Uniwersytet Wrocławski

Rafał Lewandków

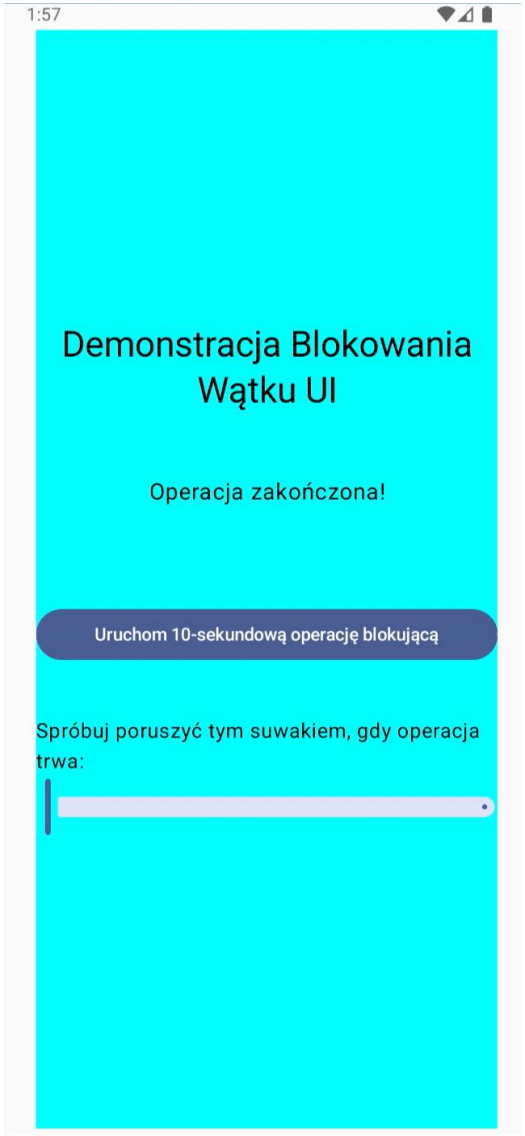
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

WYKŁAD 2

- Wprowadzenie do Wielowątkowości: Coroutines

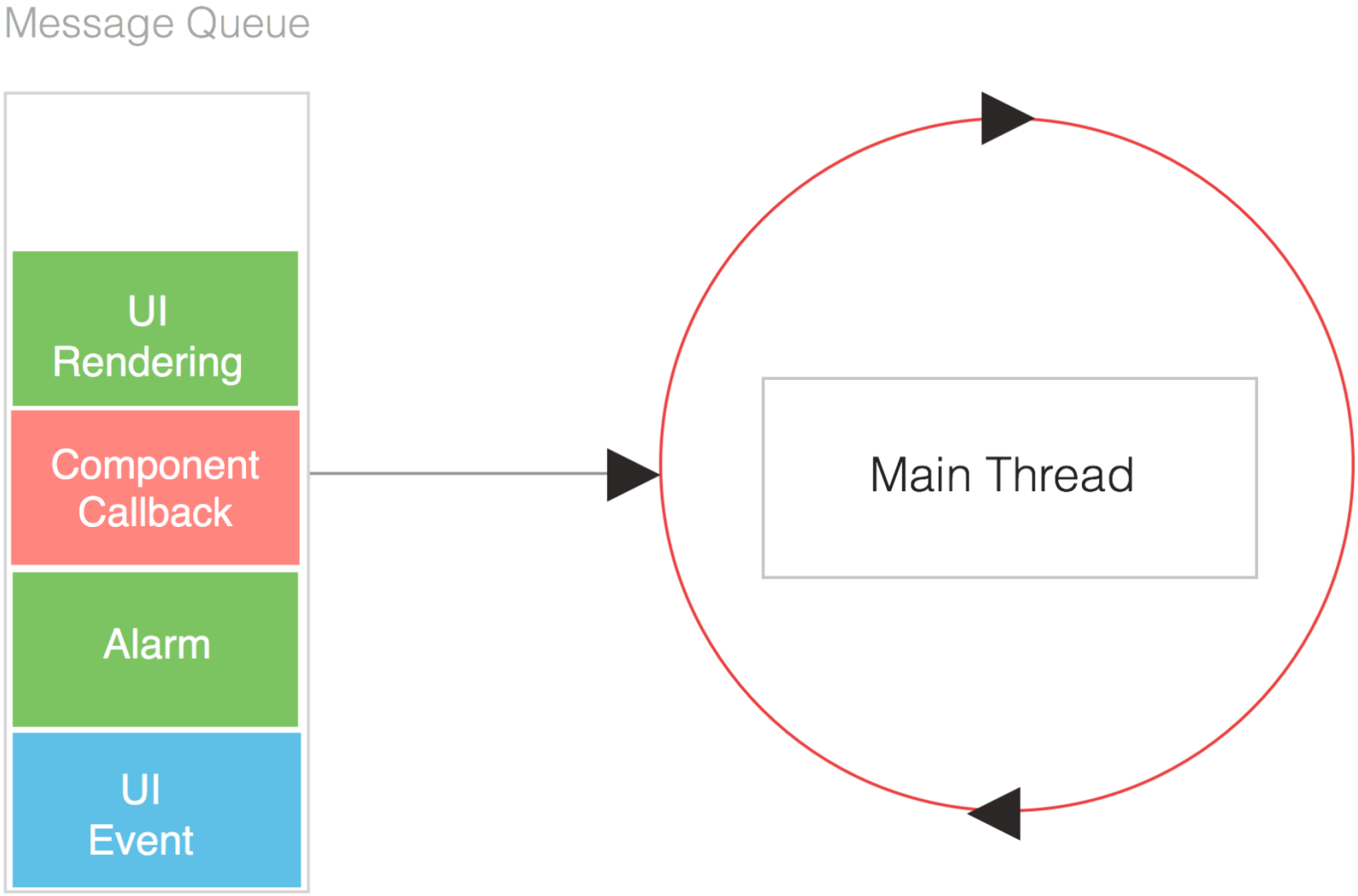
Blokowanie Wątku Głównego

Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.



