütme kapasitesini ifade ederken, coroutines daha hafif iş parçacıkları ile görevlerin askıya alınıp devam ettirilmesine olanak tanır. Concurrency, çoklu iş parçacıklarıyla gerçek zamanlı paralel işlemlerle ilgilenirken, coroutines bu işlemleri daha verimli şekilde yönetmek için kullanılır.

kullanım durumlarının nasıl ele alındığına dair örnekler sunacağım. Umarım bu kitabı okurken çok eğlenirsiniz.

Kitap kime hitap ediyor?

Hem backend hem de Android'de deneyimli bir geliştirici olarak, bu kitapta öncelikli olarak bu iki perspektife odaklanıyorum. Şu anda Kotlin'in iki ana endüstri uygulaması bunlar ve coroutines'in büyük ölçüde bu kullanım durumlarına uygun olarak tasarlandığı açıkça görülüyor[3]. Bu nedenle, bu kitabın esas olarak Android ve backend geliştiriciler için tasarlandığını söyleyebilirsiniz, ancak Kotlin kullanan diğer geliştiriciler için de oldukça faydalı olacaktır.

Bu kitapda , okuyucuların Kotlin'i iyi bildiği varsayılarak konular anlatılmıştır. Eğer bilmiyorsanız, Kotlin Essentials adlı diğer kitabıma başlamanızı tavsiye ederim.

Kitabın bölümleri

Kitap şu bölümlere ayrılmıştır:

- **Bölüm 1: Kotlin Coroutines'i Anlamak** – Kotlin Coroutines'in ne olduğunu ve gerçekte nasıl çalıştığını açıklamaya adanmıştır.
- **Bölüm 2: Kotlin Coroutines Kütüphanesi** – kotlinx.coroutines kütüphanesindeki en önemli kavramları ve bunları nasıl iyi kullanacağınızı açıklar.
- **Bölüm 3: Kanal (Channel) ve Akış (Flow)** – kotlinx.coroutines kütüphanesindeki Kanal ve Akış'a odaklanır.
- **Bölüm 4: Kotlin Coroutines'in Uygulamada Kullanımı** – Kotlin Coroutines'in yaygın kullanım durumlarını ve en iyi uygulamalarını gözden geçirir.

[3]Google'ın Android ekibi, bu kitapta sunacağımız bazı özelliklerin tasarımında ve oluşturulmasında iş birliği yaptı.

Konuların kapsamı

Bu kitap, yürüttüğüm bir kursa dayanmaktadır. Kursun yapıldığı farklı zamanlarda farklı katılımcıları neyin ilgilendirdiğini ve neyin ilgilendirmediğini gözlemleme fırsatım oldu. Katılımcılar tarafından en sık belirtilen unsurlar şunlardır:

- **Coroutines gerçekten nasıl çalışır?** (Bölüm 1)
- **Coroutines pratikte nasıl kullanılır?** (Bölüm 2, 3 ve özellikle 4)
- **En iyi uygulamalar nelerdir?** (Bölüm 2, 3 ve özellikle 4)
- **Kotlin coroutines'in test edilmesi** (*Kotlin Coroutines'in Testi* Bölüm 2'de ve *Flow'un Testi* Bölüm 3'te)
- **Flow nedir ve nasıl çalışır?** (Bölüm 3)

Geliştiriciler İçin Kotlin Serisi

Bu kitap, *Geliştiriciler İçin Kotlin* adlı kitap serisinin bir parçasıdır ve şu kitapları içerir:

- *Kotlin Essentials* - Kotlin tüm temel özelliklerini kapsar.
- *Functional Kotlin* - Fonksiyon tipleri, lambda ifadeleri, koleksiyon işleme, DSL(Domain-Specific Language)'ler ve kapsam fonksiyonları gibi fonksiyonel Kotlin özelliklerine odaklanmıştır.
- *Kotlin Coroutines: Derinlemesine İnceleme* - Kotlin Coroutines özelliklerinin tamamını, bunların nasıl kullanılacağını ve test edileceğini, flow kullanımı, en iyi uygulamalar ve en yaygın hataları kapsar.
- *Advanced Kotlin* - Jenerik varyans değiştiricileri, delege etme, çoklu platform programlama, açıklama işleme, KSP ve derleyici eklentileri gibi gelişmiş Kotlin özelliklerine odaklanmıştır.
- *Effective Kotlin: En İyi Uygulamalar* - Kotlin programlamanın en iyi uygulamalarına odaklanmıştır.

Endişelenmeyin, bu kitabı anlamak için serinin önceki kitaplarını okumanız gerekmiyor. Ancak, Kotlin hakkında daha fazla bilgi edinmek istiyorsanız, serideki diğer kitapları göz önünde bulundurmanızı öneririm.

Kurallar ve Kılavuzlar

Koddan somut bir öğe kullandığımda, kod yazı tipini kullanacağım. Bir kavramı adlandırırken, kelimeyi büyük harfle başlatacağım. Rastgele bir türün öğesine başvururken küçük harf kullanacağım. Örneğin:

- `Flow` bir tür veya arayüzdür, kod yazı tipiyle yazılır (örneğin, "Fonksiyonun `Flow` döndürmesi gerekir");
- Flow bir kavramı temsil eder, bu yüzden büyük harfle başlar (örneğin, "Bu, Kanal ve Flow arasındaki temel farkı açıklar");
- flow bir örnektir, bir liste veya küme gibi, bu yüzden küçük harflerle yazılır (örneğin, "Her flow birkaç öğeden oluşur").

Başka bir örnek: `List`, somut olarak bir liste arayüzüne veya türüne atıfta bulunur ("`l` türü `List`'tir"), List bir kavramı temsil eder ("Bu, List ve Set arasındaki temel farkı açıklar") ve list birçok listeden biridir ("`list` değişkeni bir liste tutar").

Kitapta Amerikan İngilizcesi kullandım, ancak "cancellation/cancelled" yazımında, "Cancelled" coroutine durumu yazımına bağlı kaldığım için İngiliz yazımını tercih ettim.[4]

Kod Kuralları

Sunulan kod parçalarının çoğu çalıştırılabilir durumdadır, bu yüzden bunları bir Kotlin dosyasına kopyalayıp yapıştırırsanız çalıştırabilmelisiniz. Tüm kod parçalarının kaynak kodu aşağıdaki depoda yayımlanmıştır:

https://github.com/MarcinMoskala/coroutines_sources

Kod parçalarının sonuçları `println` fonksiyonu kullanılarak sunulur. Sonuçlar genellikle bu kod parçalarının sonunda yorum satırları olarak yer alır. Çıktılar arasında bir gecikme varsa, bu gecikme parantez içinde gösterilecektir. İşte bir örnek:

[4]Yazarın üslubu ve anlatım dili mümkün olduğunca korunmaya çalışılmakla birlikte gerekli yerlerde doğrudan çeviri yerine mana açısından ifade edilmek istenen dile aktarılmıştır. Programlamada kullanılan kavramların daha iyi anlaşılabilmesi adına bazı kavramların etimolojik olarak anlamı irdelenmiş ve lafz ile mana arasındaki semantik ilişkinin okuyucu nezdinde daha kolay anlaşılması hedeflenmiştir.

```
suspend fun main(): Unit = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
// Hello,
// (1 sec)
// World!
```

Bazı durumlarda, çıktıyı yazdıran satırın yanına yorumları ekleyeceğim. Bunu, sıranın net olduğu durumlarda yaparım:

```
suspend fun main() {
    println("Hello,") // Hello,
    delay(1000L) // (1 sec)
    println("World!") // World!
}
```

Bazen, kodun veya sonuçların bazı kısımları yorum satırında `...` ile kısaltılır. Bu gibi durumlarda, bunu "burada daha fazlası olmalıydı, ancak yazar bunu atlamaya karar verdi" şeklinde okuyabilirsiniz.

```
launch(CoroutineName("Name1")) { /*...*/ }
launch(CoroutineName("Name2") + Job()) { /*...*/ }
```

Bazı kod parçalarına, davranışını daha kolay açıklamak için satır sonuna bir numara ekledim. Bu, şu şekilde görünebilir:

```
suspend fun main() {
    println("Hello,") // 1
    delay(1000L) // 2
    println("World!") // 3
}
```

Yukarıda 1. satırda "Hello," yazdırılır, ardından 2. satırda `delay` olduğu için bir saniye bekleriz ve 3. satırda " World!" yazdırılır.

Bazı kod parçalarında tuhaf biçimlendirme fark edebilirsiniz. Bunun nedeni, bu kitabın satır uzunluğunun sadece 67 karakter olmasıdır, bu yüzden sayfa genişliğine uyacak şekilde biçimlendirmeyi ayarladım.

Sürüm

Bu kitabın 3.1 sürümüdür. Kotlin 2.0.20, kotlinx.coroutines 1.8.0 ve kotlinx.coroutines-test 1.8.0 sürümlerine dayanmaktadır. Son güncelleme Temmuz 2024'te yapılmıştır.

Alıştırmalar ve çözümler

Çoğu bölümün sonunda alıştırmalar bulacaksınız. Bu alıştırmalar, materyali daha iyi anlamanıza yardımcı olmak için tasarlanmıştır. Bu alıştırmaların başlangıç kodu ve birim testleri, GitHub'daki MarcinMoskala/kotlin-exercises projesinde bulunabilir. Bu projeyi klonlayıp alıştırmaları yerel olarak çözebilirsiniz. Çözümler kitabın sonunda yer almaktadır.

Öneriler

Bu kitapla ilgili önerileriniz veya düzeltmeleriniz varsa, bunları contact@kt.academy adresine gönderebilirsiniz.

Teşekkürler

Bu kitap, gözden geçirenlerin değerli öneri ve yorumları olmadan bu kadar iyi olamazdı. Hepsine teşekkür etmek istiyorum. En aktif olanlardan başlayarak gözden geçirenlerin tam listesi aşağıdadır.

Nicola Corti -Google Developer Expert unvanına sahip bir Kotlin geliştiricisidir. 1.0 sürümünden önce bile bu dili kullanmaya başlamış ve mobil geliştiriciler için Detekt, Chucker, AppIntro gibi birçok açık kaynaklı kütüphanenin bakımını üstlenmiştir. Şu anda Meta bünyesinde, en popüler çapraz platform mobil framework'lerinden biri olan React Native'in çekirdek

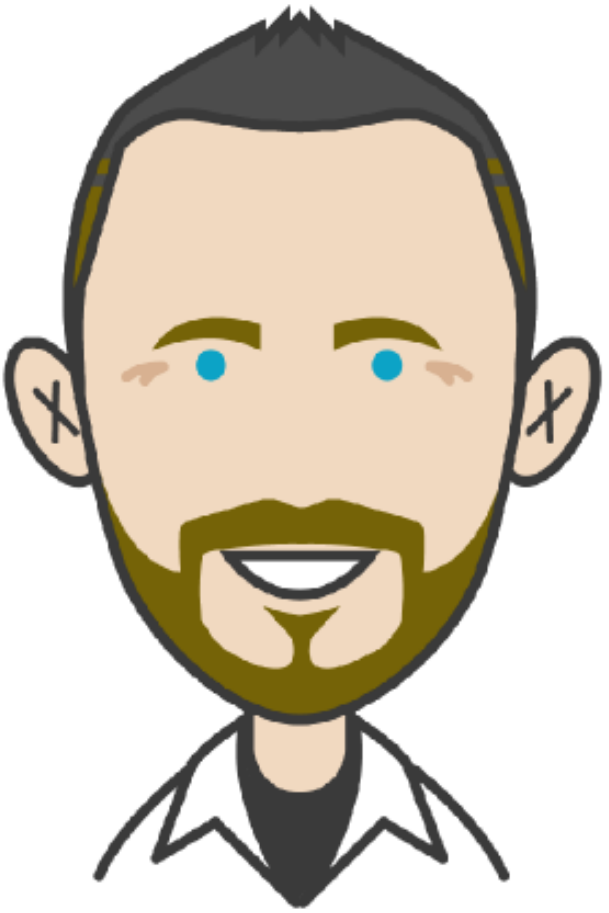
ekibinde çalışmaktadır. Ayrıca, geliştirici topluluğunun aktif bir üyesidir. Uluslararası konferanslarda konuşmalar yapmak, CFP komitelerine katılmak ve Avrupa genelinde geliştirici topluluklarını desteklemek gibi birçok etkinlikte yer almaktadır. Boş zamanlarında pasta yapmayı, podcast yayınlamayı ve koşmayı sever.

Garima Jain-Garima Jain – Hindistan'dan bir Android Google Developer Expert olup, toplulukta @ragdroid olarak da bilinir. GoDaddy'de Baş Android Mühendisi olarak çalışmaktadır. Aynı zamanda uluslararası bir konuşmacı ve aktif bir teknik blog yazarıdır. Topluluk içinde diğer insanlarla etkileşimde bulunmaktan ve düşüncelerini paylaşmaktan keyif alır. Boş zamanlarında televizyon programları izlemeyi, masa tenisi ve basketbol

oynamayı sever. Kurgu ve kodlamaya olan tutkusu sayesinde teknolojiyi kurgu ile harmanlamaktan ve fikirlerini konuşmalar ile blog yazıları aracılığıyla paylaşmaktan büyük zevk alır.

Ilmir Usmanov -2017'den beri Kotlin derleyicisinde coroutine desteği üzerinde çalışan bir JetBrains yazılım geliştiricisidir. Coroutine tasarımının stabilizasyonu ve uygulanmasından sorumlu olmuştur. Daha sonra inline sınıflar gibi diğer özellikler üzerinde çalışmaya başlamıştır. Şu anda coroutines ile ilgili çalışmaları hata düzeltme ve optimizasyon ile sınırlıdır, çünkü coroutines artık tamamlanmış ve stabil bir dil özelliği haline geldiğinden, ekstra zaman harcamaya gerek duyulmamaktadır.

Sean McQuillan - Google'da Geliştirici Temsilcisi . Twilio ve diğer San Francisco girişimlerinde on yıllık deneyime sahip olan Sean, ölçeklenebilir uygulamalar oluşturma konusunda uzmandır. Yüksek kaliteli uygulamaları hızlı bir şekilde oluşturmak için harika araçlar kullanmaya tutku duyar. Android üzerinde çalışmadığı zamanlarda onu piyano çalarken veya şapka örerken bulabilirsiniz.



Igor Wojda - On yılı aşkın yazılım geliştirme deneyimine sahip tutkulu bir mühendis. Android uygulama mimarisi ve Kotlin diliyle derinlemesine ilgileniyor ve açık kaynak topluluğunun aktif bir üyesi. Igor, konferans konuşmacısı, ' Kotlin In Action' kitabının teknik düzeltmeni ve 'Android Development with Kotlin' kitabının yazarıdır. Kodlama tutkusunu diğer geliştiricilerle paylaşmaktan keyif alır.

Jana Jarolimova - Avast'ta bir Android geliştiricisidir. Kariyerine Prag Şehir Üniversitesi'nde Java dersleri vererek başladı, ardından mobil geliştirme alanına geçti ve bu kaçınılmaz olarak onu Kotlin'e ve bu dile olan sevgisine yönlendirdi.

Richard Schielek - Deneyimli bir geliştirici ve Kotlin ile coroutines'i daha stabil hale gelmeden önce üretimde kullanan erken dönem

benimseyicilerdendir. Avrupa uzay endüstrisinde birkaç yıl çalışmıştır.

Vsevolod Tolstopyatov - Kotlin Libraries ekibinin lideridir. JetBrains'de çalışıyor ve API tasarımı, eşzamanlılık (concurrency), JVM iç yapıları, performans ayarlamaları ve metodolojilerle ilgileniyor.

Lukas Lechner, **İbrahim Yılmaz**, **Dean Djermanović** ve **Dan O'Neill**.

Ayrıca, kitabın tamamında yaptığı mükemmel düzeltmeler için dil redaktörümüz **Michael Timberlake**'e de teşekkür etmek istiyorum.

Bölüm 1: Kotlin Coroutines'i Anlamak

Kotlin Coroutines kütüphanesine başlamadan önce, bazı temel kavramlara göz atmak faydalı olacaktır. Coroutines nedir? Askıya alma (suspension) mekanizması nasıl çalışır? Bu süreçlerin perde arkasında neler olup bitiyor? Tüm bu soruları yanıtlayarak çeşitli araçlar ve uygulamalar vasıtasıyla nasıl daha verimli çalışabileceğimizi keşfedeceğiz.

Neden Kotlin Coroutines?

Birçok geliştirici coroutineleri hafif iş parçacıkları (lightweight thread) olarak değerlendirir, ancak coroutineler bundan çok daha fazlasıdır. 1963 yılında ilk kez tanımlanan bir kavramın[1] endüstriyel ölçekte uygulanabilir bir versiyonunun geliştirilmesi uzun yıllar almıştır[2].

Kotlin Coroutines kütüphanesi, yarım yüzyıllık akademik çalışmalarda sunulan güçlü özellikleri, gerçek dünya senaryolarına güvenli ve basit bir şekilde entegre edilebilecek bir biçimde sunmaktadır. Dahası, Kotlin Coroutines çoklu platform (multiplatform) desteğine sahiptir. Bu, JVM, JS, iOS gibi tüm Kotlin platformlarında ve ortak (common) modüllerde kullanılabileceği anlamına gelmektedir.

Bununla birlikte, coroutineler oldukça verimlidir: RxJava veya Reactor gibi kütüphanelerden belirgin şekilde daha verimli olup, klasik engelleme tabanlı (blocking) iş parçacıklarına kıyasla karşılaştırılamayacak kadar yüksek performanslıdır. Ayrıca, kod yapısını kökten değiştirmezler. Kotlin Coroutines'in sunduğu yeteneklerin büyük bir kısmını neredeyse zahmetsiz bir şekilde kullanabiliriz (ki aynı şeyi RxJava veya geri çağrı (callback) mekanizmaları için söylemek mümkün değildir). Bu özellikleri sayesinde Kotlin Coroutines, yeni başlayanlar için de oldukça erişilebilir bir teknoloji sunmaktadır[3].

Bu bölümde, Kotlin Coroutines'in temel avantajlarını genel bir bakış açısıyla

[1]Conway, Melvin E. (Temmuz 1963). "Design of a Separable Transition-diagram Compiler" (Ayrılabilir Geçiş Diyagramı Derleyicisinin Tasarımı). Communications of the ACM. ACM. 6 (7): 396–408. doi:10.1145/366663.366704. ISSN 0001-0782. S2CID 10559786.

[2]Endüstri standardına uygun ve evrensel ilk coroutine'lerin 2009 yılında Go programlama dili tarafından tanıtıldığını düşünüyorum. Ancak, daha önceki bazı dillerde de coroutine kavramının uygulandığını belirtmek gerekir. Örneğin, Lisp gibi bazı eski dillerde coroutine kullanımı vardı; ancak bu kullanım yaygınlık kazanmadı. Bunun temel nedeninin, bu dillerdeki coroutine uygulamalarının gerçek dünya senaryolarını destekleyecek şekilde tasarlanmamış olması olduğunu düşünüyorum. Lisp (tıpkı Haskell gibi), çoğunlukla bilim insanları için bir deney alanı olarak kullanılmış, profesyonel yazılım geliştirme amacıyla geniş ölçekte benimsenmemiştir.

[3]Bu durum, coroutine kavramını doğru bir şekilde anlayıp etkin bir şekilde kullanmanın önemini değiştirmemektedir.

ele alacak ve daha sonraki bölümlerde bu avantajları detaylı bir şekilde inceleyeceğiz.

Basitlik (Simplicity)

Kotlin Coroutines, yaygın kullanım senaryolarını basitleştirmek için tasarlanmıştır. Bize, işlem sırasında ana iş parçacığını (main thread) bloklamadan emredici (imperative) bir stil kullanma veya daha az kod tekrarı (boilerplate) ile reaktif bir yaklaşım benimseme imkanı sunar.

Bu özellik özellikle Android ve JavaScript gibi ön yüz (frontend) platformlarında büyük avantaj sağlar. Uygulamalarımızın büyük bir kısmı ana iş parçacığında (main thread) çalışır ve burada engelleyici işlemler (blocking operations) mümkün değildir. Ancak coroutine'lerin askıya alınması (suspending) herhangi bir sorun oluşturmaz.

Kotlin Coroutines, kodun en doğal ve sezgisel şekilde yazılmasına, kolay hata ayıklamaya (debugging) ve bakımı daha basit hale getirmeye yardımcı olur. Callback mekanizmalarına veya reaktif akışlara (reactive streams) kıyasla çok daha kolay uygulanabilir ve yönetilebilir bir yapı sunar.

Aşağıdaki örnekte, coroutine'lerin nasıl basit ve okunaklı bir şekilde kullanıldığını görebiliriz:

```
fun onCreate() {
    viewModelScope.launch {
        val user = fetchUser() // coroutine askıya alınır (suspend)
        displayUser(user) // ana iş parçacığında (main thread) çalışır
        val posts = fetchPosts(user) // coroutine tekrar askıya alınır
        displayPosts(posts) // ana iş parçacığında çalışır
    }
}
```

Bu kullanım kolaylığı, birden fazla asenkron işlemi gerçekleştirmemiz gerektiğinde özellikle belirgin hale gelir. Asenkron işlemler, Kotlin Coroutines kütüphanesinin bir parçası olan `async` ve `await` fonksiyonlarıyla kolayca yönetilebilir.

```
suspend fun fetchUser(): UserData = coroutineScope {
    val userDetails = async { api.fetchUserDetails() }
    val posts = async { api.fetchPosts() }
    UserData(userDetails.await(), posts.await())
}
```

Asenkron işlemler birbirleriyle kolayca birleştirilebilir. Örneğin, bir koleksiyondaki her öğe için asenkron bir işlem başlatabilir ve tüm işlemlerin tamamlanmasını bekleyebiliriz.

```
suspend fun getUserArticleDetails(
    userId: String
): List<ArticleDetails> = coroutineScope {
    articleRepo.getArticles(userId)
        .filter { it.isPublic }
        .map { async { articleRepo.getArticleDetails(it.id) } }
        .awaitAll()
}
```

Coroutines ayrıca, diğer kütüphanelerde bulunmayan eşzamanlama (synchronization) araçları sunar. Bunun yanı sıra, yönetimi deneyimli bir geliştirici için basit ve sezgiseldir.