Blokowanie Wątku Głównego

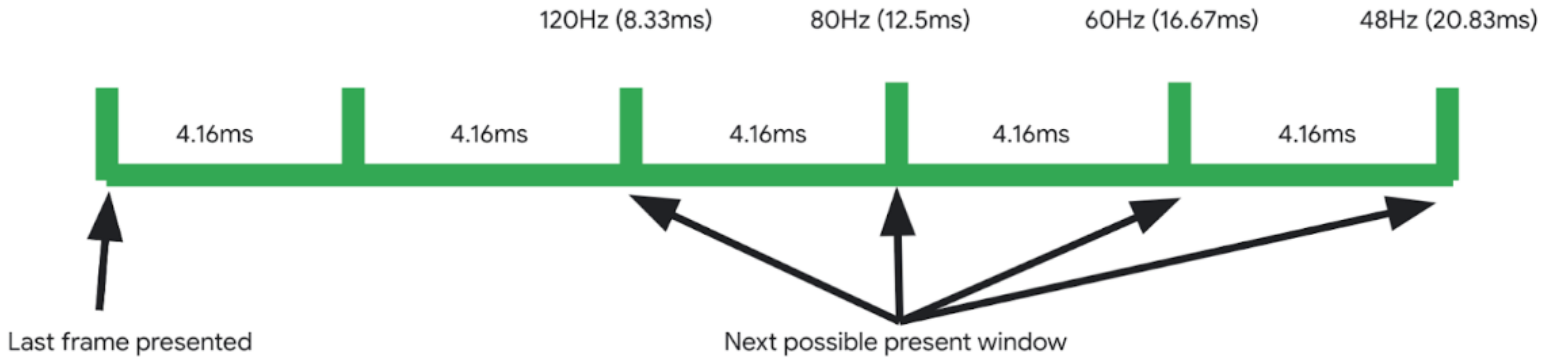
Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.

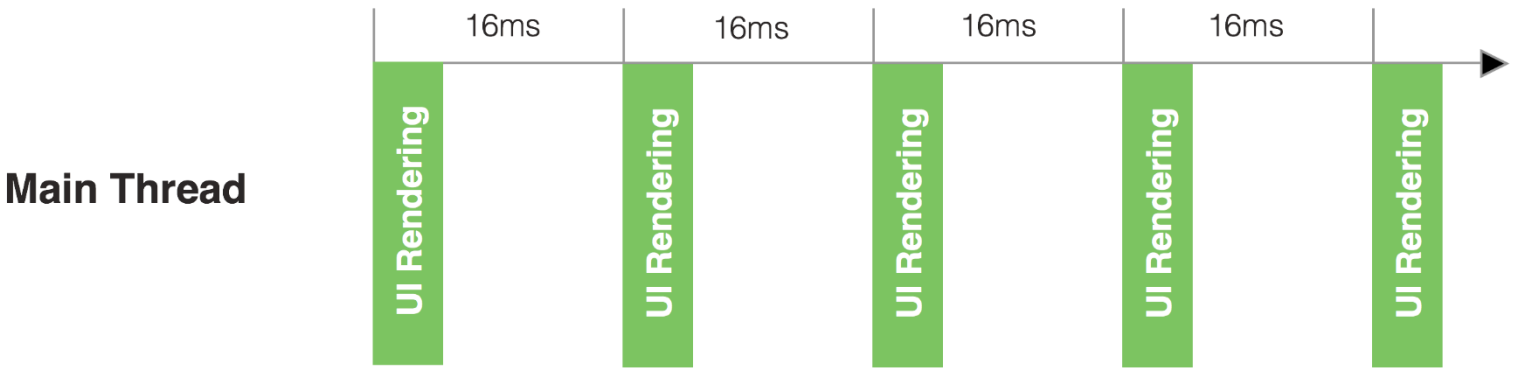


https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Blokowanie Wątku Głównego

Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.

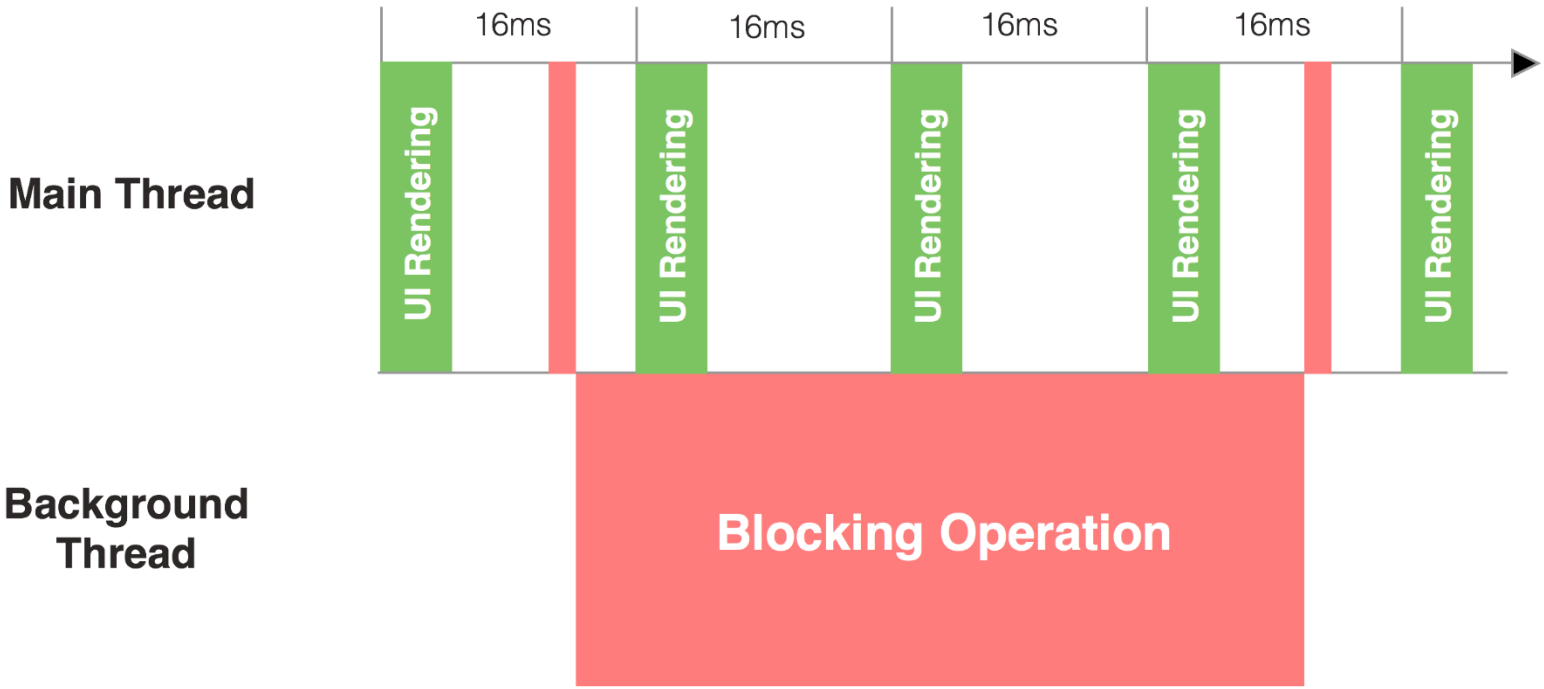




https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Blokowanie Wątku Głównego

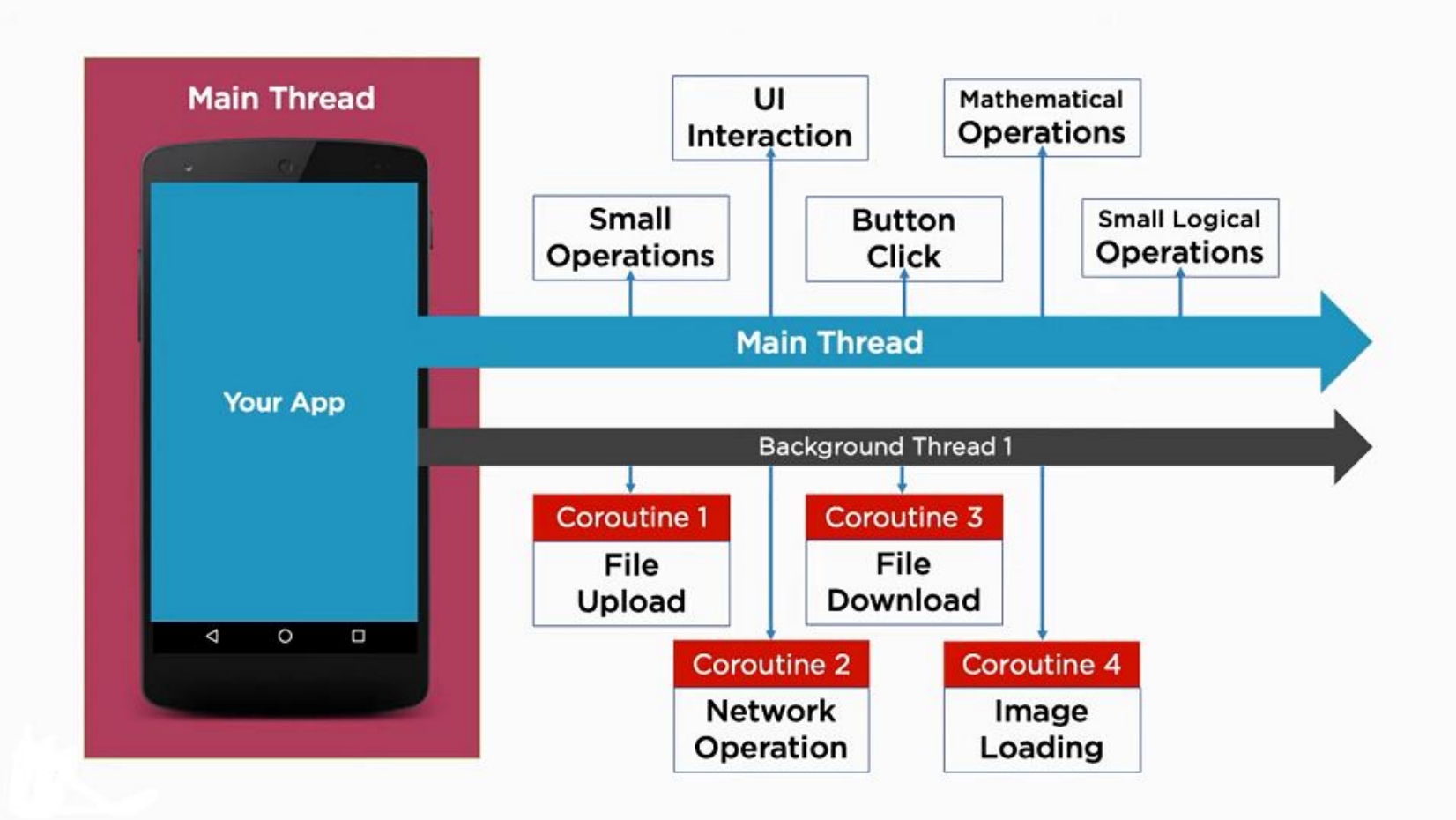
Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.