Performans

Kotlin Coroutines, yüksek verimlilik sağlamak için tasarlanmıştır. Hafif yapıları sayesinde büyük ölçekli sistemlerde geniş çapta kullanılabilirler. Askıya alınan (suspend) fonksiyonlar `Single`[4]'dan çok daha hafif, `Flow`[5] ise, `Observable`[6]'dan çok daha verimlidir. Askıya alınan fonksiyonların ve `Flow` yapısının nasıl çalıştığını incelediğimizde, mümkün olan en basit ve en verimli yapıda olduklarını görebiliriz.

Coroutines, iş parçacıklarına (threads) kıyasla çok daha verimlidir çünkü iş parçacıkları yüksek bellek tüketimi gerektirir ve başlatılması ile durdurulması

[4]Single: Tek bir değer üreten ve tamamlanan bir veri akışıdır. API çağrıları gibi tek seferlik işlemler için kullanılır.

[5]Flow: Kotlin Coroutines içinde, reaktif veri akışlarını temsil eden bir yapıdır. Hafif ve geri basıncı (backpressure) yönetir.Flow, `Cold Flow` ve`Hot Flow` olmak üzere iki temel türü vardır. `Cold Flow`, her gözlemci için yeni bir akış başlatırken, `Hot Flow` (`StateFlow` ve `SharedFlow`) birden fazla gözlemciyle durum paylaşabilir. Android'de `StateFlow`, UI durum yönetimi için yaygın olarak kullanılır.

[6]Observable: RxJava'da çok sayıda veri yayabilen bir veri akışıdır. UI olayları ve sürekli veri akışlarında kullanılır.

maliyetlidir. Coroutines, iş parçacıkları üzerinde çalışan hafif bir soyutlama katmanıdır. İş parçacıklarını en verimli şekilde kullanarak büyük ölçekli uygulamalarda kolayca yönetilebilirler.

Bu durumu, 100.000 kullanıcının bir arka uç (backend) servisine veri talebi göndermesini simüle eden aşağıdaki kod parçacıklarıyla açıklayabiliriz. İlk kod parçası, 100.000 iş parçacığını başlatarak bunları bir saniye bekletmektedir (örneğin, bir veritabanı yanıtını beklemek gibi). Bu kod çalıştırıldığında, noktaların ekrana yazdırılmasının zaman aldığı ve hatta sistemin `OutOfMemoryError` hatası verebileceği gözlemlenebilir. Bu, büyük ölçekli iş parçacığı kullanımının getirdiği maliyettir.

İkinci kod parçası ise, iş parçacıkları yerine coroutines kullanıyor ve işlemleri askıya alarak bekletiyor . Bu kod çalıştırıldığında, program bir saniye bekleyip ardından tüm noktaları ekrana yazdırmaktadır . Tüm bu coroutine işlemlerini başlatmanın maliyeti o kadar düşüktür ki, neredeyse fark edilmez.

```
import kotlin.concurrent.thread

fun main() {
    repeat(100_000) {
        thread {
            Thread.sleep(1000L)
            print(".")
        }
    }
}
```

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

Kotlin Coroutines her halükarda Java 21 ile sunulan **Project Loom**[7]'un sanal iş parçacıklarından daha hızlı değildir, çünkü bu sanal iş parçacıkları da

[7]Observable: RxJava'da çok sayıda veri yayabilen bir veri akışıdır. UI olayları ve sürekli veri akışlarında kullanılır.

aslında kendi coroutine yapılarıyla çalışmaktadır. Ancak Kotlin Coroutines, iptal mekanizmaları (cancellation) ve eşzamanlama (synchronization) gibi konularda ek verimlilik sağlayan birçok gelişmiş araç sunmaktadır.

İptal Mekanizması (Cancellation)

Kotlin Coroutines, zahmetsiz bir iptal mekanizması sunar. Bu durum, özellikle ön yüz (frontend) geliştirme bağlamında büyük önem taşır, çünkü kullanıcı ekranı terk ettiğinde veya bir arayüz öğesi gizlendiğinde tüm işlemlerin otomatik olarak iptal edilmesi gerekir. Coroutine'lerin iptal edilmesi , geri çağırma (callback) tabanlı API'ler veya reaktif akışlar (reactive streams) ile kıyaslandığında oldukça basittir.

```
fun onCreate() {
    viewModelScope.launch {
        // Kullanıcı ekranı terk ettiğinde iptal edilir
        val news = getNewsFromApi()
        val sortedNews = news
            .sortedByDescending { it.publishedAt }
        view.showNews(sortedNews)
    }
}
```

Arka uç (backend) geliştirme bağlamında, iptal mekanizmasının avantajları daha az görünür olsa da, yine de büyük önem taşır. Örneğin, önceki bölümde ele alınan `fetchUser` fonksiyonunu düşünelim. Eğer `fetchPosts` başarısız olursa, `fetchUserDetails` fonksiyonu hemen iptal edilir ve böylece sistem kaynakları serbest bırakılır. Bu, yapısal eşzamanlılık (structured concurrency) ilkesinin bir sonucu olup Kotlin Coroutines tarafından tanıtılmıştır.

```
suspend fun fetchUser(): UserData = coroutineScope {
    // Eğer fetchPosts başarısız olursa, fetchUserDetails iptal edilir
    val userDetails = async { api.fetchUserDetails() }
    // Eğer fetchUserDetails başarısız olursa, fetchPosts iptal edilir
    val posts = async { api.fetchPosts() }
    UserData(userDetails.await(), posts.await())
}
```

İptal mekanizması, çoğu zaman doğrudan görünmese de, birçok farklı durumda büyük avantajlar sağlar. Örneğin, Ktor sunucusu (coroutine

tabanlı bir arka uç çerçevesi) üzerinde çalışan bir istemciyi ele alalım. Eğer bir istemci sunucuya bir istek gönderip yanıtı beklemeden bağlantıyı kapatırsa, coroutine iptal mekanizması sayesinde sunucu, isteği anında iptal ederek sistem kaynaklarını gereksiz yere tüketmekten kaçınabilir. Ancak, engelleyici (blocking) iş parçacıkları kullanıldığında, sunucu genellikle isteğin tamamlanmasını beklemek zorunda kalır ve kaynaklar serbest bırakılamaz. Bu avantaj, WebSockets veya RSocket gibi bağlantıların dinamik olarak kesilebildiği iletişim protokolleri kullanan sunucular için özellikle kritiktir. Kullanıcı bağlantıyı herhangi bir anda kesebileceğinden, coroutine'lerin sunduğu otomatik iptal mekanizması, yüksek trafiğe sahip servislerde büyük bir verimlilik avantajı sağlar.

```
// Ktor sunucu örneği
fun Route.messagesApi() {
    get("/message/statistics") {
        // HTTP bağlantısı kesildiğinde otomatik olarak iptal edilir
        val statistics = calculateMessageStatistics()
        call.respond(statistics)
    }
    rSocket("/message/channel") {
        RSocketRequestHandler {
            requestChannel { header, control ->
                // RSocket bağlantısı kesildiğinde otomatik olarak iptal edilir
                messagesFlow(header, control)
            }
        }
    }
}
```

Eşzamanlama (Synchronization)

Kotlin Coroutines, eşzamanlı işlemleri senkronize etmek için çeşitli araçlar sunar. Değiştirilebilir (mutable) bir duruma erişimi senkronize etmek ve çakışmaları önlemek istediğimizde, tek bir iş parçacığına (thread) sınırlı bir dispatcher[8] kullanabiliriz. Çoğu durumda, bu yaklaşım diğer senkronizasyon araçlarına kıyasla daha basit ve daha averimlidir.

[8]Dispatcher , coroutine'lerin hangi iş parçacığında (thread) çalışacağını belirleyen bir mekanizmadır. Farklı dispatcher türleri farklı kullanım senaryolarına uygundur:

```
class UserDownloader(
    private val networkService: NetworkService
) {
    private val users = mutableListOf<User>()
    private val dispatcher = Dispatchers.IO.limitedParallelism(1)

    suspend fun downloaded(): List<User> = withContext(dispatcher) {
        users.toList()
    }

    suspend fun fetchUser(id: Int) = withContext(dispatcher) {
        val newUser = networkService.fetchUser(id)
        users += newUser
    }
}
```

Eğer birden fazla asenkron işlemi başlatıp tamamlanmasını beklemek istiyorsak, `supervisorScope` gibi bir coroutine scope fonksiyonunu kullanabiliriz. Yapısal eşzamanlılık (structured concurrency) sayesinde, alt coroutine'lerin tamamlanmasını beklemek bu yapıya doğal olarak dahildir.

```
suspend fun process() {
    val updates = fetchUpdates()
    supervisorScope {
        updates.forEach { update ->
            launch {
                processUpdate(update)
            }
        }
    }
    println("Tüm güncellemeler işlendi")
    sendUpdatesProcessedNotification()
}
```

Bunun yanı sıra, diğer senkronizasyon ihtiyaçları için Kotlin Coroutines içinde yerleşik birçok mekanizma bulunmaktadır:

- Bir coroutine'in başka bir coroutine'i beklemesi gerekiyorsa, `Job` nesnesi üzerinden `join` fonksiyonu kullanılabilir.
- Bir coroutine tarafından üretilen değerin başka bir coroutine tarafından beklenmesi gerekiyorsa, `CompletableDeferred` kullanılabilir.
- Coroutine'lerin güvenli bir şekilde veri alışverişi yapması gerekiyorsa, `Channel` kullanılabilir.

Bu araçların tamamı yerleşik (built-in) olarak gelir ve oldukça kolay kullanılabilir. Kotlin Coroutines'in sunduğu bu senkronizasyon araçları sayesinde, eşzamanlı işlemler daha yönetilebilir ve verimli hale gelmektedir.

```
class SomeService(
    private val scope: CoroutineScope
) {
    fun startTasks() {
        val job = scope.launch {
            // ...
        }
        scope.launch {
            // ...
            job.join()
            // ...
        }
    }
}
```

Test Edilebilirlik (Testability)

Coroutines'in en büyük avantajlarından biri de test edilebilirliğidir. Kotlin Coroutines, sanal zaman (virtual time)[9] kullanarak çalışmayı destekler. Bu özellik sayesinde, diğer kütüphanelerle test edilmesi zor olan senaryolar için hızlı, kesin ve belirleyici testler yazmak mümkündür. Örneğin, `fetchUser` fonksiyonunun gerçekten kullanıcı bilgilerini ve gönderileri asenkron olarak çektiğini test etmek istediğinizi varsayalım. Her işlemin bir saniye sürdüğünü simüle eden bir test yazarak, fonksiyonun toplamda tam olarak bir saniye sürdüğünü doğrulayabilirsiniz. Bu tür bir test, sanal zaman sayesinde hızlı, kesin ve öngörülebilir olur.

[9] Sanal zaman (virtual time), coroutine tabanlı testlerde zamanla ilgili işlemleri hızlandırmak için kullanılan bir mekanizmadır. `runTest` fonksiyonu, testler sırasında coroutine gecikmelerini (`delay`) hızlandırarak gerçek zamanlı bekleme ihtiyacını ortadan kaldırır. Bu, testlerin hızlı ve belirleyici olmasını sağlar.

```
@Test
fun `should fetch data asynchronously`() = runTest {
        val api = mockk<Api> {
            coEvery { fetchUserDetails() } coAnswers {
                delay(1000)
                UserDetails("John Doe")
            }
            coEvery { fetchPosts() } coAnswers {
                delay(1000)
                listOf(Post("Hello, world!"))
            }
        }
        val useCase = FetchUserDataUseCase(api)
        val userData = useCase.fetchUser()
        assertEquals("John Doe", userData.user.name)
        assertEquals("Hello, world!", userData.posts.single().title)
        assertEquals(1000, currentTime)
    }
```

Sanal zaman, zaman aşımı (timeouts), tekrar deneme (retries) ve zamanla ilgili diğer işlemleri test etmeyi de mümkün kılar. Ayrıca, farklı senaryoların nasıl işlediğini doğrulamamıza olanak tanır. Örneğin:

- X, Y'den daha hızlı ancak Z'den daha yavaş olursa ne olur?
- X, Z'den daha hızlı olursa nasıl bir sonuç alırız?

Tüm bu senaryoları kolayca simüle edip test etmek mümkündür. Kotlin Coroutines'in sunduğu test edilebilirlik avantajlarından biri de budur.

Flow

Kotlin Coroutines, asenkron veri akışlarını ifade etmek ve işlemek için güçlü bir soyutlama sağlar. Bu soyutlama Flow[10] olarak adlandırılır. Flow, RxJava veya Reactor akışlarına kıyasla çok daha hafiftir ve uygulanması oldukça basittir (ilerleyen bölümlerde yalnızca birkaç satır kod ile nasıl uygulanabileceğini göstereceğim). Ayrıca, birçok kullanışlı operatörü içeren zengin bir özellik kümesine sahiptir.

[10] Flow: Kotlin Coroutines içinde, reaktif veri akışlarını temsil eden bir yapıdır. Hafif ve geri basıncı (backpressure) yönetir.Flow, `Cold Flow` ve`Hot Flow` olmak üzere iki temel türü vardır. `Cold Flow`, her gözlemci için yeni bir akış başlatırken, `Hot Flow` (`StateFlow` ve `SharedFlow`) birden fazla gözlemciyle durum paylaşabilir. Android'de `StateFlow`, UI durum yönetimi için yaygın olarak kullanılır.

```
fun notificationStatusFlow(): Flow<NotificationStatus> =
    notificationProvider.observeNotificationUpdate()
        .distinctUntilChanged()
        .scan(NotificationStatus()) { status, update ->
            status.applyNotification(update)
        }
        .combine(
            userStateProvider.userStateFlow()
        ) { status, user ->
            statusFactory.produce(status, user)
        }
```

Flow'lar `SharedFlow` ve `StateFlow` sayesinde sıcak akışlar (hot flows) olabilir. Bu mekanizmalar, olayların veya durum değişikliklerinin birden fazla gözlemciye yayımlanmasına olanak tanır. Böylece, yalnızca bir gözlemcinin birden fazla akış tarafından kullanılmasını sağlamak veya gözlemlenebilir bir değişken durumu temsil etmek mümkündür. Android üzerinde, değiştirilebilir durumu temsil etmek için yaygın olarak kullanılan standart yöntem budur.

```
class NotificationProvider(
    notificationClient: NotificationClient,
    scope: CoroutineScope
) {
    private val notifications = notificationClient.observe()
        .shareIn(
            scope = scope,
            started = SharingStarted.WhileSubscribed(),
        )

    fun observe(): Flow<Notification> = notifications
}
```

```
// Android'de değiştirilebilir durumu temsil etme yöntemi
class NewsViewModel : BaseViewModel() {
    private val _loading = MutableStateFlow(false)
    val loading: StateFlow<Boolean> = _loading

    private val _news = MutableStateFlow(emptyList<News>())
    val news: StateFlow<List<News>> = _news

    fun onCreate() {
        newsFlow()
            .onStart { _loading.value = true }
            .onCompletion { _loading.value = false }
            .onEach { _news.value = it }
```

```
14              .catch { _failure.value = it }
15              .launchIn(viewModelScope)
16      }
17  }
```

Coroutines Çoklu Platform Desteğine Sahiptir

Kotlin Coroutines, tüm Kotlin hedeflerinde ve ortak modüllerde kolayca kullanılabilen bir çoklu platform (multiplatform) kütüphanesidir. Bu evrensellik, çoklu platform projeleri veya kütüphaneleri yazarken büyük bir avantaj sağlar. Özellikle ön yüz (frontend) tarafında çoklu platform bileşenlerinin popülaritesi artmaktadır, ancak bu avantaj arka uç (backend) geliştirmede de etkili bir şekilde kullanılabilir. En önemlisi, birçok Kotlin kütüphanesi çoklu platform desteğine sahiptir, bu sayede aynı kütüphane hem istemci tarafında hem de sunucu tarafında kullanılabilir. Bazı projelerde, aynı kodu hem arka uçta hem de ön yüzde, hatta JavaScript üzerinde kullanabilmek için çoklu platform kütüphaneleri tanımladım. Bu gibi durumlarda coroutines oldukça faydalı olmaktadır.

Kotlin Coroutines ile İlgili En Büyük Problem

Kotlin Coroutines ile ilgili en büyük problemin, karmaşık yapıları soyutlayarak gizlemeleri olduğuna inanıyorum[11]. Eğer coroutines en iyi uygulamalara uygun bir şekilde veya detaylı bilgiyle kullanılıyorsa, bu büyük bir avantajdır. Ancak, eğer coroutine'leri nasıl çalıştığını tam olarak anlamadan sıra dışı işlemler yapmaya çalışırsanız, çeşitli problemlere yol açabilir. Örneğin, coroutine'lerin birbirini beklemesi veya yanlışlıkla iptal etmesi gibi sorunlar ortaya çıkabilir. Bunun çözümü, coroutine'lerin nasıl çalıştığını anlamaktan geçer. Doğru soyutlama seviyesini anladıktan sonra, coroutine'leri güvenli ve verimli bir şekilde kullanabilirsiniz. Eğitim burada kritik bir rol oynamaktadır ve bu kitabı yazmamın ana nedeni de budur. Amacım, coroutine'lerin nasıl çalıştığını ve nasıl güvenli ve verimli bir şekilde kullanılabileceğini anlatmaktır.

[11] Kotlin Coroutines, işlem zamanlaması, iptal mekanizmaları ve iş parçacıkları arasındaki görev geçişlerini yöneten karmaşık bir yapıya sahiptir. İç mekanizmasını anlamadan kullanıldığında, coroutine'ler arasındaki bekleme, iptal ve kaynak tüketimi gibi konularda beklenmeyen problemler yaşanabilir. Bu nedenle, coroutine'lerin çalışma prensiplerini iyi anlamak önemlidir.

Sonuç

Kotlin Coroutines, asenkron kod yazmak için güçlü bir araçtır. Basit, verimli ve test edilebilir bir yapı sunar. Senkronizasyon için birçok araç sağlar ve çoklu platform desteğine sahiptir. Hafif iş parçacıkları (lightweight threads) değillerdir, ancak birçok farklı şekilde kullanılabilen güçlü bir soyutlamadır. Yeni başlayanlar için uygun olmalarına rağmen, nasıl çalıştıklarını anlamadan kullanıldığında riskli olabilirler. Bu nedenle, kendinizi eğitmek çok önemlidir. Bu kitabın, coroutine'lerin nasıl çalıştığını anlamanıza ve onları güvenli ve verimli bir şekilde nasıl kullanacağınızı öğrenmenize yardımcı olacağını umuyorum.

- `Dispatchers.Default` : CPU yoğun işlemler için optimize edilmiştir.
- `Dispatchers.IO` : G/Ç (I/O - Input/Output) işlemleri için uygundur.
- `Dispatchers.Main`: UI işlemleri için kullanılır (Android'de yaygındır).
- `Dispatchers.Unconfined*`: İlk çalıştığında mevcut iş parçacığını kullanır ancak daha sonra farklı bir iş parçacığına geçebilir.
- `Dispatchers.IO.limitedParallelism(1)`: Tek iş parçacığıx kullanarak coroutine'lerin sıralı çalışmasını sağlar ve senkronizasyon ihtiyacını azaltır.

Sequence Builder (Dizi Üretici)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Gerçek hayattaki kullanım alanları

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Faktöriyel Dizisi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Asal Sayılar Dizisi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alma (Suspension) Mekanizması Nasıl Çalışır?

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Oyun Metaforu Üzerinden Thread ve Coroutine Karşılaştırması

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınabilen (Suspending) Fonksiyonlar

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

İlk Askıya Alma (Suspension) İşlemi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Devam Nesnesinde (`Continuation`) Neler Saklanır?

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine'i Geciktirme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bir Değer ile Devam Etme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

İstisna (Exception) ile Devam Ettirme (Resume)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya alınan Coroutine'dir , fonksiyon değil

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Geri Çağırma (Callback) Fonksiyon Sarmalayıcıları

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Devam (Continuation) Saklama

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Mekanizmasının İç Yapısı

Devam Nesnesi Geçişli Programlama (Continuation-Passing Style)

Çok Basit Bir Fonksiyon

Durum Bilgisine Sahip Bir Fonksiyon

Bir Değer ile Devam Ettirilen Fonksiyon

Çağrı Yığını (Call Stack)

Farklı Bağlamlarda Askıya Alınan Fonksiyonlar

Gerçek Kod

Askıya Alınan Fonksiyonların Performansı

Özet

Alıştırma: Bir Devam Nesnesi (Continuation) Neleri Saklar?

Coroutine'ler: Yerleşik Destek(built-in support) vs Kütüphane(library)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bölüm 2: Kotlin Coroutines Kütüphanesi

Yerleşik coroutine desteğinin nasıl çalıştığını anladığımıza göre, artık `kotlinx.coroutines` kütüphanesine odaklanmanın zamanı geldi. Bu bölümde, bu kütüphaneyi doğru şekilde kullanmak için gereken her şeyi öğreneceğiz. Coroutine oluşturucularını (coroutine builders), farklı coroutine bağlamlarını (coroutine contexts) ve iptal mekanizmasını (cancellation) detaylı olarak inceleyeceğiz. Son olarak, coroutine yapılandırma yöntemleri, test etme teknikleri ve paylaşılan bir duruma güvenli erişim gibi pratik bilgileri ele alacağız.

Eşzamanlı İş Parçacıkları(Coroutines) Nasıl Başlatılır?

Kotlin'de askıya alınabilir fonksiyonlar (* suspending functions *), kendi başlarına bir eşzamanlı iş parçacığı (*coroutine *) değildir; ancak, bir eşzamanlı iş parçacığını askıya alabilen fonksiyonlardır.Bu nedenle, yalnızca bir eşzamanlı iş parçacığı içinden çağrılabilirler.Kotlin'de bir eşzamanlı iş parçacığını başlatmanın üç temel yöntemi vardır:

-Asenkron iş parçacığı başlatıcıları (`launch` ve `async`): Belirtilen bir çalışma alanı (* scope *) içinde asenkron olarak eşzamanlı bir iş parçacığı başlatır . -Engelleyici(bloklayan) iş parçacığı başlatıcıları (`runBlocking` ve `runTest`): Mevcut işlem dizisi(*thread *) üzerinde bir eşzamanlı iş parçacığı başlatır ve tamamlanana kadar işlem dizisini durdurur . -Eşzamanlı iş parçacığı kapsam fonksiyonları(*coroutine scope functions *): Bir eşzamanlı iş parçacığı oluşturur ve mevcut eşzamanlı iş parçacığı, yeni başlatılan iş tamamlanana kadar askıya alınır.Bu yöntemlerin her biri farklı kullanım senaryolarına sahiptir . Bu bölümde, bu yöntemlerin nasıl çalıştığını ve pratikte nasıl kullanıldığını detaylı bir şekilde ele alacağız.

Asenkron Eşzamanlı İş Parçacığı Başlatıcıları

Engelleyici Eşzamanlı İş Parçacığı Başlatıcıları (Blocking Coroutine Builders)

Yapısal Eşzamanlılık (Structured Concurrency)

Ebeveyn-Çocuk (*Parent-Child*) İlişkisinin Önemli Sonuçları

Eşzamanlı İşlem Kapsamı Fonksiyonları(Coroutine scope functions)

Coroutine Kapsamı ve Coroutine Başlatıcıları Arasındaki Farklar

Özet

Alıştırma: `UserDetailsRepository`

Alıştırma: `BestStudentUseCase`

Alıştırma: CommentService

Alıştırma: mapAsync

Genel Bakış

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Bağlamı (Coroutine Context)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`CoroutineContext` Arayüzü(`interface`)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`CoroutineContext` İçinde Öğe Bulma

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bağlam (Context) Ekleme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Boş İşlem Kapsamı (Empty Coroutine Context)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Öğeleri Çıkartma (Subtracting Elements)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Kapsamı Katlama (Folding Context)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Bağlamı ve Başlatıcılar

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınabilir Bir Fonksiyonda Bağlama (Context) Erişim

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Kapsam Fonksiyonları ve Bağlam Mirası

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınabilir Fonksiyonlarda Bağlamı Değiştirme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Kendi İşlem Bağlamımızı Oluşturma

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Eşzamanlı İşlemler ve İş Parçacığı Öğeleri

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Bağlam Yayılımını Anlamak

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: CounterContext

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Yöneticileri (Dispatchers)

Kotlin'in eşzamanlı işlem kütüphanesi, bir eşzamanlı işlemin (coroutine) hangi iş parçacığında (thread) veya iş parçacığı havuzunda çalışacağını belirleme yeteneği sunar. Bu, iş parçacığı yöneticileri (dispatchers) kullanılarak gerçekleştirilir.

İngilizce sözlükte *dispatcher* kelimesi, "özellikle acil durum araçları olmak üzere, insanları veya araçları ihtiyaç duyulan yerlere göndermekten sorumlu kişi(hareket veya sevk memuru)" olarak tanımlanır. Kotlin'de ise `CoroutineContext`, belirli bir eşzamanlı işlemin hangi iş parçacığında çalışacağını belirler.

> Kotlin Coroutines'teki iş parçacığı yöneticileri (dispatchers), RxJava'daki zamanlayıcılar (schedulers) ile benzer bir kavrama sahiptir.

Varsayılan Coroutine Yöneticisi (Default Dispatcher)

Eğer bir eşzamanlı işlem için özel bir iş parçacığı yöneticisi belirtmezseniz, varsayılan olarak `Dispatchers.Default` atanır. `Dispatchers.Default`, işlemci yoğunluklu (CPU-intensive) görevleri çalıştırmak için tasarlanmıştır.

Bu yöneticinin iş parçacığı havuzu, kodun çalıştığı makinedeki işlemci çekirdeklerinin sayısına eşit olacak şekilde oluşturulur (ancak en az iki iş parçacığı olacak şekilde ayarlanmıştır). Teorik olarak, iş parçacıklarını verimli kullanmanız (örneğin, işlemci yoğunluklu hesaplamalar yapıp iş parçacıklarını engellemeden çalıştırmanız) durumunda, bu en uygun iş parçacığı sayısıdır.

Bu iş parçacığı yöneticisinin nasıl çalıştığını görmek için aşağıdaki kodu çalıştırabilirsiniz:

```
// Örnek Başlangıcı
suspend fun main() = coroutineScope {
    repeat(1000) {
        launch {
            // veya launch(Dispatchers.Default) olarak da belirtilebilir
            // İşlemciyi yoğun şekilde çalıştırmak için rastgele uzun tamsayılar üreten b\
ir liste oluşturur
            List(1_000_000) { Random.nextLong() }.maxOrNull()

            // Mevcut iş parçacığının adını alır
            val threadName = Thread.currentThread().name
            println("Çalışan iş parçacığı: $threadName")
```

Benim makinemdeki örnek çıktı (12 çekirdekli bir işlemcim var, bu yüzden havuzda 12 iş parçacığı bulunuyor):

```
Çalışan iş parçacığı: DefaultDispatcher-worker-1
Çalışan iş parçacığı: DefaultDispatcher-worker-5
Çalışan iş parçacığı: DefaultDispatcher-worker-7
Çalışan iş parçacığı: DefaultDispatcher-worker-6
Çalışan iş parçacığı: DefaultDispatcher-worker-11
Çalışan iş parçacığı: DefaultDispatcher-worker-2
Çalışan iş parçacığı: DefaultDispatcher-worker-10
Çalışan iş parçacığı: DefaultDispatcher-worker-4
...
...
```

> Uyarı: `runBlocking`, herhangi bir dağıtıcı (dispatcher) belirtilmediğinde kendi varsayılan dağıtıcısını kullanır.
> Bu nedenle, `runBlocking` kapsamı içinde çalıştırılan işlemler otomatik olarak `Dispatchers.Default` kullanmaz.
> Eğer yukarıdaki örnekte `coroutineScope` yerine `runBlocking` kullansaydık,
> tüm işlemler `"main"` iş parçacığında çalışacaktı.

Varsayılan Dağıtıcıyı (Dispatcher) Sınırlama

Diyelim ki yüksek işlem gücü gerektiren bir süreciniz var ve bu sürecin `Dispatchers.Default` tarafından kullanılan tüm iş parçacıklarını (threads) tüketerek aynı dağıtıcıyı kullanan diğer eşzamanlı işlemleri engelleyebileceğinden şüpheleniyorsunuz.

Böyle bir senaryoda, `Dispatchers.Default` üzerinde `limitedParallelism` fonksiyonunu kullanarak bir sınırlandırma getirebiliriz. Bu fonksiyon, aynı iş parçacığı havuzunu kullanmaya devam ederken, aynı anda çalışabilecek iş parçacığı sayısını belirli bir maksimum değere sınırlar. Böylece, yoğun işlem gerektiren görevlerin tüm kaynakları tüketmesi önlenerek sistemde daha dengeli bir iş yükü dağılımı sağlanır.

Bu yöntem, özellikle birden fazla eşzamanlı işlemin yürütüldüğü sistemlerde kaynak yönetimini optimize etmek ve iş parçacıklarının verimli bir şekilde tahsis edilmesini sağlamak için önemlidir.

```
// Bu satır, Dispatchers.Default dispatcher'ını kullanarak belirli bir eşzamanlılık (conc\
urrency) sınırı tanımlar.
private val dispatcher = Dispatchers.Default
    .limitedParallelism(5)
```

Eğer daha önce `limitedParallelism` fonksiyonunu gördüyseniz, `Dispatchers.Default` ile `Dispatchers.IO` üzerinde tamamen farklı bir davranış sergilediğini bilmelisiniz. Bu farkları ilerleyen bölümlerde detaylı olarak ele alacağız.

> `limitedParallelism`, kotlinx-coroutines kütüphanesinin `1.6` sürümüyle birlikte tanıtılmıştır. Bu nedenle, eski projelerde bu fonksiyonun kullanımına rastlamayabilirsiniz.

Ana Dağıtıcı (Main Dispatcher)

Android ve birçok uygulama çerçevesi, genellikle kullanıcı arayüzü (UI) ile etkileşime giren tek iş parçacığı olan bir `ana iş parçacığı(Main Thread)` kavramına sahiptir. Android'de bu iş parçacığı özellikle önemlidir çünkü UI güncellemeleri yalnızca bu iş parçacığı üzerinden gerçekleştirilebilir.Bu nedenle, çok sık kullanılması gerekir fakat bu büyük bir özenle yapılmalıdır.. Ana iş parçacığı engellenirse (örneğin uzun süren bir işlem çalıştırılırsa), tüm uygulama donarak tepki vermez hale gelir.Android'de `Ana İş Parçacığı` üzerinde bir eşzamanlı işlem başlatmak için `Dispatchers.Main` kullanılır.

```
// 'suspend' anahtar kelimesi, bu fonksiyonun askıya alınabilir olduğunu belirtir. Yani, \
bu fonksiyon
// başka askıya alınabilir fonksiyonlar içinde çağrılabilir ve bekleyebilir.
suspend fun showUserName(name: String) =
// withContext, çalışma bağlamını (dispatcher) değiştirir. Bu örnekte, UI işlemlerini ana\
 iş parçacığında
    // (Main thread) yapabilmek için Dispatchers.Main kullanılır.
    withContext(Dispatchers.Main) {
        // userNameTextView'in metin değerini, fonksiyona geçirilen 'name' parametresiyle\
 günceller.
        userNameTextView.text = name
    }
```

`Dispatchers.Main`, Android platformunda kullanılabilmesi için `kotlinx-coroutines-android` bağımlılığı eklenerek etkinleştirilebilir. Benzer şekilde, JavaFX için `kotlinx-coroutines-javafx`, Swing için ise `kotlinx-coroutines-swing` bağımlılıkları eklenmelidir. Eğer bir proje, ana iş parçacığını belirleyen bir bağımlılığa sahip değilse, `Dispatchers.Main` kullanılamaz. Özellikle, ön yüz (frontend) kütüphaneleri birim testlerinde (unit tests) genellikle kullanılmaz. Bu nedenle, test ortamında `Dispatchers.Main` varsayılan olarak tanımlı değildir. Birim testlerinde `Dispatchers.Main` kullanabilmek için `kotlinx-coroutines-test` kütüphanesinde yer alan `Dispatchers.setMain(dispatcher)` fonksiyonu ile özel bir dağıtıcı (dispatcher) atanmalıdır.

```
class SomeTest {

    // Tek iş parçacıklı bir yürütücü (executor) oluşturulur ve
    // bu yürütücü bir coroutine dağıtıcısına (dispatcher) dönüştürülür.
    private val dispatcher = Executors
        .newSingleThreadExecutor()
        .asCoroutineDispatcher()

    @Before
    fun setup() {
        // Testler başlamadan önce, ana iş parçacığını (Main dispatcher)
        // tek iş parçacıklı özel dağıtıcıya yönlendirir.
        Dispatchers.setMain(dispatcher)
    }

    @After
    fun tearDown() {
        // Testler tamamlandıktan sonra, ana dağıtıcı orijinal
        // haline döndürülür.
        Dispatchers.resetMain()
        dispatcher.close() // Oluşturulan yürütücü kapatılır.
    }
```

```
23
24      @Test
25      fun testSomeUI() = runBlocking {
26          launch(Dispatchers.Main) {
27              // Burada ana iş parçacığı üzerinde çalışacak
28              // test edilecek UI kodları yer alır.
29          }
30      }
31  }
```

Android'de genellikle `Dispatchers.Main` varsayılan olarak kullanılır. Eğer kullandığınız kütüphaneler iş parçacıklarını engellemek yerine askıya alıyorsa (*suspending functions*) ve herhangi bir karmaşık hesaplama yapmıyorsanız, pratikte yalnızca `Dispatchers.Main` kullanmanız yeterli olabilir. Ancak, eğer CPU yoğun (*CPU-intensive*) işlemler yapıyorsanız, bunları `Dispatchers.Default` üzerinde çalıştırmanız gereklidir. Çoğu uygulama için `Dispatchers.Main` ve `Dispatchers.Default` yeterli olur.

Ancak, bazı durumlarda bir iş parçacığını engellemeniz (*blocking operations*) gerekebilir. Örneğin:

- Büyük dosyaların okunması veya yazılması gibi uzun süren G/Ç (*I/O*) işlemleri.
- Engelleyici (*blocking*) fonksiyonlar içeren bir kütüphanenin kullanılması.

Bu tür işlemleri ana iş parçacığında (*Main thread*) çalıştırmak, uygulamanın donmasına neden olur. Varsayılan dağıtıcıyı (*Dispatchers.Default*) engellemek, tüm iş parçacıklarının kilitlenmesine neden olabilir. Bu durumda, hesaplama yapmak için iş parçacığı bulunamayacağı için uygulama yanıt vermez hale gelir. Bu nedenle, bu tür engelleyici işlemleri yönetmek için `Dispatchers.IO` kullanılmalıdır. `Dispatchers.IO`, büyük G/Ç işlemlerini daha verimli bir şekilde yürütmek için tasarlanmış bir iş parçacığı havuzu sağlar.

Giriş/Çıkış (GÇ/IO) Dağıtıcısı

`Dispatchers.IO`, iş parçacıklarını engelleyen (*blocking*) G/Ç (*I/O*) işlemleri için tasarlanmıştır. Örneğin, dosya okuma/yazma işlemleri veya engelleyici (*blocking*) fonksiyon çağrıları gibi işlemler için kullanılır.

Aşağıdaki kodun yürütülmesi yaklaşık 1 saniye sürer çünkü `Dispatchers.IO` aynı anda 50'den fazla aktif iş parçacığının çalışmasına izin verir. Bu, büyük ölçekli G/Ç işlemlerinin verimli bir şekilde yönetilmesini sağlar.

```kotlin
// Bu program Kotlin Coroutines ile IO işlemlerinin paralelleştirilmesini göstermektedir
suspend fun main() {
    val time = measureTimeMillis {
        // measureTimeMillis fonksiyonu, içindeki kod bloğunun çalışma süresini milisaniye ci\
        // coroutineScope, tüm alt coroutine'lerin tamamlanmasını bekleyen bir kapsamdır
        coroutineScope {
            // repeat(50), aynı bloğu 50 kez tekrarlar
            repeat(50) {
                // launch, yeni bir coroutine başlatır
                launch(Dispatchers.IO) {
                    // Dispatchers.IO, bloke edici işlemler için optimize edilmiş bir thread \
                    // Thread.sleep(1000), mevcut thread'i 1 saniye boyunca bloke eder
                    Thread.sleep(1000)
                }
            }
        }
    }
    // Ölçülen süreyi ekrana yazdırır, yaklaşık 1000 milisaniye (1 saniye) olması beklenir
    println(time) // ~1000
}
```

> `Dispatchers.IO` yalnızca iş parçacıklarını engelleyen (*blocking*) API'lere ihtiyacınız olduğunda gereklidir.
> Eğer askıya alınabilir (*suspending*) fonksiyonlar kullanıyorsanız, herhangi bir dağıtıcıyı (*dispatcher*) kullanabilirsiniz. Ağ veya veritabanı işlemleri gibi askıya alınabilir fonksiyonlar sunan bir kütüphane kullanıyorsanız, `Dispatchers.IO` kullanmanıza gerek yoktur. Bu nedenle, birçok projede `Dispatchers.IO` kullanımı tamamen gereksiz olabilir.

`Dispatchers.IO`, teorik olarak sınırsız bir iş parçacığı havuzu (*thread pool*) gibi çalışır.
Başlangıçta boş olan bu havuz, ihtiyaç duyuldukça yeni iş parçacıkları oluşturur ve belirli bir süre kullanılmazlarsa devre dışı bırakır. Ancak, bu tür sınırsız bir iş parçacığı yönetimi doğrudan kullanıldığında güvenli değildir.
Çok fazla aktif iş parçacığı oluşturmak, performansın yavaş ve kontrolsüz bir şekilde düşmesine neden olur ve en sonunda bellek tükenmesi (*out-of-memory error*) gibi hatalara yol açabilir.

Bu yüzden Kotlin Coroutines kütüphanesindeki tüm temel dağıtıcılar, aynı anda kullanabilecekleri iş parçacığı sayısını sınırlandırır:

- `Dispatchers.Default`: İşlemcinizin çekirdek sayısı ile sınırlıdır.
- `Dispatchers.IO`: Aynı anda en fazla 64 iş parçacığına izin verir (Eğer sistemde 64'ten fazla çekirdek varsa, çekirdek sayısı kadar olabilir).

```
// Örnek Başlangıcı
suspend fun main() = coroutineScope {
    // 1000 adet eşzamanlı işlem başlatılır
    repeat(1000) {
        // IO işlemleri için özel dağıtıcı kullanılır
        launch(Dispatchers.IO) {
            // İş parçacığını 200 milisaniye boyunca bloke eder
            Thread.sleep(200)

            // Mevcut iş parçacığının adını alır
            val threadName = Thread.currentThread().name
            // Hangi iş parçacığında çalıştığını yazdırır
            println("Running on thread: $threadName")
        }
```

Daha önce belirttiğimiz gibi, `Dispatchers.Default` ve `Dispatchers.IO` aynı iş parçacığı havuzunu paylaşmaktadır.
Bu durum, önemli bir optimizasyon sağlar. İş parçacıkları yeniden kullanılır ve çoğu zaman yeniden dağıtım (*redispatching*) gerekmez. Örneğin, `Dispatchers.Default` üzerinde çalışan bir işlem (*coroutine*), `withContext(Dispatchers.IO) { ... }` bloğuna ulaştığında, genellikle aynı iş parçacığında çalışmaya devam eder[1]. Ancak burada değişen şey, iş parçacığınınDispatchers.Default sınırı yerine `Dispatchers.IO` sınırına dahil olmasıdır.

Bu iki dağıtıcının sınırları bağımsızdır, yani biri diğerini engelleyemez (*starvation* durumu oluşmaz).
Böylece CPU yoğun işlemler (`Dispatchers.Default`) ve bloklayan IO işlemleri (`Dispatchers.IO`) birbirlerini
engellemeden verimli bir şekilde çalıştırılabilir.

```
// Bu bir main fonksiyon ama suspend olarak tanımlanmış, yani coroutine içinde çalışabilir
suspend fun main(): Unit = coroutineScope {
    // Yeni bir coroutine başlatılıyor ve Dispatchers.Default üzerinde çalışacak şekilde \
ayarlanıyor
    // Dispatchers.Default, CPU yoğun işlemler için optimize edilmiş bir dispatcher
    launch(Dispatchers.Default) {
        // Şu anda çalışan thread'in adını yazdırıyor
        println(Thread.currentThread().name)

        // withContext ile çalışma bağlamı Dispatchers.IO'ya değiştiriliyor
        // Dispatchers.IO, I/O işlemleri için tasarlanmış, daha fazla eşzamanlılık sağlay\
```

[1]Bu mekanizma belirli bir çıktı üretmeyen, deterministik olmayan bir yapıdadır.

```
an bir dispatcher
        withContext(Dispatchers.IO) {
            // IO dispatcher bağlamına geçilmiş olmasına rağmen, hala aynı thread üzerind\
e çalışıyor mu kontrol ediliyor
            println(Thread.currentThread().name)
        }
    }
}
// Çıktı:
// DefaultDispatcher-worker-2  <- Default dispatcher üzerinde çalışırken kullanılan threa\
d adı
// DefaultDispatcher-worker-2  <- IO dispatcher'a geçildiğinde hala aynı thread kullanılı\
yor
```

Bu durumu daha net görmek için, hem `Dispatchers.Default` hem de `Dispatchers.IO` kullanılarak maksimum iş parçacığı (*thread*) kapasitesine ulaşıldığını hayal edin. Bu durumda, aktif iş parçacıklarının sayısı her iki dağıtıcının (*dispatcher*) sınırlarının toplamı kadar olacaktır. Örneğin, `Dispatchers.IO` için 64 iş parçacığına izin verildiğini ve sisteminizde 8 çekirdek bulunduğunu varsayarsak, paylaşılan havuzda toplam 72 aktif iş parçacığı olur. Bu, iş parçacıklarının verimli şekilde yeniden kullanılmasını sağlarken, aynı zamanda iki dağıtıcının birbirinden bağımsız şekilde çalışmasına olanak tanır.