https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Coroutines

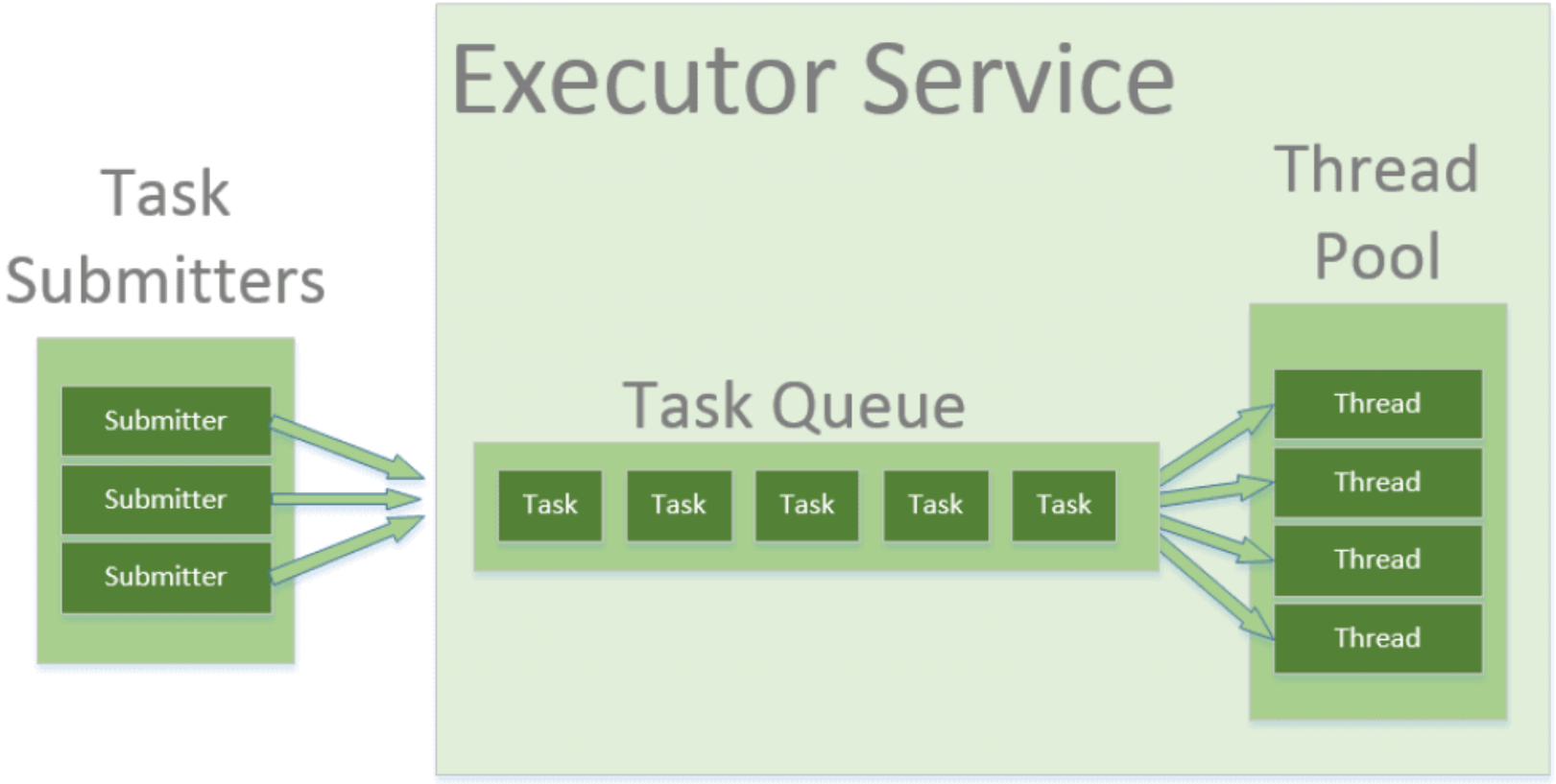
Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.



medium.com%2F%40mkcode0323%2Fthread-vs-coroutines-choosing-the-right-concurrency-approach-in-kotlin-android-3a56368d9768

Coroutines

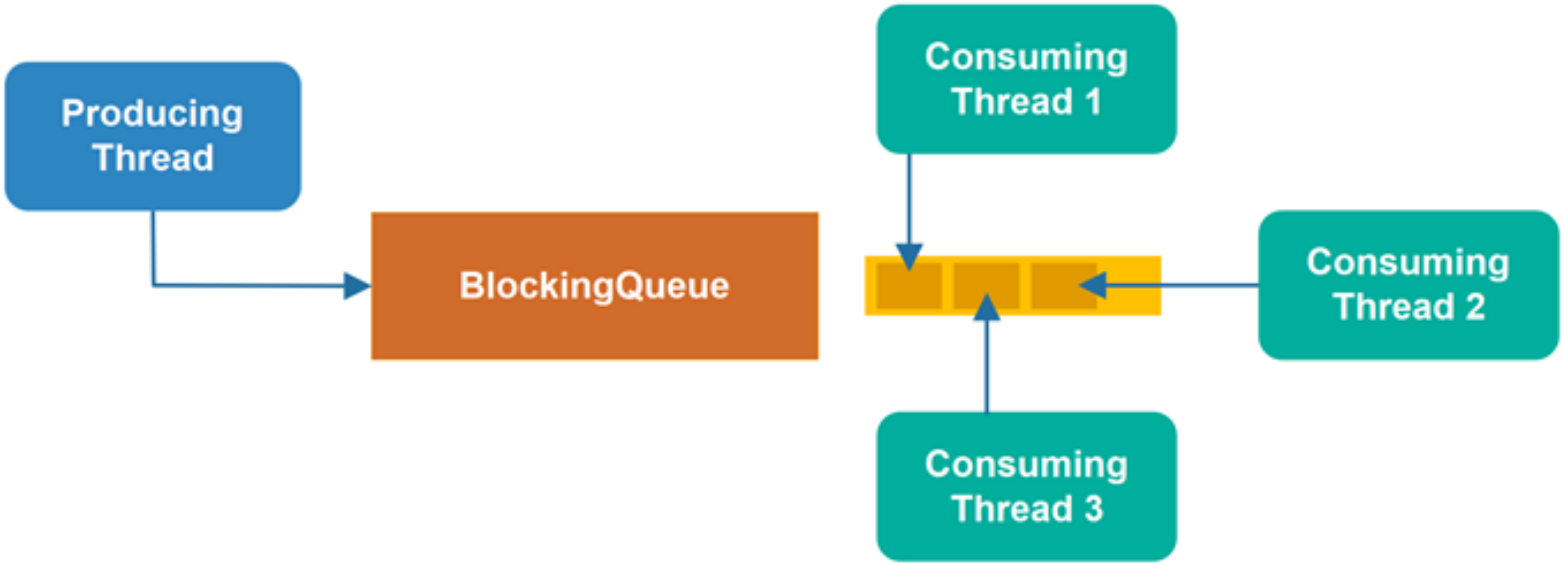
Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.



medium.com%2F%40mkcode0323%2Fthread-vs-coroutines-choosing-the-right-concurrency-approach-in-kotlin-android-3a56368d9768

Coroutines

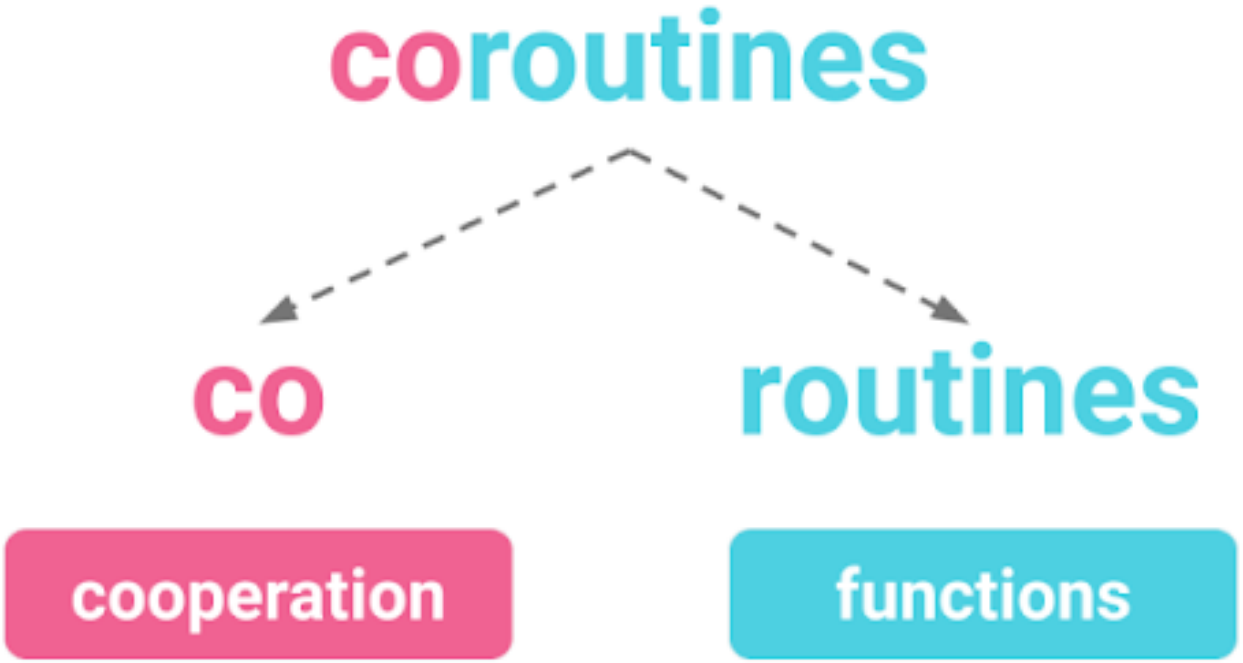
Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.



https://jenkov.com/tutorials/java-concurrency/thread-congestion.html

Coroutines

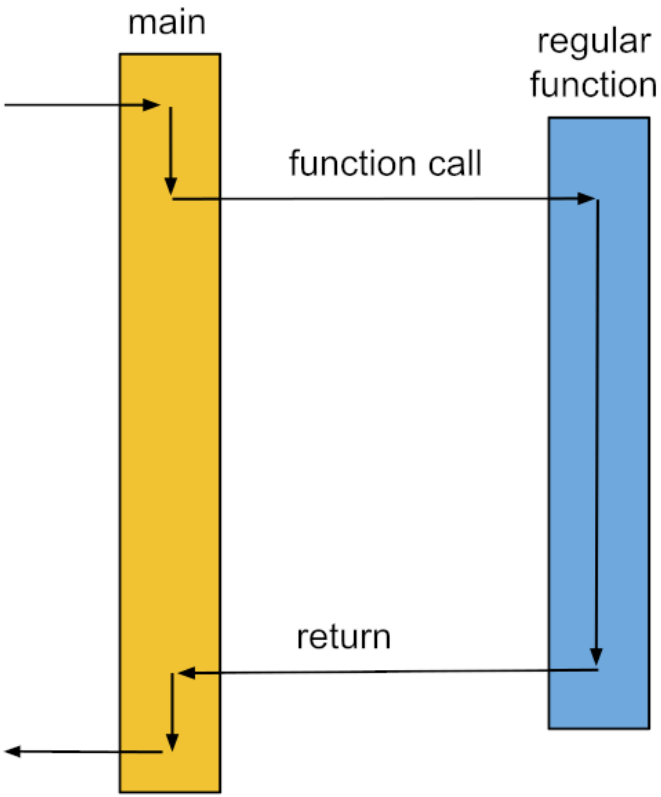
Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.