`Dispatchers.IO` kullanımının en yaygın olduğu senaryo, harici kütüphanelerden gelen *bloklayan fonksiyonları* (*blocking functions*) çağırma ihtiyacıdır. En iyi uygulama (*best practice*), bu tür fonksiyonları `withContext(Dispatchers.IO)` bloğu içinde sarmalamaktır.

```
// DiscUserRepository sınıfı, UserRepository arayüzünü (interface) uygulayarak
// kullanıcı verilerini diskten okuma işlemini gerçekleştirir
class DiscUserRepository(
// DiscReader, dosya sisteminden veri okuyabilen bir bağımlılık
// Constructor enjeksiyonu ile dışarıdan sağlanıyor
    private val discReader: DiscReader
) : UserRepository {
    // UserRepository arayüzünden gelen getUser metodunu uygulama
// suspend fonksiyon olarak tanımlanmış, yani coroutine içinde çağrılabilir
    override suspend fun getUser(): UserData =
// IO dispatcher kullanılıyor çünkü disk okuma işlemi bir I/O operasyonudur
        withContext(Dispatchers.IO) {
// Diskten "userName" değerini okuyor ve bu değerle yeni bir UserData nesnesi oluşturuyor
            UserData(discReader.read("userName"))
        }
}
```

`Dispatchers.IO` için bir sınırın neden gerekli olduğunu daha iyi anlamak için, belirli aralıklarla çalışan ve çok sayıda eşzamanlı işlem başlatan bir görev (*task*) hayal edin. Örneğin, bir *bloklayan* API (*blocking* API) kullanarak toplu e-posta gönderimi yapmanız gerektiğini düşünelim.Bu işlem için SendGrid gibi harici bir e-posta servis sağlayıcısı kullanılabilir. Eğer `Dispatchers.IO` herhangi bir sınıra sahip olmasaydı, her e-posta için ayrı bir iş parçacığı (*thread*) oluşturulurdu. Yani, 100.000 e-posta gönderimi yapmanız gerekiyorsa, sistem 100.000 iş parçacığı başlatmaya çalışırdı. Bu da yaklaşık 100 GB RAM tüketimine neden olur ve uygulamanın çökmesine yol açardı. İşte bu nedenle, `Dispatchers.IO` belirli bir sınır ile çalışır. Böylece, iş parçacıklarının kontrolsüz şekilde artmasının önüne geçilir ve sistem kaynakları verimli bir şekilde kullanılır.

```
class NewsletterService {
    // SendGrid email servisi için bir API istemcisi oluşturuluyor
    // API_KEY değişkeni, SendGrid'e erişim için gereken kimlik anahtarını içeriyor
    private val sendGrid = SendGrid(API_KEY)

    // Bülten gönderme işlemini gerçekleştiren suspend fonksiyon
    suspend fun sendNewsletter(
        newsletter: Newsletter,  // Gönderilecek bültenin içeriği
        emails: List<Email>      // Bültenin gönderileceği email adresleri listesi
    ) = withContext(Dispatchers.IO) {  // IO dispatcher kullanılıyor çünkü bu bir ağ işle\
midir
        // Her email için ayrı bir coroutine başlatılıyor
        emails.forEach { email ->
            // Her email için paralel olarak yeni bir coroutine başlatılıyor
            launch {
                // SendGrid API'sini kullanarak email gönderiliyor
                // createNewsletter fonksiyonu, her alıcı için özelleştirilmiş bülten olu\
şturuyor
                sendGrid.api(createNewsletter(email, newsletter))
            }
        }
    }

    // Diğer yardımcı fonksiyonlar burada yer alıyor (kodda gösterilmemiş)
    // ...
}
```

Bu sınır, sistem kaynaklarını korur ancak işlemlerin daha uzun sürmesine neden olabilir. Örneğin, her bir e-posta gönderiminin ortalama 100 milisaniye sürdüğünü ve toplamda 100.000 e-posta gönderilmesigerektiğini düşünelim. `Dispatchers.IO` 64 iş parçacığı ile sınırlandırıldığında, tüm e-postaların gönderilmesiyaklaşık 3 dakika sürecektir. `Dispatchers.IO` ile ilgili temel

sorun, tüm uygulama genelinde tek bir sınıra sahip olmasıdır.Bu, bir servis (*service*) yoğun bir şekilde `Dispatchers.IO` kullanırken, diğer servislerin iş parçacığına erişimini engelleyebilir. Örneğin, aynı uygulama içerisinde hem toplu e-posta gönderimi yapan bir servis hem de kullanıcı kayıt işlemleri için onay e-postaları gönderen bir servis olduğunu düşünelim. Eğer her iki servis de `Dispatchers.IO` kullanıyorsa, öncelikli olarak toplu e-posta işlemi tamamlanana kadar kayıt onay e-postaları gönderilemeyecektir. Bu da kullanıcıların kayıt işlemi sırasında uzun süre beklemesine neden olur. Bu tür durumların yaşanmaması için, bağımsız sınırlara sahip özel dağıtıcılar (*dispatchers*) oluşturulmalıdır.

Özel Sınırlamalı Dağıtıcı

`Dispatchers.IO`, `limitedParallelism` fonksiyonuna özel bir davranış tanımlar. Bu fonksiyon, bağımsız bir iş parçacığı sınırı belirleyerek yeni bir dağıtıcı (*dispatcher*) oluşturur. Örneğin, her biri bir saniyeliğine bir iş parçacığını engelleyen 100 eşzamanlı işlem (*coroutine*) başlattığınızı düşünelim:

- Eğer bu işlemleri doğrudan `Dispatchers.IO` üzerinde çalıştırırsanız, toplam işlem süresi 2 saniye sürebilir.
- Eğer işlemleri `Dispatchers.IO.limitedParallelism(100)` üzerinde çalıştırırsanız,
  her biri aynı anda çalışabileceği için toplam işlem süresi sadece 1 saniye sürecektir.

Bu iki dağıtıcının (*dispatcher*) yürütme süreleri aynı anda ölçülebilir çünkü bu iki dağıtıcının sınırları birbirinden bağımsızdır. Bu sayede, `limitedParallelism` kullanarak farklı görevler için belirli bir iş parçacığı sınırı belirlemek mümkündür.Bu yöntem, kritik işlemlerin diğer işlemlerle çakışmasını önlemek için kullanışlıdır.

```
// Ana fonksiyon, suspend olarak işaretlenmiş ve bir coroutineScope içinde çalışıyor
suspend fun main(): Unit = coroutineScope {
    // İlk coroutine başlatılıyor
    launch {
        // Standard IO dispatcher ile ölçüm yapılıyor
        printCoroutinesTime(Dispatchers.IO)
        // Çıktı: "Dispatchers.IO took: 2074" - yaklaşık 2 saniye sürdü
    }

    // İkinci coroutine başlatılıyor
    launch {
        // Dispatchers.IO'dan özel, paralel işlem limiti 100 olan yeni bir dispatcher olu\
şturuluyor
        val dispatcher = Dispatchers.IO
            .limitedParallelism(100)
        printCoroutinesTime(dispatcher)
        // Çıktı: "LimitedDispatcher@XXX took: 1082" - yaklaşık 1 saniye sürdü
    }
}

// Belirli bir dispatcher ile 100 coroutine başlatıp bunların tamamlanma süresini ölçen f\
onksiyon
suspend fun printCoroutinesTime(
    dispatcher: CoroutineDispatcher
) {
    // Bu bloğun ne kadar sürdüğünü ölçmek için measureTimeMillis kullanılıyor
    val test = measureTimeMillis {
        coroutineScope {
            // 100 kez tekrarlayan bir döngü
            repeat(100) {
                // Her tekrarda belirtilen dispatcher üzerinde yeni bir coroutine başlatı\
lıyor
                launch(dispatcher) {
                    // Her coroutine içinde 1 saniye bekletiliyor (bloklandığına dikkat e\
din)
                    Thread.sleep(1000)
                }
            }
        }
        // coroutineScope, içindeki tüm coroutine'ler tamamlanana kadar bekler
    }
    // Toplam geçen süre ve hangi dispatcher'ın kullanıldığı yazdırılıyor
    println("$dispatcher took: $test")
}
```

Kavramsal olarak, `Dispatchers.Default` ve `Dispatchers.IO` tarafından kullanılan sınırsız bir iş parçacığı havuzu vardır, ancak her iki dağıtıcı da bu iş parçacıklarına sınırlı erişime sahiptir. `Dispatchers.IO` üzerinde `limitedParallelism` kullandığımızda, bağımsız bir iş parçacığı havuzuna sahip yeni bir dağıtıcı oluştururuz. Bu yeni dağıtıcı, `Dispatchers.IO` sınırlarından tamamen bağımsızdır.

Buna karşılık, eğer `limitedParallelism` fonksiyonunu `Dispatchers.Default` veya başka bir dağıtıcı üzerinde kullanırsak, orijinal dağıtıcıdaki sınırı koruyan, ancak ek bir sınıra sahip yeni bir dağıtıcı oluşturmuş oluruz. Yani, `Dispatchers.Default` üzerinde `limitedParallelism` kullanıldığında, oluşturulan yeni dağıtıcı hala orijinal sınırların içinde çalışır ve bağımsız bir havuz oluşturmaz.

```

// Sınırsız thread havuzuna sahip  dispatcher (temel thread havuzu)
private val pool = ...

Dispatchers.IO =   Thread havuzunu 64 thread ile sınırlar

Dispatchers.IO.limitedParallelism(x) = Thread Havuzu x degeri ile sınırlandırıldı

Dispatchers.Default = CPU çekirdek sayısı kadar thread ile sınırlıdır

Dispatchers.Default.limitedParallelism(x) =
    Default'un thread sayısını x ile sınırlar
```

![[Dispatchers.Default, işlemci çekirdek sayısı ile sınırlıdır. Dispatchers.IO, 64 iş parçacığı (veya çekirdek sayısı) ile sınırlıdır. Dispatchers.Default üzerinde `limitedParallelism` kullanıldığında, Dispatchers.Default'a ek bir sınır getiren bir dağıtıcı oluşturulur. Buna karşılık, Dispatcher.IO üzerinde `limitedParallelism` kullanıldığında, Dispatchers.IO'dan tamamen bağımsız bir sınırı olan bir dağıtıcı oluşturulur. Ancak, tüm bu dağıtıcılar aynı sınırsız iş parçacığı havuzunu paylaşır.]] (dispatchers_pools.png)

Birçok geliştirici, `limitedParallelism` fonksiyonunun `Dispatchers.IO` için farklı bir davranış sergilemesini kafa karıştırıcı bulmaktadır. Bunun nedeni, `limitedParallelism` ile oluşturulan sınırlamanın `Dispatchers.IO` sınırıyla doğrudan bir ilgisinin olmamasıdır. Bu durumu akla takrib için, bağımsız bir iş parçacığı sınırı olan bir dağıtıcı oluşturmayı açıkça belirten basit bir fonksiyon ekleyebiliriz:

```
/**
 * Belirtilen thread limitiyle sınırlı özel bir dispatcher oluşturur.
 * Bu, farklı iş tipleri için ayrı thread havuzları oluşturmayı sağlar.
 *
 * @param threadLimit Maksimum eşzamanlı thread sayısı
 * @return Thread sayısı sınırlı bir CoroutineDispatcher
 */
fun limitedDispatcher(threadLimit: Int) = Dispatchers.IO
        .limitedParallelism(threadLimit)
```

İş parçacıklarını yoğun bir şekilde engelleyebilecek sınıflar için en iyi uygulama, kendi bağımsız sınırlarına sahip özel dağıtıcılar tanımlamaktır. Bu sınır ne kadar büyük olmalıdır? Bunun kararını kendiniz vermelisiniz, ancak unutmayın ki çok fazla iş parçacığı kullanımı, kaynaklarımızın verimsiz kullanımına yol açabilir. Öte yandan, uygun bir iş parçacığı bulunmasını beklemek de performans açısından istenmeyen bir durumdur. En önemli nokta, belirlenen sınırın `Dispatchers.IO` ve diğer bir servis, başka bir servisin işlemlerini engellemez.

```
/**
 * Diskten kullanıcı verilerini okuyan repository sınıfı.
 *
 * @param discReader Disk okuma işlemlerini gerçekleştiren servis
 */
class DiscUserRepository(
    private val discReader: DiscReader
) : UserRepository {
    /**
     * Disk I/O işlemleri için maksimum 5 thread ile sınırlandırılmış özel dispatcher.
     * Not: Doğru metot adı 'limitedParallelism' olmalıdır, mevcut kod hatalıdır.
     */
    private val dispatcher = Dispatchers.IO
        .limitParallelism(5)

    /**
     * Disk üzerinden kullanıcı verilerini asenkron olarak okur.
     * Özel thread havuzunu kullanarak I/O işlemlerini gerçekleştirir.
     *
     * @return Disk üzerinden okunan kullanıcı verisi
     */
    override suspend fun getUser(): UserData =
        withContext(dispatcher) {
            UserData(discReader.read("userName"))
        }
}
```

Haber bülteni gönderme ve kayıt onay e-postaları örneğine geri dönelim. Bu hizmetlerin her biri, kendi sınırına sahip bir dağıtıcıya (dispatcher) sahip olmalıdır. Bu sayede birbirlerini engellemezler. Buradaki asıl soru, bu sınırların ne kadar büyük olması gerektiğidir. Ancak bunun basit bir cevabı yoktur. Bu, performans mı yoksa kaynak kullanımı mı daha önemli sorusuna bağlıdır. Eğer performansı ön planda tutuyorsak, sınırı çoğu zaman kullanılacak iş parçacığı sayısına göre ayarlamalıyız. Eğer kaynak kullanımını önemsiyorsak, sınırı en yoğun zamanlarda kullanılacak iş parçacığı sayısına göre belirlemeliyiz. Bu durumda, haber bültenlerinin gönderilmesinin biraz daha uzun sürmesi benim için sorun teşkil etmeyecektir; hatta bir saat sürse bile kabul edilebilir olur. Bu nedenle, bu iş için küçük bir sınır, örneğin 5, kullanabilirim. Öte yandan, kayıt onay e-postalarının mümkün olduğunca hızlı gönderilmesini istiyorum ve bu hizmetin aşırı yoğun şekilde kullanılacağını düşünmüyorum. Bu yüzden daha büyük bir sınır, örneğin 50, belirleyebilirim. Ancak, en uç durumlarda bile çok fazla iş parçacığını engellemek istemem. Çünkü bu, saldırılara karşı savunmasız hale gelmemize neden olabilir (çok fazla istek, yeterli sayıda iş parçacığını engelleyerek bellek taşmalarına(out-of-memory)yol açabilir).

```
// NewsletterService sınıfı toplu e-posta (bülten) gönderimini yönetir
class NewsletterService {
    // En fazla 5 paralel işleme izin veren özel dispatcher
    // Bu, API rate limit'leri aşmamak için önemli
    private val dispatcher = Dispatchers.IO.limitedParallelism(5)

    // E-posta gönderimi için SendGrid servisini kullanıyoruz
    private val sendGrid = SendGrid(API_KEY)

    // Bir bülteni birden çok e-posta adresine gönderen suspend fonksiyon
    suspend fun sendNewsletter(
        newsletter: Newsletter,
        emails: List<Email>
    ) = withContext(dispatcher) {
        // Listedeki her adres için paralel coroutine başlatır
        // Fakat dispatcher sayesinde aynı anda en fazla 5 tanesi çalışır
        emails.forEach { email ->
            launch {
                // Her adres için özelleştirilmiş bülten oluşturup gönder
                sendGrid.api(createNewsletter(email, newsletter))
            }
        }
    }

    // ...
}

```

```
// AuthorizationService kullanıcı doğrulama e-postaları göndermekten sorumlu
class AuthorizationService {
    // 50 paralel işleme izin veren dispatcher
    // Doğrulama e-postaları kritik olduğundan daha yüksek limit kullanılmış
    private val dispatcher = Dispatchers.IO.limitedParallelism(50)

    // Tek bir kullanıcıya doğrulama e-postası gönderen fonksiyon
    suspend fun sendAuthEmail(
        user: User
    ) = withContext(dispatcher) {
        // Kullanıcı için doğrulama e-postası oluştur ve gönder
        sendGrid.api(createConfirmationEmail(user))
    }

    // ...
}
```

Sabit İş Parçacığı Havuzuna Sahip Dağıtıcı (Dispatcher)

Bazı geliştiriciler, kullandıkları iş parçacığı havuzları üzerinde daha fazla kontrol sahibi olmayı tercih eder ve Java, bu amaçla oldukça güçlü bir API sunar. Örneğin, `Executors` sınıfını kullanarak sabit veya önbellekli bir iş parçacığı havuzu oluşturabiliriz. Bu havuzlar, `ExecutorService` veya `Executor` arayüzlerini uygular ve `asCoroutineDispatcher` fonksiyonu kullanılarak bir dağıtıcıya (dispatcher) dönüştürülebilir.

```
// Sabit sayıda thread içeren bir thread havuzu tanımlıyoruz
private val NUMBER_OF_THREADS = 20

// Java'nın Executors sınıfını kullanarak 20 threadlik sabit bir thread havuzu oluşturuyo\
ruz
// ve bunu Kotlin coroutine dispatcher'ına dönüştürüyoruz
val dispatcher = Executors
    // 20 threadlik sabit bir thread havuzu oluştur
    .newFixedThreadPool(NUMBER_OF_THREADS)
    // Java ExecutorService'i Kotlin CoroutineDispatcher'a dönüştür
    .asCoroutineDispatcher()
```

> `limitedParallelism`, `kotlinx-coroutines` kütüphanesinin 1.6 sürümünde tanıtılmıştır.
> Önceki sürümlerde, bağımsız iş parçacığı havuzlarına sahip dağıtıcılar genellikle `Executors` sınıfı kullanılarak oluşturuluyordu.

Bu yaklaşımın en büyük dezavantajı, `ExecutorService.asCoroutineDispatcher()` ile oluşturulan bir dağıtıcının `close` fonksiyonu ile kapatılmasının gerekmesidir.Geliştiriciler sıklıkla bu adımı unutmaktadır ve bu da iş parçacıklarının sızıntısına (thread leak) yol açmaktadır. Bir diğer sorun ise, sabit sayıda iş parçacığı içeren bir havuz oluşturulduğunda, kaynakların verimli şekilde kullanılamamasıdır. Kullanılmayan iş parçacıkları, diğer servislerle paylaşılmadan boşa çalıştırılmaya devam eder.

Tek İş Parçacığı ile Sınırlı Dağıtıcı (Dispatcher)

Çok iş parçacıklı dağıtıcılar kullanıldığında, paylaşılan durum sorunu dikkate alınmalıdır.
Aşağıdaki örnekte, 10.000 eşzamanlı işlem `i` değişkenini 1 artırmaktadır.
Teorik olarak, `i` değişkeninin değeri 10.000 olmalıdır, ancak daha küçük bir değer elde edilir.
Bunun nedeni, birden fazla iş parçacığının aynı anda paylaşılan değişken üzerinde değişiklik yapmasıdır.
Bu durum veri tutarsızlığı sorununa yol açar.

```
1  //Örnek Başlangıcı
2
3  // Global bir değişken, tüm coroutine'ler arasında paylaşılacak
4  var i = 0
5
6  // Ana fonksiyon suspend olarak işaretlenmiş, bu sayede coroutine'ler içinde çalışabilir
7  suspend fun main(): Unit = coroutineScope {
8      // 10.000 kez tekrar eden bir döngü
9      repeat(10_000) {
10         // Her döngüde Dispatchers.IO thread havuzunda yeni bir coroutine başlatılıyor
11         // Yorum'da belirtildiği gibi Dispatchers.Default da kullanılabilir
```

Bu sorunu çözmenin birçok yolu vardır (*Paylaşılan Durumlarla İlgili Sorunlar* bölümünde daha ayrıntılı ele alınacaktır) ,ancak olası çözümlerden biri, yalnızca tek bir iş parçacığı kullanan bir dağıtıcı (dispatcher) kullanmaktır.Eğer aynı anda yalnızca tek bir iş parçacığı kullanılırsa, ek bir senkronizasyon mekanizmasına gerek kalmaz.

Klasik yöntem, `Executors` kullanarak böyle bir dağıtıcı oluşturmaktı.Ancak, bu yöntem bir iş parçacığını sürekli aktif tutar ve artık kullanılmadığında kapatılması gerekir.Daha modern bir çözüm olarak, iş parçacıklarını

engellememek için `Dispatchers.Default` veya `Dispatchers.IO`kullanılarak ve paralelliği 1 ile sınırlandırarak tek iş parçacıklı bir dağıtıcı oluşturulabilir.

```
val dispatcher = Dispatchers.IO
    .limitedParallelism(1)

// Eski yöntem:
// val dispatcher = Executors.newSingleThreadExecutor()
//     .asCoroutineDispatcher()
```

```
//Örnek Başlangıcı
// Global bir sayaç değişkeni
var i = 0

// Ana coroutine kapsamında çalışan suspend fonksiyon
suspend fun main(): Unit = coroutineScope {
    // Dispatchers.Default dispatcher'ını limitedParallelism ile sınırlandırıyoruz
    // limitedParallelism(1) -> maksimum 1 iş parçacığı kullanılabilir
    // Bu, tüm coroutine'lerin aynı tek thread üzerinde sırayla çalışacağı anlamına gelir
    val dispatcher = Dispatchers.Default
        .limitedParallelism(1)

    // 10000 kez tekrar edecek bir döngü
    repeat(10000) {
```

Tek bir iş parçacığı ile sınırlı bir dağıtıcı (dispatcher) kullanmanın en büyük dezavantajı, eğer bu iş parçacığını engellersek çağrıların ardışık olarak işlenecek olmasıdır.

```
// Ana coroutine kapsamında çalışan suspend fonksiyon
suspend fun main(): Unit = coroutineScope {
    // Dispatchers.Default dispatcher'ını limitedParallelism ile sınırlandırıyoruz
    // limitedParallelism(1) -> maksimum 1 iş parçacığı kullanılabilir
    val dispatcher = Dispatchers.Default
        .limitedParallelism(1)

    // Özel dispatcher ile bir coroutine başlatıyoruz ve referansını saklıyoruz
    val launch = launch(dispatcher) {
        // Bu coroutine içinde 5 kez tekrar ediyoruz
        repeat(5) {
            // Her bir tekrarda YENİ BİR coroutine başlatıyoruz
            // DİKKAT: Bu iç coroutine'ler için özel bir dispatcher belirtilmemiş
            // Bu nedenle bu iç coroutine'ler üst coroutine'in dispatcher'ını DEĞİL,
            // varsayılan dispatcher'ı (Dispatchers.Default) kullanır ve PARALEL çalışabi\
```

```
lirler          launch {
                    // Her bir iç coroutine, thread'i 1 saniye boyunca bloklayacak
ir                  // NOT: Coroutine bağlamında Thread.sleep kullanmak iyi bir pratik değild\
                    // Bunun yerine delay() kullanılmalıdır, ancak bu örnekte özellikle
                    // thread'in bloklanması gösterilmektedir
                    Thread.sleep(1000)
                }
            }
        }

    // Ana coroutine, başlattığımız dış coroutine'in tamamlanmasını bekliyor
    // ve bu işlemin ne kadar sürdüğünü ölçüyor
    val time = measureTimeMillis { launch.join() }

    // Ölçülen süreyi yazdırıyoruz
    // Sonuç yaklaşık 5 saniye olacak çünkü 5 iç coroutine PARALEL çalışıyor
    println("Took $time") // Took 5006
}
```

Project Loom'dan Gelen Sanal İş Parçacıklarını Kullanma

JVM platformu, *Project Loom* olarak bilinen yeni bir teknoloji sunmuştur[2]. Bu teknolojinin en büyük yeniliği, geleneksel iş parçacıklarına kıyasla çok daha hafif olan sanal iş parçacıkları (virtual threads) kavramını getirmesidir. Sanal iş parçacıklarının engellenmesi, geleneksel iş parçacıklarının engellenmesine kıyasla çok daha düşük bir maliyetle gerçekleşmektedir.

Project Loom, Kotlin Coroutines'ı etkin bir şekilde kullanan geliştiriciler için büyük bir avantaj sunmamaktadır, çünkü Kotlin Coroutines zaten zahmetsiz iptal mekanizmaları ve test süreçlerinde sanal zaman yönetimi gibi oldukça gelişmiş özellikler sağlamaktadır[3]. Ancak, Project Loom'un sanal iş parçacıkları, `Dispatchers.IO` yerine kullanılabilir ve iş parçacıklarının engellenmesinin kaçınılmaz olduğu durumlarda faydalı olabilir[4].

Project Loom'u kullanabilmek için, JVM'in 19 veya daha üst bir sürümünü çalıştırmak gereklidir. Ayrıca, ön izleme özelliklerini etkinleştirmek için `--enable-preview` bayrağının kullanılması zorunludur. Daha sonra, `Executors`

[2]Bu ifadeyi iyimser bir bakış açısıyla yazıyorum; umuyorum ki bu metni okuduğunuzda *Project Loom* kararlı bir sürüme ulaşmış olur. Ancak, bu yazının hazırlandığı 2023 yılının başlarında, henüz kararlı bir sürüm mevcut değildir.(Project Loom'un kararlı sürümü Java 21 ile birlikte Eylül 2023'te yayınlandı. Java 21'de Virtual Threads (JEP 444) ve Structured Concurrency (JEP 453) gibi Project Loom'un ana özellikleri resmi olarak eklendi.)

[3]Bu konu, *Kotlin Süreçlerinin Test Edilmesi* bölümünde ayrıntılı olarak ele alınacaktır.

[4]Çözüm, *Jan Vladimir Mostert* tarafından yazılan *Running Kotlin coroutines on Project Loom's virtual threads* başlıklı makaleden esinlenilmiştir.

sınıfındaki `newVirtualThreadPerTaskExecutor` metoduyla bir yürütücü (executor) oluşturabilir ve bunu bir *coroutine* dağıtıcısına dönüştürebiliriz.

```
// Project Loom'un sanal thread'lerini kullanan bir CoroutineDispatcher tanımlıyoruz
val LoomDispatcher = Executors
    // Java 21'de tanıtılan, her görev için yeni bir sanal thread oluşturan bir executor
    .newVirtualThreadPerTaskExecutor()
    // Java Executor'ını Kotlin CoroutineDispatcher'a dönüştüren extension fonksiyon
    .asCoroutineDispatcher()
```

Bu tür bir dağıtıcı, nesne bildirimi (*object declaration*) kullanılarak da uygulanabilir:

```
// LoomDispatcher, bir singleton (tek örnek) nesne olarak tanımlanmış
// ve ExecutorCoroutineDispatcher sınıfından türetilmiştir
// Bu sayede Kotlin coroutine'leri ile Java sanal thread'leri arasında köprü görevi görür
object LoomDispatcher : ExecutorCoroutineDispatcher() {

    // Java Executor arayüzünü uygulayan, sanal thread başlatan bir executor tanımlıyoruz
    // Bu executor, her çağrıldığında yeni bir sanal thread başlatacak
    override val executor: Executor = Executor { command ->
        // Her görev için yeni bir sanal thread oluşturulur
        Thread.startVirtualThread(command)
        // Sanal thread'ler platform thread'lerine göre çok daha hafiftir ve milyonlarca \
oluşturulabilir
    }

    // dispatch metodu, coroutine'lerin çalışma zamanı tarafından çağrılır
    // Bu metod, bir coroutine bloğunu çalıştırmak için gerekli işlemleri gerçekleştirir
    override fun dispatch(
        // Coroutine'in bağlamı
        context: CoroutineContext,
        // Çalıştırılacak coroutine bloğu
        block: Runnable
    ) {
        // Yukarıda tanımladığımız executor'ı kullanarak bloğu çalıştırırız
        executor.execute(block)
        // Bu, pratik olarak her coroutine bloğunun yeni bir sanal thread üzerinde çalışa\
cağı anlamına gelir
    }

    // close metodu, dispatcher'ın kaynaklarını serbest bırakmak için kullanılır
    // Bu özel dispatcher'da, kapatma işlemine izin vermiyoruz
    override fun close() {

        error("Dispatchers.LOOM üzerinde çağrılamaz")
        // Hata fırlatarak kapatma denemelerini engelliyoruz
        // Çünkü bu dispatcher singleton olduğundan ve sanal thread'lerin yaşam döngüsünü
        // JVM kendisi yönettiğinden, manuel kapatmaya gerek yoktur
    }
}
```

Bu dağıtıcıyı diğer dağıtıcılar gibi kullanabilmek için, `Dispatchers` nesnesi üzerinde bir genişletme özelliği tanımlayabiliriz. Bu, aynı zamanda bu dağıtıcının keşfedilebilirliğini artıracaktır.

```
// Dispatchers sınıfına bir extension property (genişletme özelliği) ekliyoruz
// Bu, Dispatchers.Loom şeklinde erişilebilen yeni bir özellik tanımlar
val Dispatchers.Loom: CoroutineDispatcher
    // Bu özelliğe erişildiğinde LoomDispatcher nesnesini döndürür
    get() = LoomDispatcher
```

Şimdi, yeni dağıtıcımızın gerçekten bir iyileştirme olup olmadığını test etmemiz gerekiyor. Beklentimiz, iş parçacıklarını (thread) engellediğimizde diğer dağıtıcılara kıyasla daha az bellek ve işlemci süresi tüketmesidir. Bunun için ya hassas ölçümler yapabileceğimiz bir test ortamı kurabiliriz ya da farkın herkes tarafından gözlemlenebileceği aşırı bir senaryo oluşturabiliriz. Bu kitapta, ikinci yaklaşımı tercih ettim.

100.000 eşzamanlı işlem başlatarak her birini 1 saniye boyunca engelledim. Bunların başka bir işlem yapması, örneğin ekrana bir şey yazdırması veya bir değeri artırması, sonucu önemli ölçüde değiştirmezdi. Bu işlemlerin tamamının `Dispatchers.Loom` üzerinde çalıştırılması yaklaşık iki saniye sürdü.

```
suspend fun main() = measureTimeMillis {
    coroutineScope {
        repeat(100_000) {
            launch(Dispatchers.Loom) {
                Thread.sleep(1000)
            }
        }
    }
}.let(::println) // 2 273
```

Bu yeni dağıtıcıyı alternatif bir seçenekle karşılaştıralım. `Dispatchers.IO`yu doğrudan kullanmak adil olmaz, çünkü 64 iş parçacığı ile sınırlıdır ve bu durumda işlevin tamamlanması 26 dakikadan fazla sürecektir. Bunun yerine, iş parçacığı sınırını eşzamanlı işlemlerin (coroutine) sayısına yükseltmeliyiz. Bu sınırı artırarak testi gerçekleştirdiğimde, kodun yürütülmesi 23 saniyeden fazla sürdü, yani `Dispatchers.Loom` sürümüne kıyasla on kat daha uzun. Ayrıca, `Dispatchers.IO` sürümü çok daha fazla bellek ve işlemci süresi tüketti. Bu nedenle, mümkün olduğunda `Dispatchers.IO` yerine `Dispatchers.Loom` kullanmalıyız.

```
// Bu bir suspend fonksiyonu, yani asenkron ve coroutine içinden çağrılabilir
// main fonksiyonu tüm işlemin ne kadar sürdüğünü ölçmek için measureTimeMillis içine alı\
nmış
suspend fun main() = measureTimeMillis {
    // coroutineScope, içindeki tüm coroutine'ler tamamlanana kadar bekler
    coroutineScope {
        // 100.000 kez aynı işlemi tekrar eder
        repeat(100_000) {
            // Her tekrarda Loom Dispatcher kullanarak yeni bir coroutine başlatır
            launch(Dispatchers.Loom) {
                // Her coroutine 1 saniye bekler
                Thread.sleep(1000)
            }
        }
    }
    // Tüm coroutine'ler tamamlandıktan sonra ölçüm biter
}.let(::println)
// Ölçülen süreyi (milisaniye cinsinden) konsola yazdırır, sonuç: 2273 ms
```

Şu an itibariyle, Project Loom hâlâ yeni ve kullanımı zor, ancak `Dispatchers.IO` için heyecan verici bir alternatif olduğunu söylemeliyim. Bununla birlikte, gelecekte bunu doğrudan kullanmanız gerekmeyecek, çünkü Kotlin Coroutines ekibi, Project Loom stabil hâle geldiğinde sanal iş parçacıklarını varsayılan olarak kullanmayı planladıklarını belirtti. Umarım bu süreç en kısa sürede gerçekleşir.

Bağlı Olmayan Dağıtıcı (Unconfined Dispatcher)

Ele almamız gereken son dağıtıcı `Dispatchers.Unconfined`dir. Bu dağıtıcı, diğer dağıtıcılardan farklıdır çünkü herhangi bir iş parçacığını değiştirmez. Başlatıldığında, hangi iş parçacığında başlatıldıysa onun üzerinde çalışır. Eğer askıya alınıp yeniden devam ettirilirse, onu devam ettiren iş parçacığı üzerinde çalışmaya devam eder.

```
//Örnek Başı
fun main() {
    // Coroutine'i devam ettirebilmek için gerekli olan continuation referansını saklayac\
ak değişken
    var continuation: Continuation<Unit>? = null

    // "Thread1" adında yeni bir thread başlatıyoruz
    thread(name = "Thread1") {
        // Dispatcher.Unconfined kullanarak bir coroutine başlatıyoruz
        // Unconfined dispatcher, coroutine'in başladığı thread'de çalışır ve suspend nok\
talarından
        // sonra çağıran thread'de devam eder
        runBlocking(Dispatchers.Unconfined) {
            // İlk başta Thread1 üzerinde çalışıyoruz
            println(Thread.currentThread().name) // Thread1 yazdırır

            // suspendCancellableCoroutine fonksiyonu, coroutine'i askıya alır ve continu\
ation
            // nesnesini lambda fonksiyonuna sağlar
            suspendCancellableCoroutine {
                // Continuation'ı daha sonra kullanmak üzere saklıyoruz
                continuation = it
                // Buradan sonra coroutine askıya alınır ve continuation.resume çağrılana\
 kadar bekler
            }

            // Coroutine tekrar başladığında, şimdi Thread2 üzerinde çalışıyoruz
            // Çünkü continuation.resume Thread2 üzerinden çağrıldı
            println(Thread.currentThread().name) // Thread2 yazdırır
```

Eski projelerde, birim testlerinde `Dispatchers.Unconfined` kullanımına rastlamak mümkündür. Örneğin, `launch` çağıran bir fonksiyonu test etmek istediğinizi düşünelim; zaman senkronizasyonu sağlamak kolay olmayabilir. Bunun bir çözümü, diğer tüm dağıtıcılar yerine `Dispatchers.Unconfined` kullanmaktır. Eğer tüm kapsamlar (`scope`) içinde bu dağıtıcı kullanılırsa, her şey aynı iş parçacığı üzerinde çalışır ve işlem sırasını daha kolay kontrol edebiliriz. Ancak bu yöntem artık gereksizdir, çünkü Kotlin Coroutines için test yaparken kullanabileceğimiz çok daha iyi araçlar bulunmaktadır. Bu konuyu kitabın ilerleyen bölümlerinde ele alacağız.

Performans açısından bakıldığında, `Dispatchers.Unconfined` en hafif dağıtıcıdır, çünkü iş parçacıkları arasında geçiş yapmaya gerek duymaz. Bu nedenle, kodumuzun hangi iş parçacığında çalıştığını önemsemediğimiz durumlarda tercih edilebilir. Ancak, pratikte bu dağıtıcıyı rastgele kullanmak iyi bir yaklaşım olarak kabul edilmez. Peki ya yanlışlıkla bir engelleyici (blocking) işlem çağırırsak ve işlem `Main` iş parçacığında çalışıyorsa? Bu durum, tüm uygulamanın donmasına neden olabilir. İşte bu yüzden, `Dispatchers.Unconfined` üretim kodlarında özel durumlar dışında kullanılmamalıdır.

Anında Ana İş Parçacığı Görevlendirmesi (Immediate Main Dispatching)

Bir eşzamanlı işlemi (coroutine) görevlendirmenin belirli bir maliyeti vardır. `withContext` çağrıldığında, işlem askıya alınmalı, olası bir sıraya alınmalı ve ardından tekrar devam ettirilmelidir. Eğer zaten aynı iş parçacığı üzerinde çalışıyorsak, bu küçük ama gereksiz bir maliyettir. Aşağıdaki fonksiyona göz atalım:

```
// Bu bir suspend fonksiyon, bu nedenle sadece bir coroutine içinden veya
// başka bir suspend fonksiyonundan çağrılabilir
suspend fun showUser(user: User) =
// withContext, belirtilen dispatcher'da kod bloğunu çalıştırır ve
    // bloğun son ifadesini döndürür
    withContext(Dispatchers.Main) {
        // Dispatchers.Main, ana UI thread'i üzerinde çalışır
        // Bu satır kullanıcı adını UI elemanına atar
        userNameElement.text = user.name
        // Burada diğer UI güncellemeleri olabilir
        // ...
    }
```

Eğer bu fonksiyon zaten ana iş parçacığında (Main dispatcher) çağrılmış olsaydı, gereksiz bir yeniden görevlendirme maliyeti oluşurdu. Daha da önemlisi, `withContext` nedeniyle ana iş parçacığında uzun bir işlem kuyruğu oluşursa, kullanıcı verileri gecikmeli olarak gösterilebilir (çünkü bu eşzamanlı işlem, diğer işlemlerin tamamlanmasını beklemek zorunda kalacaktır). Bunu önlemek için `Dispatchers.Main.immediate` kullanılır. Bu yapı, yalnızca gerektiğinde iş parçacığını yeniden görevlendirir. Böylece, aşağıdaki fonksiyon eğer zaten ana iş parçacığında çağrılmışsa, tekrar görevlendirilmez ve doğrudan çalıştırılır.

```
suspend fun showUser(user: User) =
    // withContext, belirtilen dispatcher'da kod bloğunu çalıştırır
    withContext(Dispatchers.Main.immediate) {
        // Dispatchers.Main.immediate ana UI thread'i üzerinde çalışır,
        // "immediate" özelliği sayesinde eğer zaten Main thread'deyse
        // gereksiz dispatcher değişimini atlar ve kodu anında çalıştırır
        userNameElement.text = user.name
        // Burada diğer UI güncellemeleri olabilir
        // ...
    }
```

Bir fonksiyonun zaten ana iş parçacığında (Main dispatcher) çağrılmış olabileceği bir durumda `withContext` kullanmamız gerekiyorsa, argüman olarak `Dispatchers.Main.immediate` tercih edilir. Bu sayede, eğer mevcut iş parçacığı zaten Main dispatcher ise, gereksiz bir yeniden görevlendirme işlemi yapılmaz ve işlem doğrudan çalıştırılır. Şu an itibariyle, diğer dağıtıcılar (dispatchers) anında görevlendirme (immediate dispatching) desteği sunmamaktadır.

Devamlılık (Continuation) Kesici

Görevlendirme (dispatching) işlemi, Kotlin diline entegre edilmiş devamlılık kesme (continuation interception) mekanizmasına dayanır. Bu bağlamda, `ContinuationInterceptor` adlı bir coroutine bağlamı (`CoroutineContext`) bulunmaktadır. Bu bağlam, `interceptContinuation` metodunu kullanarak bir coroutine askıya alındığında (suspend edildiğinde) devamlılığı (continuation) değiştirme yeteneğine sahiptir[5].

Ayrıca, `releaseInterceptedContinuation` metodu, kesilmiş devamlılık işlemi sona erdiğinde çağrılır. Bu mekanizma, coroutine'lerin farklı iş parçacıkları veya dağıtıcılar (dispatchers) üzerinde yönetilmesini sağlar.

[5]Önbellekleme mekanizması sayesinde her devamlılık için sarmalama işlemi yalnızca bir kez gerçekleştirilmelidir.

```
// ContinuationInterceptor, CoroutineContext'in bir elementi olarak tanımlanmış
// bir arayüzdür (interface). Coroutine'lerin execution akışını kontrol eder.
public interface ContinuationInterceptor :
    CoroutineContext.Element {

    // Key sınıfı, CoroutineContext içindeki bu elementin anahtarıdır
    // Bu anahtar, context içinden ContinuationInterceptor'ı almak için kullanılır
    // Örn: context[ContinuationInterceptor] şeklinde erişilir
    companion object Key :
        CoroutineContext.Key<ContinuationInterceptor>

    // Bu metot, bir continuation'ı intercept eder (araya girer) ve ona müdahale edebilir
    // Orijinal continuation'ı alır ve genellikle onu sarmalayan (wrapping) özelleştirilm\
iş
    // bir continuation döndürür. Bu sarmalama, coroutine'in çalışma ortamını değiştirmek\
 için kullanılır
    // Örneğin, belirli bir thread'e veya thread havuzuna dispatch etmek için
    fun <T> interceptContinuation(
        continuation: Continuation<T>
    ): Continuation<T>

    // Bu metot, intercepted (müdahale edilmiş) bir continuation'ı
    // kullanımdan sonra serbest bırakmak için çağrılır
    // Kaynak temizliği için fırsat sağlar. Varsayılan implementasyonu boştur,
    // ihtiyaç duyulduğunda interceptor tarafından override edilebilir
    fun releaseInterceptedContinuation(
        continuation: Continuation<*>
    ) {
    }

    //... (Diğer metotlar)
}
```

Coroutine'in devamlılığını (continuation) sarmalama yeteneği, büyük bir kontrol imkanı sunar. Dağıtıcılar (dispatchers), `interceptContinuation` metodunu kullanarak devamlılığı `DispatchedContinuation` ile sararlar. Bu yapı, belirli bir iş parçacığı havuzunda çalışmasını sağlar ve dağıtıcıların temel çalışma prensibini oluşturur. Ancak aynı bağlam (context), `kotlinx-coroutines-test` kütüphanesindeki `runTest` gibi birçok test kütüphanesi tarafından da kullanılır. Coroutine bağlamındaki her elemanın benzersiz bir anahtarı (key) olması gerektiğinden, birim testlerinde genellikle dağıtıcıları test dağıtıcılarıyla değiştirmek için bağlama enjekte ederiz. Bu konuya, coroutine testlerine ayrılmış bölümde daha ayrıntılı olarak değineceğiz.

```
// Kullanıcı veritabanı işlemlerini gerçekleştiren repository sınıfı
// Diskten kullanıcı verilerini okur ve UserRepository arayüzünü uygular
class DiscUserRepository(
  // Disk okuma işlemlerini gerçekleştiren bileşen
  private val discReader: DiscReader,
  // İşlemlerin hangi coroutine bağlamında (thread'de) çalışacağını belirler
  // Varsayılan olarak Dispatchers.IO kullanılır (IO işlemleri için optimize edilmiş thre\
ad havuzu)
  private val dispatcher: CoroutineContext = Dispatchers.IO,
) : UserRepository {
  // Kullanıcı verisini asenkron olarak getiren suspend fonksiyon
  override suspend fun getUser(): UserData =
    // İşlemleri belirtilen dispatcher'a (genellikle IO thread) yönlendirir
    withContext(dispatcher) {
      // Diskten "userName" değerini okur ve UserData nesnesine dönüştürür
      UserData(discReader.read("userName"))
    }
}

// Repository sınıfının birim testlerini içeren test sınıfı
class UserReaderTests {

  // Test fonksiyonu - runTest ile özel coroutine test ortamında çalıştırılır
  @Test
  fun `some test`() = runTest {
    // HAZIRLIK AŞAMASI (GIVEN)
    // Gerçek disk okuma işlemleri yerine test için özel sahte bir implementasyon kullanı\
lır
    val discReader = FakeDiscReader()

    // Test edilecek repository örneği oluşturulur
    val repo = DiscUserRepository(
      discReader,
      // TEST ORTAMI ENTEGRASYONU
      // Test coroutine'inin dispatcher'ını kullanarak gerçek IO thread'leri yerine
      // test ortamının kontrolünde çalışmasını sağlarız
      // ContinuationInterceptor, dispatcher mantığını yöneten temel bileşendir
      this.coroutineContext[ContinuationInterceptor]!!
    )
    //... (Test kodunun devamı)
  }
}
```

Dağıtıcıların farklı görevlerdeki performansı

Farklı dağıtıcıların çeşitli görevlerde nasıl performans gösterdiğini göstermek için bazı karşılaştırmalar yaptım. Tüm durumlarda, aynı görevi gerçekleştiren 100 bağımsız coroutine çalıştırılmıştır. Tablodaki sütunlar farklı görevleri temsil etmektedir:

- Bir saniye askıya alma (süreç bekleme)
- Bir saniye engelleme (bloklama)
- CPU yoğun işlemler
- Bellek yoğun işlemler (çoğunlukla bellek erişimi, tahsis ve serbest bırakma sürecinde zaman harcayan işlemler)

Satırlar ise bu süreçlerin çalıştırıldığı farklı dağıtıcıları göstermektedir. Aşağıdaki tablo, ortalama yürütme süresini milisaniye cinsinden göstermektedir.