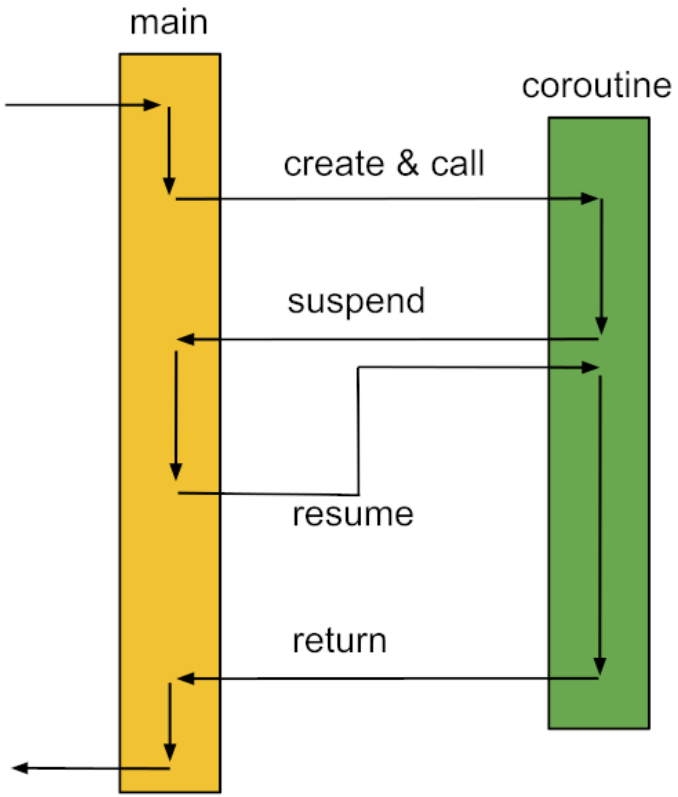


http://www.igfasouza.com/blog/coroutines/

Coroutines

Gdy długa, **synchroniczna operacja** jest wykonywana bezpośrednio w odpowiedzi na interakcję użytkownika aplikacja (i całe urządzenie) przestaje być **responsywna**.

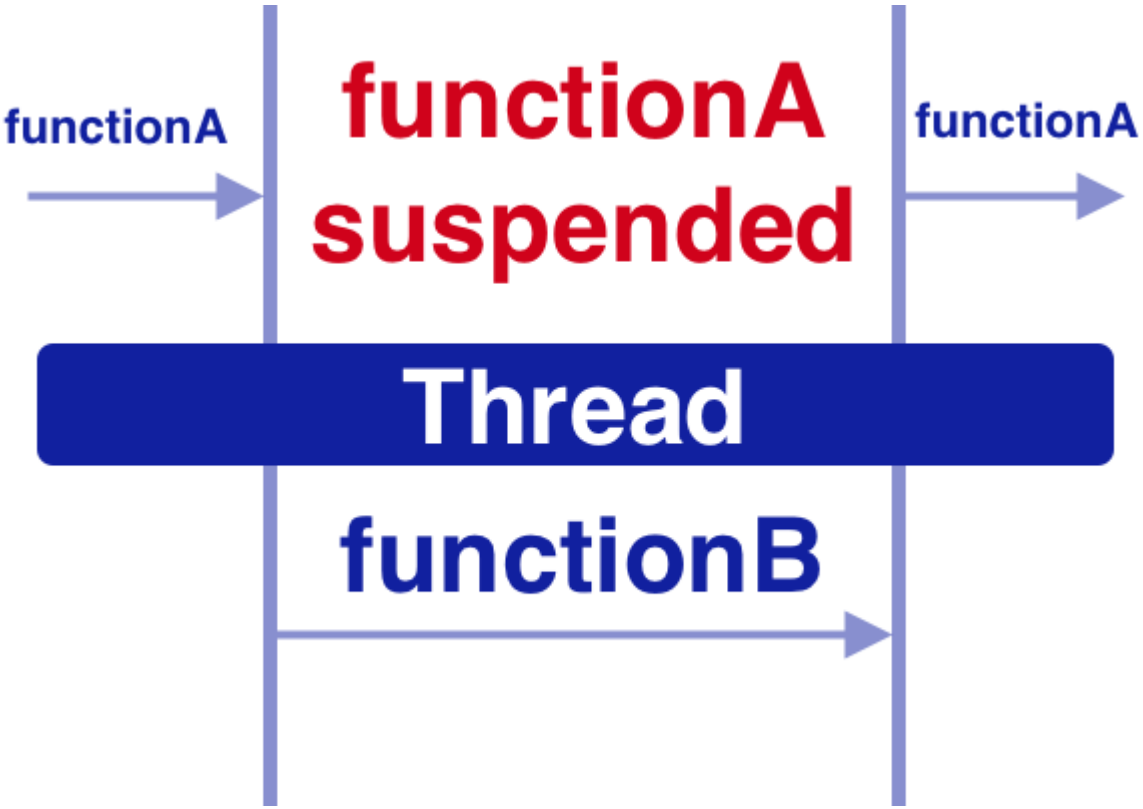




https://blog.eiler.eu/posts/20210512

suspend fun

W poprzednim przykładzie widzieliśmy, że Thread.sleep() **blokuje aplikację**. Rozwiązaniem jest użycie **mechanizmu**, który pozwala **'zapauzować'** zadanie **bez blokowania wątku**. Do tego służą funkcje oznaczone słowem kluczowym **suspend.**



https://outcomeschool.com/blog/suspend-function-in-kotlin-coroutines

Uniwersytet Wrocławski

suspend fun

W poprzednim przykładzie widzieliśmy, że Thread.sleep() **blokuje aplikację**. Rozwiązaniem jest użycie **mechanizmu**, który pozwala **'zapauzować'** zadanie **bez blokowania wątku**. Do tego służą funkcje oznaczone słowem kluczowym **suspend.**

Thread.sleep(10000)	**delay(10000)**
Blokuje cały wątek	Zawiesza **tylko** korutynę
Nikt inny nie może używać wątku	Wątek jest zwalniany
Powoduje zamrożenie ui jeżeli użyte na wątku głównym	Ui pozostaje w pełni responsywne

suspend fun

Funkcja z możliwością zawieszenia wykonania

```
suspend fun fetchDataFromServer(): String {
    println("Operacja w tle rozpoczęta na wątku: ${Thread.currentThread().name}")
    delay( timeMillis = 5000)
    println("Operacja w tle zakończona.")
    return "Dane pobrane z serwera!"
}
```

suspend fun **może być** wywołana wewnątrz **innej** suspend fun

Uniwersytet Wrocławski

suspend fun

```
@Composable
fun SuspendFunctionDemoScreen() {
    var statusText by remember { mutableStateOf( value = "Gotowy do działania.") }

    Column(...) {
        Text(...)
        Spacer(modifier = Modifier.height( height = 50.dp))

        Text(...)
        Spacer(modifier = Modifier.height( height = 20.dp))

        Button(
            onClick = {
                statusText = "Próba uruchomienia operacji..."

                val result = fetchDataFromServer()

                // statusText = result
            },
```

Suspend function 'suspend fun fetchDataFromServer(): String' can only be called from a coroutine or another suspend function.