	Askıya alma	**Engelleme**	**CPU**	**Bellek**
Tek iş parçacığı	1 002	100 003	39 103	94 358
Default (8 iş parçacığı)	1 002	13 003	8 473	21 461
Giriş/Çıkış-IO (64 iş parçacığı)	1 002	2 003	9 893	20 776
100 iş parçacığı	1 002	1 003	10 379	21 004

Bu verilerden çıkarılabilecek bazı önemli gözlemler şunlardır:

1. Süreç yalnızca askıya alınıyorsa, kullanılan iş parçacığı sayısı performans üzerinde belirgin bir etkiye sahip değildir.
2. Engelleme işlemleri söz konusu olduğunda, kullanılan iş parçacığı sayısı arttıkça tüm süreçlerin tamamlanma süresi kısalır.
3. CPU yoğun işlemler için `Dispatchers.Default` en verimli seçenektir[6].
4. Bellek yoğun işlemlerde, daha fazla iş parçacığı kullanımı belirli bir performans artışı sağlayabilir; ancak bu artış genellikle sınırlıdır ve önemli bir fark yaratmaz.

Aşağıda test edilen fonksiyonlar gösterilmektedir[7]:

[6] Bunun temel nedeni, daha fazla iş parçacığı kullanıldıkça işlemcinin bu iş parçacıkları arasında geçiş yapmak için daha fazla zaman harcaması ve böylece anlamlı işlemleri yürütmek için daha az zamanının kalmasıdır. Ayrıca, Dispatchers.IO engelleyici işlemler için tasarlanmıştır ve CPU yoğun işlemler için kullanılmamalıdır, çünkü bu durumda başka bir süreç tüm iş parçacıklarını bloke edebilir.

[7] Tüm kod, şu adreste bulunabilir: https://kt.academy/dispatchers-benchmarks.

```
// Yoğun CPU kullanımı gerektiren bir fonksiyon
// CPU'yu yoğun şekilde kullanarak bir sipariş nesnesinden kahve üretir
fun cpu(order: Order): Coffee {
  // Çok büyük bir sayıdan başlayarak yoğun hesaplama yapar
  var i = Int.MAX_VALUE
  while (i > 0) {
    // Sayının çift/tek durumuna göre farklı şekilde azaltarak CPU'yu yoğun çalıştırır
    i -= if (i % 2 == 0) 1 else 2
  }
  // Hesaplama sonucunu kullanarak yeni bir Coffee nesnesi döndürür
  return Coffee(order.copy(customer = order.customer + i))
}

// Yüksek bellek kullanımı gerektiren bir fonksiyon
// Büyük nesneler oluşturarak belleği yoğun şekilde kullanır
fun memory(order: Order): Coffee {
  // İç içe listeler oluşturarak bellek kullanımını artırır
  // 1. seviye liste
  val list = List(1_000) { it }
  // 2. seviye liste (liste içinde liste)
  val list2 = List(1_000) { list }
  // 3. seviye liste (liste içinde liste içinde liste)
  val list3 = List(1_000) { list2 }

  // Büyük listenin hash kodunu kullanarak Coffee nesnesi döndürür
  return Coffee(
    order.copy(
      customer = order.customer + list3.hashCode()
    )
  )
}

// Thread'i bloke eden fonksiyon
// Thread.sleep() kullanarak çağrıldığı thread'i bloke eder
fun blocking(order: Order): Coffee {
  // Thread'i 1 saniye durdurur (bloke eder)
  Thread.sleep(1000)
  // Siparişten kahve üretir
  return Coffee(order)
}

// Coroutine'i askıya alan (suspend) fonksiyon
// delay() kullanarak coroutine'i askıya alır, thread'i bloke etmez
suspend fun suspending(order: Order): Coffee {
  // Coroutine'i 1 saniye askıya alır, thread serbest kalır
  delay(1000)
  // Siparişten kahve üretir
  return Coffee(order)
}
```

Özet

Dağıtıcılar, bir sürecin hangi iş parçacığında veya iş parçacığı havuzunda başlatılacağını ve yürütüleceğini belirleyen önemli bileşenlerdir. Doğru dağıtıcı seçimi, eşzamanlı işlem yönetimi, kaynak kullanımı ve uygulama performansı açısından kritik öneme sahiptir. Aşağıda en yaygın kullanılan dağıtıcılar ve kullanım alanları açıklanmaktadır:

- `Dispatchers.Default`: Çok iş parçacıklı işlemci havuzunu (CPU-bound thread pool) kullanarak çalışır ve işlemci yoğun hesaplamalar gerektiren görevler için uygundur. İşlemci çekirdek sayısına bağlı olarak iş parçacıklarını ölçeklendirir, böylece optimum performans sağlar.
- `Dispatchers.Main`: Android, Swing veya JavaFX gibi tek iş parçacıklı kullanıcı arayüzü (UI) odaklı platformlarda ana iş parçacığında çalışacak görevler için kullanılır. Kullanıcı arayüzü bileşenlerinin değiştirilmesi veya kullanıcı etkileşimlerinin yönetilmesi gibi işlemler bu bağlamda gerçekleştirilir.
- `Dispatchers.Main.immediate`: Dispatchers.Main ile aynı iş parçacığında çalışır, ancak yürütülmesi gereken görev ana iş parçacığında zaten çalışıyorsa ek bir yeniden planlama (redispatching) gerçekleştirmez. Bu, gereksiz bağlam değişikliklerini (context switch) önleyerek işlem maliyetini düşürür.
- `Dispatchers.IO`: Ağ, dosya okuma/yazma ve veritabanı erişimi gibi engelleyici (blocking) işlemler için optimize edilmiş bir iş parçacığı havuzu sağlar. Engelleyici işlemlerin ana iş parçacığında yürütülmesini önleyerek uygulamanın tepki süresini artırır.
- `Dispatchers.IO` (sınırlı paralellik ile- limitedParallelism()) veya özel bir iş parçacığı havuzu: Çok sayıda engelleyici çağrının (örneğin, çok sayıda ağ isteğinin veya dosya işleminin) eş zamanlı olarak yürütülmesi gerektiğinde, belirli bir iş parçacığı sayısıyla sınırlandırılmış bir paralel çalışma modeli kullanılabilir. Böylece, iş parçacıklarının aşırı yüklenmesi engellenerek kaynak tüketimi dengelenir.
- `Dispatchers.Default` veya `Dispatchers.IO` (paralellik 1 ile sınırlandırılmış-limitedParallelism(1)) ya da tek iş parçacıklı özel bir dağıtıcı: Paylaşılan veri durumunu değiştiren (shared state modification) işlemlerde kullanılır. Tek iş parçacıklı yürütme modeli, aynı anda birden fazla süreç tarafından değiştirilen ortak verinin tutarlılığını koruyarak veri yarış durumlarını (race condition) ve belirsiz davranışları önler.

- `Dispatchers.Unconfined`: Sürecin hangi iş parçacığında yürütüleceğini başlangıçta belirlemez ve yürütme, çağıran iş parçacığında başlar. İşlem sürecinin devamında başka bir iş parçacığına aktarılması mümkündür. Genellikle belirli özel senaryolar ve optimizasyonlar için tercih edilir, ancak yanlış kullanımı beklenmeyen eşzamanlılık hatalarına neden olabilir.

Dağıtıcıların doğru seçilmesi, süreçlerin verimli ve öngörülebilir bir şekilde yürütülmesini sağlamak için gereklidir. Bu seçim, uygulamanın türüne, yürütülen görevlerin özelliklerine ve performans gereksinimlerine bağlı olarak dikkatlice yapılmalıdır.

•

Alıştırma: Dağıtıcıları Kullanma

Aşağıda verilen her bir fonksiyon için bir dağıtıcı tanımlamanız gerekip gerekmediğini belirleyin. Eğer gerekiyorsa, hangi dağıtıcının en uygun seçenek olduğunu analiz edin:

- `createInvoice` – Bir fatura hazırlar ve Retrofit kütüphanesi aracılığıyla SaaS API'sine gönderir. Retrofit kütüphanesi, askıya alınabilir (suspending) fonksiyonları desteklemektedir. Ayrıca, `toFakturowniaInvoiceData` fonksiyonu basit bir eşleme (mapping) işlemi gerçekleştirmektedir. Bu işlem, ağırlıklı olarak giriş/çıkış (I/O) tabanlı olduğu için `Dispatchers.IO` kullanılmalıdır.
- `sendEmail` – Bir e-posta hazırlar ve `SendGrid` sınıfının `api` metodunu kullanarak gönderir. Bu işlem, bloklayıcı (blocking) bir işlem olup bir `Response` nesnesi döndürmektedir. Bloklayıcı işlemler, ana iş parçacığında yürütülmemelidir. Dolayısıyla, iş parçacıklarının etkin yönetimi için `Dispatchers.IO` kullanılması önerilir.
- `getUserOrders` – MongoDB Kotlin sürücüsü kullanılarak veritabanından belirli bir kullanıcının siparişlerini getirir. `toList` fonksiyonu askıya alınabilir bir fonksiyon olup bir sipariş listesi döndürmektedir. Veritabanı işlemleri genellikle giriş/çıkış tabanlı olduğundan, işlemi engelleyici olmayan bir bağlamda gerçekleştirmek için `Dispatchers.IO` tercih edilmelidir.

- `upscaleImage` – TensorFlow kütüphanesi kullanılarak bir görüntüyü ölçeklendiren bir fonksiyondur. Bu tür işlemler yoğun işlemci kullanımı (CPU-bound) gerektirir ve büyük veri setleri üzerinde çalışabilir. En uygun dağıtıcı, çok iş parçacıklı işlemci havuzunu kullanan `Dispatchers.Default` olacaktır, çünkü bu dağıtıcı, işlemci çekirdeklerinin verimli bir şekilde kullanılmasını sağlar.

Her fonksiyonun doğasına uygun dağıtıcıyı seçmek, performansın optimize edilmesi ve engelleyici işlemlerin uygun bağlamlarda yürütülmesi açısından kritik öneme sahiptir.

```
override suspend fun createInvoice(
    invoiceData: InvoiceData,
) {
    invoiceApi.postInvoice(
        invoiceData.toFakturowniaInvoiceData()
    )
}

interface InvoiceApi {
    @POST("invoices.json")
    suspend fun postInvoice(
        @Body invoice: SendInvoiceData
    ): InvoiceCreationResponse
}

private val invoiceApi = retrofit2.Retrofit.Builder()
    .baseUrl(invoiceBaseUrl)
    .build()
    .create(InvoiceApi::class.java)

private val sendGrid = SendGrid(API_KEY)

suspend fun sendEmail(email: Email) {
    val request = Request().apply {
        method = Method.POST
        endpoint = "mail/send"
        body = email.toSendGridEmail()
    }
    sendGrid.api(request)
}

private val orderCollection = mongoClient
    .getDatabase("shop")
    .getCollection<Order>("orders")

suspend fun getUserOrders(userId: String): List<Order> =
```

```
    orderCollection
        .find(eq("userId", userId))
        .toList()

val model = TensorFlowModel()

suspend fun upscaleImage(image: Image): Image =
    model.upscale(image)
```

Alıştırma: DiscNewsRepository

`DiscNewsRepository`, diskte depolanan haberleri okumak için `DiscReader` sınıfını kullanan basit bir sınıftır. Ancak, diskten veri okuma işlemi bloklayıcı (blocking) bir işlemdir. Bu nedenle, mevcut `DiscNewsRepository` uygulaması askıya alınabilir fonksiyonlar içinde iş parçacıklarını bloke etmekte ve bu durum uygulamanın performans sorunları yaşamasına neden olmaktadır.

Bu problemi `getNews` fonksiyonunun yoğun bir şekilde kullanıldığını ve 200 paralel okuma işlemini desteklemesi gerektiğini varsayarak çözün.

```
class DiscNewsRepository(
    private val discReader: DiscReader
) : NewsRepository {
    override suspend fun getNews(newsId: String): News {
        val (title, content) = discReader.read("user/$newsId")
        return News(title, content)
    }
}
```

Başlangıç kodu ve birim testleri, GitHub üzerindeki MarcinMoskala/kotlin-exercises projesinde, `coroutines/dispatcher/DiscNewsRepository.kt` dosyasında bulunabilir. Bu projeyi klonlayarak alıştırmayı yerel ortamınızda çözebilirsiniz.

Alıştırma: Dağıtıcılarla Deneyler

Farklı dağıtıcıların seçimlerinin yürütme süresi üzerindeki etkisini gözlemlemek için bir deney yapın.
Bu deneyde, 100 bağımsız süreç başlatın ve her biri aşağıdaki işlemleri gerçekleştirsin:

- CPU yoğun işlem: `cpuHeavy` fonksiyonunu çalıştırın.
- Bloklayıcı işlem: `Thread.sleep(1000)` ile iş parçacığını 1 saniye boyunca durdurun.
- Askıya alınan işlem: `delay(1000)` kullanarak süreci 1 saniye beklemeye alın.

Bu deney, hangi dağıtıcının hangi tür görevler için daha uygun olduğunu analiz etmenize yardımcı olacaktır.

```
val dispatcher = Dispatchers.IO.limitedParallelism(1)
//val dispatcher = Dispatchers.Default
//val dispatcher = Dispatchers.IO
//val dispatcher = Dispatchers.IO.limitedParallelism(100)

val operation = ::cpu1
//val operation = ::blocking
//val operation = ::suspending

fun cpu1() {
    var i = Int.MAX_VALUE
    while (i > 0) i -= if (i % 2 == 0) 1 else 2
}

fun blocking() {
    Thread.sleep(1000)
}

suspend fun suspending() {
    delay(1000)
}
```

Dağıtıcıların Performans Testi

Aşağıdaki dağıtıcıları test edin:

- Tek iş parçacığı ile sınırlı dağıtıcı
- `Dispatchers.Default`
- `Dispatchers.IO`
- 100 iş parçacığı ile sınırlı dağıtıcı

Tüm kombinasyonları test edin ve aşağıdaki tabloyu elde edilen yürütme süreleri ile doldurun.Daha sonra, elde edilen sonuçları analiz ederek hangi dağıtıcının hangi tür görevler için daha verimli olduğunu açıklayın.

Dağıtıcı	CPU yoğun işlem	Bloklayıcı işlem	Askıya alma işlemi
Tek iş parçacığı			
Dispatchers.Default			
Dispatchers.IO			
100 iş parçacıklı dağıtıcı			

Yürütme süresini ölçmek için aşağıdaki kodu kullanabilirsiniz:

```
suspend fun main() = measureTimeMillis {
    coroutineScope {
        repeat(100) {
            launch(dispatcher) {
                operation()
                println("Done $it")
            }
        }
    }
}.let { println("Took $it") }
```

Başlangıç kodu ve örnek kullanım, GitHub üzerindeki MarcinMoskala/kotlin-exercises projesinde `coroutines/dispatcher/Experiments.kt` dosyasında bulunabilir. Bu projeyi klonlayarak alıştırmayı yerel ortamınızda çözebilirsiniz.

Job ve Coroutine Yaşam Döngüsü

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Job ve Coroutine İlişkileri

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Yaşam Döngüsü

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Job'un Tamamlanmasını Bekleme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Job Fabrika Fonksiyonu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutineleri Senkronize Etme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

İptal Mekanizması(Cancellation)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Temel İptal Mekanizması

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`finally` Bloğu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

invokeOnCompletion

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alt Coroutine'lerin İptali

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine Kapsamında İptal İşlemi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Sadece Bir Çağrı Daha

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Durdurulamaz Olanı Durdurmak

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

CancellationException'in Özel Durumu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

CancellationException Üst Coroutine'e Yayılmaz

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

withTimeout

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`suspendCancellableCoroutine`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: İptal Mekanizmasıyla İlgili Hataları Düzeltme

`updateUser` Fonksiyonu (Dört hata içerir, üçü iptalle ilgilidir):

İstisna Yönetimi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

İstisnalar ve Yapılandırılmış Eşzamanlılık(structured concurrency)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

SupervisorJob

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

SupervisorJob'ı coroutine oluşturucularına argüman olarak vermeyin

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

supervisorScope

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`withContext(SupervisorJob())` Kullanmayın

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

İstisnalar ve await Çağrısı

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`CoroutineExceptionHandler`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bir coroutine scope (iş kapsamı) oluşturma

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

CoroutineScope fabrika fonksiyonu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Arka plan kapsamı (background scope) oluşturma

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Android'de scope oluşturma

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: NotificationSender

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: BaseViewModel

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Değiştirilebilir duruma(mutable state) erişimi eşzamanlama (synchronizing)

Atomik değerlerin kullanımı (Using atomic values)

Volatile anotasyonu (Volatile annotation)

Eşzamanlı koleksiyonların kullanımı (Using concurrent collections)

Synchronized bloklar (Synchronized blocks)

Tek iş parçacığıyla sınırlı bir dispatcher kullanımı

Mutex

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Semaphore

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: CompanyDetailsRepository

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: CancellingRefresher

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: TokenRepository

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Suspended lazy

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: mapAsync ile eşzamanlılık sınırı (concurrency limit)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Kotlin Coroutines Testi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Zaman Bağımlılıklarının Test Edilmesi (Testing time dependencies)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`TestCoroutineScheduler` ve `StandardTestDispatcher`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

runTest

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Arka plan scope'u (Background scope)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

İptal (cancellation) ve bağlam aktarımının (context passing) test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`UnconfinedTestDispatcher`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Mock nesnelerin kullanımı (Using mocks)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Dispatcher değiştiren fonksiyonların test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Fonksiyon çalışırken gerçekleşen davranışların test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Yeni coroutine başlatan fonksiyonların test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

runTest içinde scope gerektiren sınıfların test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Ana dispatcher'ın (Main dispatcher) değiştirilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine başlatan Android fonksiyonlarının test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Test dispatcher'ı bir rule ile ayarlamak

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: UserDetailsRepository Sınıfını Test Etme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: mapAsync Fonksiyonunun Test Edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: NotificationSender Sınıfını Test Etme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Bir ViewModel Sınıfını Test Etme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bölüm 3: Channel ve Flow

Atölye çalışmalarımda, birçok katılımcı Flow hakkında bilgi edinmek için sabırsızlanıyor – ki bu yapı, coroutine'lerin tepkisel veri akışlarını (reactive streams) temsil eder.Sonunda o noktaya geldik: bu bölümde, hem Channel hem de Flow yapısını ele alacağız. Bu iki yapı da, bilinmeye değer yararlı araçlardır.

Konuya biraz daha temel bir kavram olan Channel ile başlayacağız, ardından Flow konusuna derinlemesine gireceğiz.

Channel

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Kanal Türleri (Channel Types)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Arabellek Taşması Durumunda (On Buffer Overflow)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Teslim Edilemeyen Öğeler İçin İşleyici (On Undelivered Element Handler)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Fan-out (Yayılma Modeli)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Fan-in (Toplanma Modeli)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Pipelines (Veri İşleme Hatları)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Pratik Kullanım (Practical Usage)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: UserRefresher

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Kafeterya simülasyonu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Select (Seçim)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Ertelenmiş (deferred) değerleri seçme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Channel'lardan Seçim Yapma (Selecting from channels)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: raceOf

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Sıcak (Hot) ve Soğuk (Cold) Veri Kaynakları

Sıcak ve Soğuk Akışlar

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Sıcak Channel, Soğuk Flow

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow'a Giriş

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow'un Diğer Değer Temsil Yöntemleriyle Karşılaştırılması

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow'un Özellikleri

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow Terimlendirmesi (Nomenclature)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Gerçek Hayattaki Kullanım Durumları

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`Flow`'u Anlamak

Kotlin Coroutines içindeki `Flow` kavramı, çoğu geliştiricinin sandığından çok daha basittir.Aslında yalnızca hangi işlemlerin yürütüleceğini tanımlayan bir yapıdır. Bir anlamda, askıya alınabilir (`suspending`) bir lambda ifadesine benzer, ancak bunun üzerine bazı ek unsurlar barındırır. Bu bölümde, `Flow` arayüzünü (`Flow` interface) ve `flow` yapıcısını (`builder`) nasıl uygulayabileceğinizi, bir lambda ifadesini adım adım dönüştürerek göstereceğim. Bu süreç size `Flow`'un nasıl çalıştığına dair derin bir kavrayış kazandıracaktır.Bu bölüm, kullandığı araçları gerçekten anlamak isteyen meraklı geliştiriciler için yazılmıştır. Eğer bu sizi tanımlamıyorsa, bu bölümü atlayabilirsiniz.Ancak okumaya karar verirseniz, umarım keyif alırsınız.