Uniwersytet Wrocławski

suspend fun

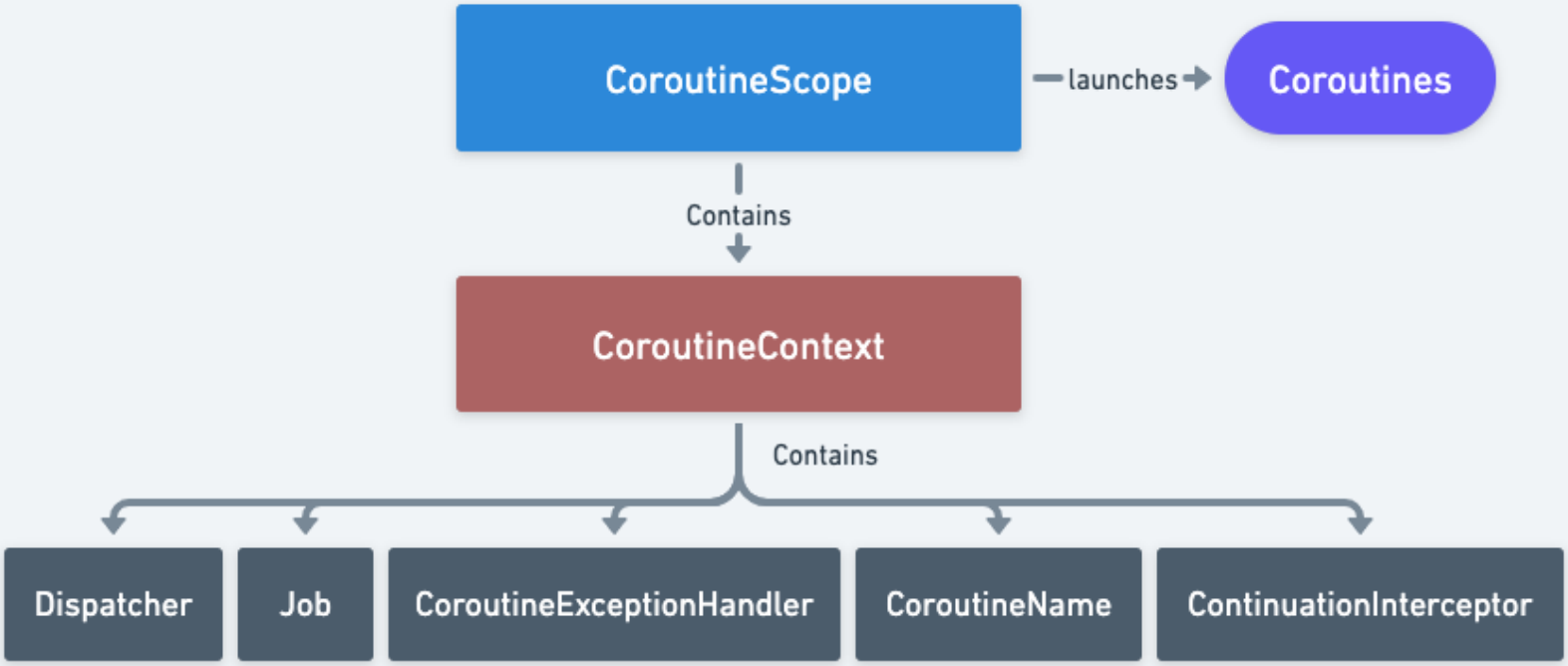
```
@Composable
fun SuspendFunctionDemoScreen() {
    var statusText by remember { mutableStateOf( value = "Gotowy do działania.") }

    Column(...) {
        Text(...)
        Spacer(modifier = Modifier.height( height = 50.dp))

        Text(...)
        Spacer(modifier = Modifier.height( height = 20.dp))

        Button(
            onClick = {
                statusText = "Próba uruchomienia operacji..."

                val result = fetchDataFromServer()

                // statusText = result
            },
```

Suspend function 'suspend fun fetchDataFromServer(): String' can only be called from a coroutine or another suspend function.

Kotlin chroni nas przed **przypadkowym** wywołaniem **potencjalnie długiej**, zawieszalnej **operacji** w miejscu, które **nie jest do tego przygotowane**. Zwykły blok onClick **nie wie, jak zarządzać** 'pauzowaniem' i 'wznawianiem' funkcji.

Coroutine Scope

Teraz musimy stworzyć odpowiednie **środowisko uruchomieniowe** dla funkcji **suspend**. Tym środowiskiem jest **CoroutineScope**, a narzędziem do startu – konstruktor **launch.**