`Flow`'u Anlamak

Hikâyemize basit bir lambda ifadesiyle başlayacağız.Her lambda ifadesi, bir kez tanımlanıp, daha sonra birden fazla kez çağrılabilir.

```
fun main() {
    // Parametre almayan ve değer döndürmeyen (Unit) bir lambda fonksiyonu tanımlanıyor
    // () -> Unit = parametresiz fonksiyon tipi, Unit dönüş tipini belirtir
    val f: () -> Unit = {
        // Lambda gövdesi içinde sırayla harfler yazdırılıyor
        print("A")
        print("B")
        print("C")
    }

    // Lambda fonksiyonu ilk kez çağrılıyor
    f() // ABC yazdırır

    // Lambda fonksiyonu ikinci kez çağrılıyor
    f() // ABC yazdırır

    // Çıktı: ABCABC (boşluk olmadan)
    // Her çağrıda aynı işlemler tekrarlandığı için aynı çıktıyı verir
}
```

Konuyu biraz daha ilginç hale getirmek için, lambda ifademizi `suspend` yapalım ve içine bir miktar gecikme (`delay`) ekleyelim. Dikkat edilmesi gereken önemli bir nokta şudur: Bu tür bir askıya alınabilir (`suspending`) lambda ifadesi her çağrıldığında işlemler sıralı (sequential) olarak yürütülür.Yani, önceki çağrı tamamlanmadan yeni bir çağrı yapılmamalıdır.

```
import kotlinx.coroutines.delay

// Ana fonksiyon, içinde askıya alınabilir (suspending) kod kullanabilmek için suspend ol\
suspend fun main() {
    // Parametre almayan ve değer döndürmeyen (Unit) bir askıya alınabilir lambda fonksiy\
onu tanımlanıyor
 belirtir: suspend () -> Unit tipi, fonksiyonun suspending olduğunu ve askıya alınabileceğini\
    val f: suspend () -> Unit = {
        print("A")       // A harfini yazdır
        delay(1000)      // 1 saniye bekle (coroutine'i askıya alır, thread'i bloklamaz)
        print("B")       // B harfini yazdır
        delay(1000)      // 1 saniye daha bekle
        print("C")       // C harfini yazdır
    }

    f()  // Fonksiyonu ilk kez çağır ve tamamlanmasını bekle
    f()  // Fonksiyonu ikinci kez çağır ve tamamlanmasını bekle
}

// Çıktı sıralaması ve zamanlama:
// A
// (1 saniye bekler)
// B
// (1 saniye bekler)
// C
// A
// (1 saniye bekler)
// B
// (1 saniye bekler)
// C

// Not: delay fonksiyonu coroutine'i askıya alır ama thread'i bloklamaz.
// Bu sayede uzun süren işlemler sırasında sistem kaynakları verimli kullanılır.
```

A lambda expression might have a parameter that can represent a function. We will call this parameter `emit`. So, when you call the lambda expression `f`, you need to specify another lambda expression that will be used as `emit`.

Bir lambda ifadesi, bir fonksiyonu temsil edebilecek bir parametre alabilir.Bu parametreye `emit` adını vereceğiz. Dolayısıyla, `f` adlı lambda ifadesini çağırdığınızda, `emit` olarak kullanılacak başka bir lambda ifadesini de belirtmeniz gerekir.

```
suspend fun main() {
    val f: suspend ((String) -> Unit) -> Unit = { emit ->
        emit("A")
        emit("B")
        emit("C")
    }
    f { print(it) } // ABC
    f { print(it) } // ABC
}
```

Gerçekte, `emit` fonksiyonunun da askıya alınabilir (`suspending`) olması gerekir.Ancak bu durum, kullandığımız fonksiyon türünü oldukça karmaşık hale getiriyor. Bu karmaşıklığı azaltmak için, `emit` adında soyut bir metoda sahip olan `FlowCollector` adlı bir fonksiyonel arayüz (function interface) tanımlayacağız.Böylece fonksiyon türü yerine bu arayüzü kullanabiliriz. Buradaki püf nokta şudur:Fonksiyonel arayüzler, lambda ifadeleri ile tanımlanabilir.Bu sayede, `f` fonksiyonunun çağrılma biçimini değiştirmemize gerek kalmaz.

```
// Ana fonksiyon, içinde askıya alınabilir (suspending) kod kullanabilmek için suspend ol\
arak tanımlanmış
suspend fun main() {
    // Daha karmaşık bir lambda fonksiyonu tanımı:
    // - suspend keyword'ü ile askıya alınabilir bir fonksiyon
    // - ((String) -> Unit) -> Unit tipi şunu ifade eder:
    //   * Parametre olarak bir fonksiyon alır (String -> Unit tipinde)
    //   * Kendisi Unit döndürür (değer döndürmez)
    val f: suspend ((String) -> Unit) -> Unit = { emit ->
        // emit parametresi, kendisine verilen String'i işleyen bir fonksiyondur
        // emit fonksiyonunu "A" parametresiyle çağır
        emit("A")
        // emit fonksiyonunu "B" parametresiyle çağır
        emit("B")
        // emit fonksiyonunu "C" parametresiyle çağır
        emit("C")
    }

    // f fonksiyonunu çağırırken, emit parametresi olarak bir lambda veriyoruz
    // Bu lambda (it) parametresini (yani "A", "B", "C" string'lerini) yazdırır
    // ABC yazdırır (her string için print çağrılır)
    f { print(it) }

    // Aynı şekilde ikinci kez çağırıyoruz
    // ABC yazdırır
    f { print(it) }

    // Çıktı: ABCABC (boşluk olmadan)
}
```

`emit` fonksiyonunu `it.emit(...)` şeklinde çağırmak pratik değildir.Bunun yerine, `FlowCollector` arayüzünü bir alıcı (receiver) haline getireceğiz. Bu sayede, lambda ifadesi içinde `FlowCollector` türünden bir alıcı (`this`) bulunmuş olur.Böylece `this.emit(...)` veya doğrudan `emit(...)`şeklinde çağrıda bulunabiliriz. Önemli olan şu ki, bu değişiklik `f` fonksiyonunun çağrılma biçimini etkilemez – `f`'yi aynı şekilde çağırmaya devam ederiz.

```
import kotlin.*

// Bir functional interface tanımlanıyor:
// - FlowCollector adında bir interface
// - Bu interface, yalnızca bir abstract metot içeriyor, dolayısıyla functional interface\
 olarak işaretlenmiş
// - emit fonksiyonu askıya alınabilir (suspend) ve bir String parametresi alıyor
fun interface FlowCollector {
    suspend fun emit(value: String)
}

// Ana fonksiyon askıya alınabilir (suspend keyword'ü var)
suspend fun main() {
    // f adında bir suspending extension fonksiyon tanımlanıyor:
    // - FlowCollector tipinde bir alıcı (receiver) üzerinden kullanılabilir
    // - Unit döner, yani hiçbir değer döndürmez
    val f: suspend FlowCollector.() -> Unit = {
        // emit fonksiyonu çağrılarak değerler yayımlanır (FlowCollector üzerinden)
        emit("A") // "A" değeri yayımlanır
        emit("B") // "B" değeri yayımlanır
        emit("C") // "C" değeri yayımlanır
    }

    // f fonksiyonunun ilk çağrısı yapılıyor:
    // - FlowCollector bir lambda ile implement ediliyor
    // - emit fonksiyon çağrısında alınan değer, it parametresi olarak bu lambda içinde k\
ullanılıyor
    f { print(it) } // "it" parametresiyle "A", "B", "C" yazdırılır (çıkış: ABC)

    // f fonksiyonunun ikinci çağrısı yapılıyor:
    // - Aynı şekilde FlowCollector bir lambda ile implement ediliyor
    // - Yine "A", "B", "C" değerleri yazdırılır
    f { print(it) } // Çıkış: ABC

    // Toplam çıktı: ABCABC (boşluk olmadan yazdırılmış)
}
```

Lambda ifadelerini doğrudan taşımak yerine, bir arayüzü (interface) uygulayan bir nesneye sahip olmayı tercih ederiz. Bu arayüze `Flow` adını vereceğiz ve tanımımızı bir nesne ifadesi (object expression) ile sarmalayacağız.

```kotlin
// FlowCollector adında bir functional interface tanımlanıyor:
// - Bir tane abstract metot (emit) içeriyor
// - emit, askıya alınabilir bir fonksiyon ve bir String parametresi alıyor
fun interface FlowCollector {
    suspend fun emit(value: String)
}

// Flow adında bir interface tanımlanıyor:
// - collect adında, askıya alınabilir (suspend) bir metot içeriyor
// - collect metodu, bir FlowCollector parametresi kabul ediyor
interface Flow {
    suspend fun collect(collector: FlowCollector)
}

// Ana programın başladığı yer
// - Ana fonksiyon askıya alınabilir işlemler içeriyor (suspend ile işaretlenmiş)
suspend fun main() {
    // builder adında bir suspending extension fonksiyon tanımlanıyor:
    // - FlowCollector türünde bir alıcı (receiver) üzerinde çalışıyor
    // - emit çağrılarıyla değerler yayımlanıyor (FlowCollector'dan)
    val builder: suspend FlowCollector.() -> Unit = {
        emit("A") // "A" değeri yayımlanır
        emit("B") // "B" değeri yayımlanır
        emit("C") // "C" değeri yayımlanır
    }

    // flow adında bir Flow nesnesi oluşturuluyor:
    // - collect metodunu override eden bir anonim sınıf kullanılıyor
    val flow: Flow = object : Flow {
        override suspend fun collect(
            collector: FlowCollector // collect metodu bir FlowCollector bekliyor
        ) {
            // FlowCollector içinde builder fonksiyonu çağrılıyor
            // - Burası FlowCollector tarafından yayımlanacak değerleri tanımlar
            collector.builder()
        }
    }

    // İlk collect çağrısı yapılıyor:
    // - Flow.collect çağrılır ve emit edilen değerler yazdırılır
    flow.collect { print(it) } // Çıktı: ABC (emit edilen değerler sırayla yazdırılır)

    // İkinci collect çağrısı yapılıyor:
    // - Aynı işlem tekrarlanır, böylece tekrar "ABC" yazdırılır
    flow.collect { print(it) } // Çıktı: ABC

    // Toplam çıktı: ABCABC (boşluk olmadan yazdırılmıştır)
}
}
```

Son olarak, `flow` oluşturucu (builder) fonksiyonunu tanımlayarak `Flow` nesnesi oluşturmayı daha da sadeleştirelim.

```
import kotlin.*

// FlowCollector adında bir functional interface tanımlanıyor:
// - Tek bir abstract metot (emit) içeriyor
// - emit askıya alınabilir (suspend) bir fonksiyon ve bir String parametresi alıyor
fun interface FlowCollector {
    suspend fun emit(value: String)
}

// Flow adında bir interface tanımlanıyor:
// - collect adında, askıya alınabilir (suspend) bir metot içeriyor
// - collect metodu, bir FlowCollector parametresi kabul ediyor
interface Flow {
    suspend fun collect(collector: FlowCollector)
}

// flow adında bir fonksiyon tanımlanıyor:
// - Bu fonksiyon, bir FlowCollector üzerinde çalışacak bir builder alır
// - Yeni bir Flow nesnesi döndürür (anonim sınıf ile oluşturulmuş)
fun flow(
    builder: suspend FlowCollector.() -> Unit // FlowCollector için bir builder
) = object : Flow {
    // Flow arayüzünün collect metodunu override eder
    override suspend fun collect(collector: FlowCollector) {
        // FlowCollector üzerine builder çağrılır
        collector.builder()
    }
}

// Ana programın başladığı yer
suspend fun main() {
    // f adında bir Flow nesnesi oluşturuluyor:
    // - flow fonksiyonu, builder parametresi ile çağrılıyor
    val f: Flow = flow {
        emit("A") // Builder içinde "A" değeri yayımlanır
        emit("B") // "B" değeri yayımlanır
        emit("C") // "C" değeri yayımlanır
    }

    // İlk collect çağrısı yapılıyor:
    // - Flow.collect çağrılır ve emit edilen değerler yazdırılır
    f.collect { print(it) } // Çıktı: ABC (emit edilen değerler sırayla yazdırılır)

    // İkinci collect çağrısı yapılıyor:
    // - Aynı işlem tekrarlanır, böylece tekrar "ABC" yazdırılır
    f.collect { print(it) } // Çıktı: ABC
```

```
47
48      // Toplam çıktı: ABCABC (boşluk olmadan yazdırılmıştır)
49  }
```

Yapmamız gereken son değişiklik, `String` türünü genel bir tür parametresiyle (generic type parameter) değiştirmektir.
Bu sayede, her türden değerin yayımlanmasına (emit) ve toplanmasına (collect) olanak tanımış oluruz.

```
1   // FlowCollector adında bir fun interface tanımlanır
2   // T türünde bir parametre alır ve emit adında askıya alınabilir (suspend) bir metod içer\
ir
3   fun interface FlowCollector<T> {
4       suspend fun emit(value: T)
5   }
6
7   // Flow adında bir generic interface tanımlanır
8   // T türünde bir parametre alır ve collect adında bir metod içerir
9   interface Flow<T> {
10      suspend fun collect(collector: FlowCollector<T>)
11  }
12
13  // flow adında bir generic fonksiyon tanımlanır
14  // T türünde bir parametre kabul eder ve bir Flow<T> nesnesi döner
15  fun <T> flow(
16      builder: suspend FlowCollector<T>.() -> Unit // FlowCollector<T>'ye uygulanabilecek b\
ir builder
17  ) = object : Flow<T> { // Flow<T> arayüzünü uygulayan bir nesne oluşturulur
18      override suspend fun collect(
19          collector: FlowCollector<T> // collect metodu FlowCollector<T> nesnesi alır
20      ) {
21          collector.builder() // Gelen collector üzerinde builder işlemi yapılır
22      }
23  }
24
25  // Programın ana fonksiyonu
26  suspend fun main() {
27      // Bir Flow<String> nesnesi oluşturulur
28      // Emit metodu kullanılarak A, B, C değerleri yayımlanır
29      val f: Flow<String> = flow {
30          emit("A")
31          emit("B")
32          emit("C")
33      }
34      // İlk collect çağrısı yapılır; yayımlanan değerler yazdırılır
35      f.collect { print(it) } // ABC
36      // İkinci collect çağrısı yapılır; aynı değerler tekrar yayımlanır
37      f.collect { print(it) } // ABC
38  }
```

İşte bu kadar! Aslında `Flow`, `FlowCollector` ve `flow` yapısı neredeyse tam olarak bu şekilde uygulanmıştır.`collect` fonksiyonunu çağırdığınızda, `flow` builder içinde tanımlanan lambda ifadesi çalıştırılır.Bu ifade içinde `emit` çağrıldığında ise, bu çağrı `collect` sırasında belirtilen lambda ifadesini çalıştırır.`Flow` yapısının çalışma mantığı tam olarak budur. Burada sunduğumuz builder, bir `flow` oluşturmanın en temel yoludur.İlerleyen bölümlerde farklı builder'ları da öğreneceğiz, ancak bunlar da genellikle arka planda `flow` yapısını kullanırlar.

```
public fun <T> Iterator<T>.asFlow(): Flow<T> = flow {
    // Iterator<T> için asFlow fonksiyonu tanımlaması
t edilir).
    // Bu fonksiyon bir Flow<T> döner. Yani, Iterator'daki her bir eleman yayımlanır (emi\
    forEach { value ->
        // Iterator'un her bir elemanında forEach bloğu çalışır.
        // forEach ile her bir eleman `value` değişkenine atanır ve işlem yapılır.

        emit(value)
        // Her eleman emit metodu ile FlowCollector'a gönderilir ve akışa yayımlanır.
    }
    // Sonuç olarak, Iterator'u bir akışa (Flow) dönüştürür.
}

public fun <T> Sequence<T>.asFlow(): Flow<T> = flow {
    // Sequence<T> için asFlow fonksiyonu tanımlaması
gulayan bir veri yapısıdır.
    // Sequence'i bir Flow<T>'ye dönüştürmek için kullanılır. Sequence, lazy evaluation uy\
    forEach { value ->
        // Sequence'deki her bir eleman forEach bloğu içinden geçer.
        // Sequence elemanları birer birer `value` değişkenine atanır.

        emit(value)
        // Emit metodu, mevcut elemanı akışa ekler.
    }
    // Sonuç olarak, bir Sequence'i bir Flow olarak kullanma imkanı sağlanır.
}

public fun <T> flowOf(vararg elements: T): Flow<T> = flow {
    // flowOf adında bir fonksiyon tanımlaması
    // Bir vararg parametre alır ve bu parametreleri bir Flow<T> olarak döner.

    for (element in elements) {
        // Parametrelerden gelen her element üzerinde işlem yapılır.
        // for döngüsü ile vararg içindeki tüm elemanlara tek tek erişilir.

        emit(element)
        // Her bir eleman, akışın bir parçası olarak yayımlanır (emit edilir).
    }
    // Bu, değişken sayıda argümanın bir Flow<T>'ye dönüştürülmesini mümkün kılar.
}
```

Flow İşlemenin (Processing) Nasıl Çalıştığı

Flow, alıcı (receiver) ile tanımlanmış askıya alınabilir (suspending) lambda ifadelerine kıyasla biraz daha karmaşık olarak düşünülebilir.Ancak asıl gücü, oluşturma, işleme ve gözlemleme işlemleri için tanımlanmış olan tüm bu fonksiyonlarda yatar. Aslında bu fonksiyonların çoğu, perde arkasında oldukça basit çalışır.İlerleyen bölümlerde bunları detaylı olarak inceleyeceğiz, ancak şimdilik şu sezgiyi edinmeni istiyorum:Bu fonksiyonların büyük çoğunluğu flow, collect ve emit kullanılarak kolayca oluşturulabilir.

Örneğin, bir flow içerisindeki her öğeyi dönüştüren map fonksiyonunu ele alalım: Bu fonksiyon yeni bir flow oluşturur, dolayısıyla flow builder'ını kullanır.Oluşturulan flow çalıştırıldığında, sarmaladığı (wrap ettiği) asıl flow da başlatılmalıdır.Bu nedenle map, builder bloğu içinde collect çağrısı yapar. Her bir öğe alındığında (collect içinde), map bu öğeyi dönüştürür ve ardından yeni flow'a emit ile aktarır.

```
// Flow'un map fonksiyonunun tanımlaması.
fun <T, R> Flow<T>.map(
    // T tipindeki bir elemanı R tipine dönüştürmek için kullanılacak dönüşüm fonksiyonu.
    transformation: suspend (T) -> R
): Flow<R> = flow {
    // Bir Flow<R> başlatılır.
    collect {
        // Orijinal Flow'dan elemanlar toplanır.
        emit(transformation(it))
        // Eleman `transformation` fonksiyonuyla işlenir ve yeni Flow'da yayımlanır (emit\
 edilir).
    }
}

suspend fun main() {
    // flowOf fonksiyonu kullanılarak bir Flow<String> oluşturulur.
    flowOf("A", "B", "C")
        .map {
            // Her bir elemanı işlerken 1 saniyelik gecikme eklenir.
            delay(1000)
            // Eleman küçük harfe dönüştürülür.
            it.lowercase()
        }
        .collect {
            // Sonuçlar toplanır ve ekrana yazdırılır.
            println(it)
        }
    // (1 saniye) a
```

```
    // (1 saniye) b
    // (1 saniye) c
}
```

Sonraki bölümlerde öğreneceğimiz yöntemlerin (method) çoğunun davranışı da bu kadar basittir. Bunu anlamak oldukça önemlidir; çünkü bu sayede yalnızca yazdığımız kodun nasıl çalıştığını daha iyi kavramakla kalmayız, aynı zamanda benzer işlevleri nasıl yazabileceğimizi de öğrenmiş oluruz.

```
fun <T> Flow<T>.filter(
    predicate: suspend (T) -> Boolean
): Flow<T> = flow {
    // `filter` fonksiyonu, bir akışın (Flow) içindeki elemanları bir koşula (predicate) \
göre süzmek için kullanılır.
    // Koşulu sağlayan elemanlar yeni bir akışta kullanıma sunulur.
    collect {
        // Orijinal Flow'daki elemanlar sırayla toplanır.
        if (predicate(it)) {
            // Eğer eleman koşulu sağlıyorsa...
            emit(it)
            // Eleman yayımlanır (emit edilir).
        }
    }
}

fun <T> Flow<T>.onEach(
    action: suspend (T) -> Unit
): Flow<T> = flow {
    // `onEach` fonksiyonu, bir akışın her elemanı üzerinde bir yan etki oluşturmak için \
kullanılır.
    // Elemanlar üzerinde belirtilen işlem (action) yapılır, ardından orijinal elemanlar \
yayımlanmaya devam eder.
    collect {
        // Orijinal Flow'daki elemanlar sırayla toplanır.
        action(it)
        // Her eleman için verilen işlem (action) gerçekleştirilir.
        emit(it)
        // İşlem tamamlandıktan sonra eleman tekrar yayımlanır (emit edilir).
    }
}

fun <T> Flow<T>.onStart(
    action: suspend () -> Unit
): Flow<T> = flow {
    // `onStart` fonksiyonu, bir akış başlatılmadan önce bir işlem (action) gerçekleştirm\
```

```
ek için kullanılır.
    // Bu işlem, akış içindeki elemanlar toplanmadan önce bir defa çalıştırılır.
    action()
    // Akışın başında belirlenen işlem (action) gerçekleştirilir.
    collect {
        // Orijinal Flow'daki elemanlar sırayla toplanır.
        emit(it)
        // Toplanan elemanlar yayımlanır (emit edilir).
    }
}
```

Flow Doğası Gereği Eşzamanlıdır (Synchronous)

`Flow`, tıpkı askıya alınabilir (`suspending`) fonksiyonlar gibi doğası gereği eşzamanlı (synchronous) çalışır. Yani `collect` çağrısı, akış (`flow`) tamamlanana kadar askıya alınmış şekilde bekler. Bu durum, `Flow`'un kendisinin yeni bir coroutine başlatmadığı anlamına gelir.Akış içindeki belirli adımlar yeni coroutine'ler başlatabilir (tıpkı `suspend` fonksiyonlarda olduğu gibi),ancak bu varsayılan bir davranış değildir. Çoğu `Flow` işleme adımı (örneğin `map`, `filter`, `onEach`) eşzamanlı şekilde çalışır.Bu nedenle, `onEach` bloğu içinde bir `delay` çağrısı yapılırsa, bu gecikme **her öğe arasında** gerçekleşir —tüm öğelerden önce değil.

```
suspend fun main() {
    // flowOf, bir Flow oluşturur ve sırasıyla "A", "B", "C" elemanlarını içerir.
    flowOf("A", "B", "C")
        // onEach, her eleman işlenmeden önce bir işlem yapar. Burada her eleman için\
 1 saniye bekleme (delay) eklenmiş.
        // collect, akıştaki yayımlanan elemanları toplar. Her elemanı alır ve println il\
e ekrana yazdırır.
        .collect { println(it) }
}
// (1 sec)
// A
// (1 sec)
// B
// (1 sec)
// C
```

Bu varsayılan davranış, `buffer` veya `conflate` gibi fonksiyonlarla değiştirilebilir.Bu fonksiyonlar, akışın yukarısındaki işlemler için yeni bir coroutine başlatır.Benzer şekilde, `channelFlow` gibi coroutine oluşturucular (builder) da gövdesi için yeni bir coroutine başlatır.Ancak unutulmamalıdır ki, varsayılan davranış eşzamanlıdır (synchronous).

* * *

Flow ve Paylaşılan Durum (Shared State)

Daha karmaşık akış (flow) işleme algoritmaları geliştirirken, değişkenlere erişimin ne zaman senkronize edilmesi gerektiğini bilmek önemlidir.Bu bölümde en önemli kullanım durumlarını inceleyeceğiz.

Özelleştirilmiş (custom) `Flow` işleme fonksiyonları geliştirirken, akış içerisinde senkronizasyon mekanizması kullanmadan değiştirilebilir (mutable) durumlar tanımlayabilirsiniz.Çünkü `Flow` adımları doğası gereği eşzamanlı (synchronous) olarak çalışır.

```
fun <T, K> Flow<T>.distinctBy(
    // Bu parametre, her elemandan benzersiz bir anahtar oluşturan bir fonksiyondur.
    keySelector: (T) -> K
    // Örneğin, bir veri sınıfında belirli bir alanı seçmek için kullanılabilir.
) = flow {
    // Yeni bir Flow oluşturuyoruz. Bu Flow, verilen orijinal akışın (Flow) elemanlarını \
filtreleyerek
    // benzersiz anahtarlara sahip olan elemanları yayımlayacak.

    val sentKeys = mutableSetOf<K>()
    // Benzersiz anahtarları saklamak için bir "Set" (küme) oluşturuyoruz. Sets, aynı ana\
htarı birden fazla kez saklamaz.
    // Burada daha önce işlenen anahtarlar depolanır.

    collect { value ->
        // Orijinal Flow'a ait tüm elemanlar sırayla alınır. Akışın elemanları, "value" d\
eğişkenine atanır.

        val key = keySelector(value)
        // Bu elemandan benzersiz bir anahtar elde etmek için "keySelector" fonksiyonu ça\
ğrılır.
        // Örneğin, bir nesne içindeki "id" alanını seçebilir.

        if (key !in sentKeys) {
            // Eğer bu anahtar daha önce işlenmemişse, yani Set'te bulunmuyorsa...
            sentKeys.add(key)
            // Anahtarı Set'e ekle (işlenmiş anahtarlar arasına ekle).

            emit(value)
            // Anahtar benzersiz olduğundan, bu eleman yayılır (emit edilir). Akışın tüke\
ticisi bu değeri alabilir.
        }
        // Eğer anahtar daha önce işlenmişse, "emit" işlemi yapılmaz ve eleman atlanır.
    }
    // Bu işlem, akıştaki her bir eleman için tekrarlanır; böylece yalnızca benzersiz ana\
htarlarla eşleşen elemanlar yayımlanır.
```

```
}
```

Aşağıda, bir `flow` adımı içinde kullanılan ve tutarlı sonuçlar üreten bir örnek yer almaktadır:
`counter` değişkeni her zaman doğru şekilde 1000'e kadar artırılır.

```
fun Flow<*>.counter() = flow<Int> {
    // Bu bir Extension Function'dır ve herhangi bir Flow türüne uygulanabilir.
    // Akıştaki her elemanı saymak için bir Flow oluşturur ve akışın her elemanına karşılık gelen sayacı yayımlar.

    var counter = 0
    // Elemanları saymak için bir sayaç oluşturulur, başlangıç değeri 0'dır.

    collect {
        // Flow'un her bir elemanı sırasıyla işlenir.
        counter++
        // Sayaç her elemanda 1 artırılır.

        // Sadece işlem yükü eklemek için boş bir işlem yapılır:
        List(100) { Random.nextLong() }.shuffled().sorted()
        // 100 rastgele uzun değer oluşturulur, karıştırılır ve sıralanır.
        // Bu işlem, akışı yapay olarak yavaşlatmak için eklenmiş.

        emit(counter)
        // Mevcut sayaç değeri yayımlanır (emit edilir).
        // Akışın tüketicisi, bu sayaç değerini alır.
    }
}

suspend fun main(): Unit = coroutineScope {
    // coroutineScope: Eş zamanlı iş parçacıkları (coroutine'ler) çalıştırır ve hepsi tamamlanana kadar bekler.

    val f1 = List(1000) { "$it" }.asFlow()
    // "f1", 0'dan 999'a kadar olan sayıları içeren bir Flow oluşturur.

    val f2 = List(1000) { "$it" }.asFlow()
        .counter()
    // "f2", 0'dan 999'a kadar olan sayıları içeren bir Flow oluşturur ve ardından "counter" fonksiyonuna uygulanır.
    // Bu akış, sayaç değerlerini yayımlar.

    launch { println(f1.counter().last()) }
    // Paralel bir coroutine başlatır.
    // "f1.counter()" çağrılır, her eleman sayılıp yayımlanır, sonra son elemandaki sayaç değeri (1000) ekrana yazdırılır.
```

```
38
39      launch { println(f1.counter().last()) }
40  ır. // Yine 0-999 elemanlarının "counter()" ile sayılıp, son sayaç değeri (1000) yazdırıl\
41
42      launch { println(f2.last()) }
43      // "f2" akışı doğrudan kullanılır. Akış zaten "counter" uygulanmış şekilde, bu nedenl\
44  e son elemanın yayımladığı değer (1000) yazdırılır.
45      launch { println(f2.last()) }
46      // Yukarıdaki ile aynı işlem yeniden yapılır ve yine 1000 değeri ekrana yazdırılır.
47  }
```

Akış (`flow`) adımı içindeki bir değişkenin dışarıya çıkarılıp bir fonksiyon içine alınması yaygın bir hatadır.Bu tür bir değişken, aynı `flow` üzerinden veri toplayan (collect eden) tüm coroutine'ler arasında paylaşılan hâle gelir. Bu durumda, değişkenin kullanımı için senkronizasyon (synchronization) gereklidir ve bu durum **akışa özgüdür (flow-specific), ancak veri toplama işlemlerine özgü (flow-collection-specific) değildir.** Bu nedenle, `f2.last()` ifadesi yaklaşık 2000 döndürür, 1000 değil – çünkü bu değer, aynı `flow`'un iki paralel yürütmesindüretilen öğelerin toplam sayısını yansıtır.

```
1   fun Flow<*>.counter(): Flow<Int> {
2       // Flow'un elemanlarını sayan bir fonksiyon.
3       var counter = 0
4       // Sayaç, her eleman işlendiğinde bir artırılır ve bu değeri yayımlar.
5
6       return this.map {
7           // Akışı, her eleman üzerinde işlem yaparak dönüştürüyoruz.
8           counter++
9           // Sayaç her eleman işlendiğinde bir artırılır.
10
11          // Akışı yapay olarak yavaşlatmak için eklenmiş bir boş işlem:
12          List(100) { Random.nextLong() }.shuffled().sorted()
13
14          counter
15          // Her elemanda sayaç değeri döndürülür.
16      }
17  }
18
19  suspend fun main(): Unit = coroutineScope {
20      // coroutineScope içinde ayrı coroutine'ler çalıştıracağız.
21
22      val f1 = List(1_000) { "$it" }.asFlow()
23      // 0'dan 999'a kadar olan sayıları içeren bir Flow oluşturuyoruz.
24
25      val f2 = List(1_000) { "$it" }.asFlow()
26          .counter()
27      // f2 akışı, sayaç fonksiyonuna uygulanmış halde. Her eleman işlendiğinde, sayaç değe\
```

```
ri yayımlanacak.
28
29      // Paralel olarak dört farklı coroutine çalıştırıyoruz:
30
31      launch { println(f1.counter().last()) }
32      // f1 üzerinden çalıştırdığımız sayaç, 0'dan 999'a kadar elemanları işler ve sayaç de\
ğeri 1000 olarak sonlanır.
33
34      launch { println(f1.counter().last()) }
35      // Yukarıdaki aynı işlem yeniden yapılır, yine sayaç değeri 1000 olarak yazdırılır.
36
37      launch { println(f2.last()) }
38      // Burada f2 zaten bir sayaca bağlanmıştı. Çakışan coroutine'ler nedeniyle sayaç değe\
rini
39      // artıran işlemler birbirine karışabilir ve bu yüzden elde edilen değer 2000'den küç\
ük olabilir.
40
41      launch { println(f2.last()) }
42      // Yukarıdaki ile aynı işlem yeniden yapılır ve yine "2000'den küçük" bir değer elde \
edilir.
43  }
```

Son olarak, nasıl ki aynı değişkenleri kullanan askıya alınabilir (`suspending`) fonksiyonlar senkronizasyon (synchronization) gerektiriyorsa, bir `Flow` içinde kullanılan bir değişken de – eğer bir fonksiyonun dışında, bir sınıfın içinde ya da üst düzeyde (top-level) tanımlanmışsa – senkronizasyon gerektirir.

```
1   var counter = 0
2   // Tüm kod boyunca paylaşılan global bir sayaç. Her coroutine, bu değişkeni paylaşır ve i\
şlem sırasında birbirini etkileyebilir.
3
4   fun Flow<*>.counter(): Flow<Int> = this.map {
5       // Flow üzerindeki her eleman için işlemler gerçekleştiriyoruz:
6
7       counter++
8       // Global "counter" değişkenini her elemanda bir artırıyoruz.
9
10      // İşlemi yavaşlatmak ve yapay yük eklemek için sade bir görev yapıyoruz:
11      List(100) { Random.nextLong() }.shuffled().sorted()
12
13      counter
14      // Arttırılmış olan sayaç değeri dönüş çıktısı olarak sağlanır.
15  }
16
17  suspend fun main(): Unit = coroutineScope {
18      // coroutineScope ile eş zamanlı olarak birden fazla coroutine başlatıyoruz.
19
20      val f1 = List(1_000) { "$it" }.asFlow()
21      // 0'dan 999'a kadar olan sayıları içeren bir akış (Flow).
22
23      val f2 = List(1_000) { "$it" }.asFlow()
24          .counter()
25      // f2 akışına önceden "counter" işlemi uygulanmış, her eleman üzerinden geçerken coun\
```

```
ter artırılır.
26
27      // Aşağıda akış işleme için dört coroutine (iş parçacığı) başlatıyoruz:
28
29      launch { println(f1.counter().last()) }
30      // `f1.counter()` her çağrıldığında global "counter" ilk halinden itibaren işlem yapa\
r.
31      // Sayaç değeri, 1000 eleman işlendiğinde yaklaşık 1000 artar. Ancak, eş zamanlı çalı\
şan diğer coroutine'ler nedeniyle daha fazla artmış olabilir.
32
33      launch { println(f1.counter().last()) }
34      // Yukarıdakiyle aynı iş yapılır. Ancak, yine global sayaç değeri diğer coroutine'ler\
 nedeniyle etkilenebilir.
35
36      launch { println(f2.last()) }
37      // f2 üzerinde zaten `counter()` fonksiyonu çağrılmış durumdaydı. Elemanlar işlendikç\
e "counter" artırılacaktır.
38      // Ancak global sayaç eş zamanlı çalışan diğer coroutine'ler nedeniyle zaten yüksek b\
ir değere ulaşmış olabilir.
39
40      launch { println(f2.last()) }
41      // f2'nin aynı işlemi tekrarlanır. Diğer coroutine'ler global sayacı artırdığı için s\
onuç "4000'den küçük" olacak şekilde etkilenmiş olur.
42  }
```

Sonuç

`Flow`, alıcı (receiver) ile tanımlanmış askıya alınabilir (`suspending`) bir lambda ifadesine kıyasla biraz daha karmaşık olarak görülebilir.Ancak `Flow` üzerinde uygulanan işleme (processing) fonksiyonları, temelde yalnızca ona yeni işlemler ekleyerek onu süsleyen yapılardır (decorator). Burada herhangi bir sihir (magic) yoktur:`Flow`'un ve çoğu `Flow` yönteminin nasıl tanımlandığı aslında oldukça basit ve doğrudandır.

Flow Oluşturma (Flow Building)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Ham Değerlerden Flow Oluşturma (Flow from raw values)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Dönüştürücüler (Converters)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bir Fonksiyonu Flow'a Dönüştürme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow ve Reactive Streams

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow Oluşturucular (Flow Builders)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow Oluşturucuyu Anlamak (Understanding flow builder)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

channelFlow

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

callbackFlow

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Flow Yardımcıları (Flow utils)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Tüm Kullanıcıları Veren Flow (All users flow)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: distinct

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow Yaşam Döngüsü Fonksiyonları (Flow lifecycle functions)

onEach

onStart

onCompletion

onEmpty

catch

Yakalanmamış İstisnalar (Uncaught exceptions)

`retry`

flowOn

launchIn

Özet

Alıştırma: TemperatureService

Alıştırma: NewsViewModel

Akış (Flow) İşleme

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`map`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`filter`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`take` and `drop`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Koleksiyon İşleme (Collection Processing) Nasıl Çalışır?

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`merge`, `zip` ve `combine`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`fold` ve `scan`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`flatMapConcat`, `flatMapMerge` ve `flatMapLatest`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Değişmeyene Kadar Ayırt Etme (distinctUntilChanged)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`buffer` ve `conflate`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`debounce`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`sample`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Terminal işlemler (Terminal operations)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: ProductService

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Flow Kata

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: MessageService

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

SharedFlow ve StateFlow

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

SharedFlow

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`shareIn`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

StateFlow

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`stateIn`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: ProductService'i Güncelleyin

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: TemperatureService'i Güncelleyin

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: LocationService

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: PriceService

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: stateIn kullanarak NewsViewModel

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow Yapılarının Test Edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Dönüştürücü (Transformation) Fonksiyonlar

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Sonsuz Flow'ların Test Edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Kaç bağlantının açıldığını belirlemek

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

ViewModel'ların Test Edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Alıştırma: Flow Testi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Bölüm 4: Uygulamada Kotlin Coroutines

Bu noktada, coroutine'leri gerçek hayat projelerinde kullanmak için gerekli tüm parçaları öğrenmiş durumdasınız. Önceki bölümler, askıya alınabilir fonksiyonların (suspending functions) kullanımından flow işlemlerine kadar tüm temel öğeleri kapsamaktadır. Ayrıca bazı kullanım senaryolarını ve yaygın hataları da inceledik.

Bu bölümde, önceki konuları gözden geçirerek Kotlin Coroutines'in gerçek projelerde nasıl kullanılacağını ve nasıl * *kullanılmaması** gerektiğini daha derinlemesine ele alacağız. En yaygın kullanım örneklerini ve en iyi uygulama (best practice) yaklaşımlarını ele alacağız.

Bu bölüm, önceki bölümler üzerine inşa edilmiştir ve onlara sıkça atıfta bulunacaktır. Bu nedenle, bu bölümü hem bir özet hem de başvurulacak temel bir rehber olarak değerlendirebilirsiniz.

Yaygın Kullanım Senaryoları

Veri/Adaptör Katmanı(Data/Adapters Layer)

Callback functions (Geri çağırma fonksiyonları)

Blocking functions (Engelleyici fonksiyonlar)

Flow ile Gözlemleme

Alan Katmanı(Domain Layer)

Concurrent calls (Eşzamanlı çağrılar)

Flow dönüşümleri

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Sunum/API/UI katmanı (Presentation/API/UI layer)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Creating custom scope (Özel kapsam oluşturma)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

runBlocking kullanımı

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Working with Flow (Flow ile çalışmak)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Diğer Dillerden Coroutine Kullanımı

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Farklı Platformlarda Thread Kullanımı

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınan(suspending) Fonksiyonları Askıya Alınmayan(non-suspending) Fonksiyonlara Dönüştürmek

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınan Fonksiyonları Engelleyen (Blocking) Fonksiyonlara Dönüştürmek

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınan (Suspend) Fonksiyonları Geri Çağırma (Callback) Fonksiyonlarına Dönüştürmek

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Platforma Özgü Seçenekler

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Diğer Dillerden Askıya Alınan (Suspending) Fonksiyonları Çağırmak

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Flow ve Tepkisel-Reaktif Akışlar (Reactive Streams)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Özet

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Coroutine'lerin Başlatılması ile Askıya Alınan Fonksiyonların Karşılaştırılması

En İyi Uygulamalar

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`async` ile Tanımlanıp Hemen `await` Edilen Yapılardan Kaçının

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`awaitAll` Kullanın

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınan Fonksiyonlar Her İş Parçacığından Güvenle Çağrılabilir Olmalıdır

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`Dispatchers.Main` Yerine `Dispatchers.Main.immediate` Kullanın

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Yoğun İşlemler İçeren Fonksiyonlarda `yield` Kullanın

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Askıya Alınan Fonksiyonlar Çocuklarının Tamamlanmasını Bekler

Job öğesinin miras alınmadığını, yalnızca üst (parent) olarak kullanıldığını anlayın

Yapısal eşzamanlılığı (structured concurrency) bozmayın

CoroutineScope oluştururken SupervisorJob kullanın

Scope içindeki alt coroutine'leri iptal etmeyi düşünün

Bir scope kullanmadan önce, hangi koşullarda iptal edildiğini göz önünde bulundurun

GlobalScope kullanmayın

`Job` oluşturucusunun kullanımından, yalnızca bir scope inşa ederken faydalanın

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

`Flow` döndüren fonksiyonlar askıya alınabilir (suspending) olmamalıdır

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Yalnızca tek bir değer bekliyorsanız, `Flow` yerine askıya alınabilir (`suspending`) bir fonksiyonu tercih edin

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Sonuç

Her güzel şeyin bir sonu vardır; ne yazık ki bu kitap için de aynı durum geçerli. Elbette Kotlin Coroutines hakkında söylenecek daha çok şey var, ancak bu kitapta temel konuları kapsamlı ve sağlam bir şekilde ele aldığımızı düşünüyorum. Artık sıra sizde: Bu bilgileri uygulamaya dökerek deneyiminizi derinleştirme zamanı geldi.

Alıştırma çözümleri

Çözüm: Faktöriyel dizisi (factorial sequence)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Asal sayılar dizisi (prime numbers sequence)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Geri Çağırma Fonksiyonu Sarmalayıcıları (Callback Function Wrappers)

Çözüm: Devam Nesnesi Saklama (Continuation Storage)

Çözüm: Bir Devam Nesnesi (Continuation) Ne Saklar?

Çözüm: UserDetailsRepository

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: BestStudentUseCase

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: CommentService

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: mapAsync

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Bağlam (Context) Aktarımını Anlamak

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: CounterContext

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Dispatcher Kullanımı (Using Dispatchers)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: DiscNewsRepository

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Dispatcher ile Deneyler

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: İptalle (Cancellation) İlgili Hataların Düzeltilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: NotificationSender

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: BaseViewModel

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: CompanyDetailsRepository

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: CancellingRefresher

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: TokenRepository

Çözüm: Suspended lazy

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Eşzamanlılık (Concurrency) Sınırı ile mapAsync Fonksiyonu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: UserDetailsRepository Testi

Çözüm: mapAsync'in test edilmesi

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Solution: Testing the NotificationSender class

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: UserRefresher

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Kafeterya Simülasyonu

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `raceOf` Fonksiyonu

Çözüm: Flow Yardımcı Fonksiyonları (Flow Utils)

Çözüm: Tüm `user`'lari Yayınlayan `Flow`

Çözüm: `distinct` Operatörü

Çözüm: TemperatureService

Çözüm: `NewsViewModel`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `ProductService`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: Flow Kata

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `MessageService`

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `ProductService`'in Güncellenmiş Hali (`SharedFlow` ile)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `TemperatureService`'in Güncellenmiş Hali (`SharedFlow` ile)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `LocationService` (`stateIn` ile)

Bu içerik örnek kitapta mevcut değildir. Kitap Leanpub'dan https://leanpub.com/kotlincoroutinestrke adresinden satın alınabilir.

Çözüm: `PriceService`

Çözüm: `NewsViewModel` – `stateIn` Kullanılarak