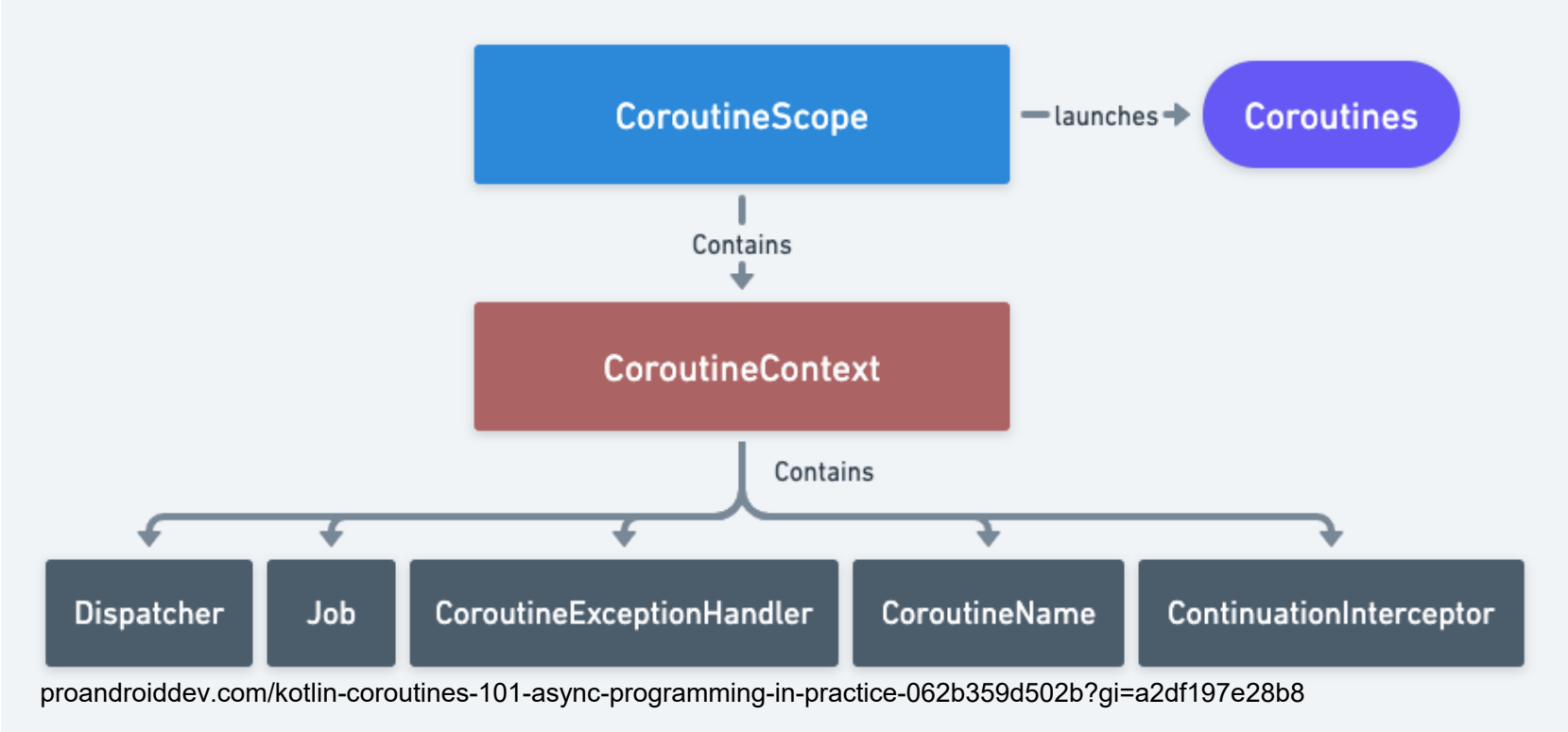


proandroiddev.com/kotlin-coroutines-101-async-programming-in-practice-062b359d502b?gi=a2df197e28b8

Uniwersytet Wrocławski

Coroutine Scope

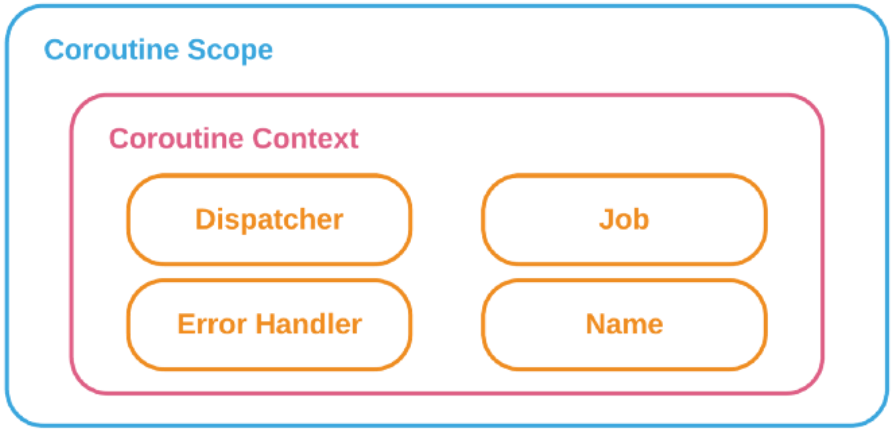
Teraz musimy stworzyć odpowiednie **środowisko uruchomieniowe** dla funkcji **suspend**. Tym środowiskiem jest **CoroutineScope**, a narzędziem do startu – konstruktor **launch.**



proandroiddev.com/kotlin-coroutines-101-async-programming-in-practice-062b359d502b?gi=a2df197e28b8

CoroutineScope jest jak nadzorca na placu budowy. Kiedy kończy się dzień pracy (np. **zamykamy ekran**), nadzorca upewnia się, że **wszyscy robotnicy** (**korutyny**) poszli do domu. Nikt **nie zostaje** po godzinach i nie powoduje problemów (**wycieków pamięci**).

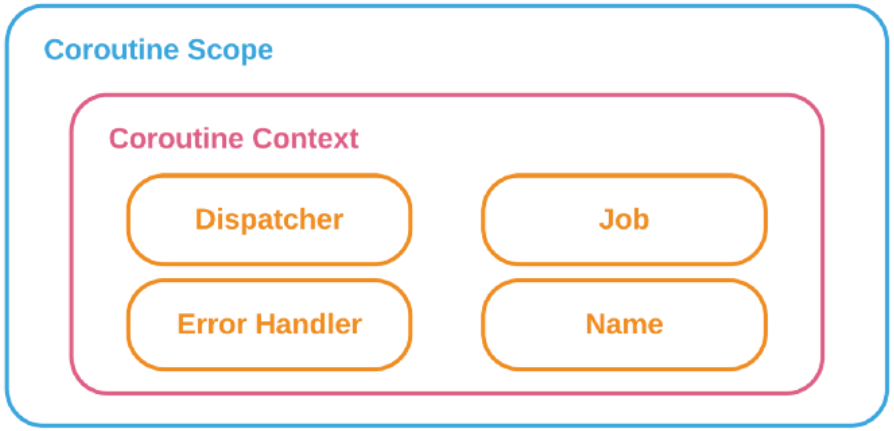
Coroutine Scope



https://write.agrevolution.in/kotlin-coroutines-part-3-coroutine-context-bd5543389190

CoroutineContext to **zbiór zasad i konfiguracji**, które definiują, jak i gdzie dana korutyna ma być wykonana. Określa on jej zachowanie, na przykład na którym wątku powinna działać i jak zarządzać jej cyklem życia.

Coroutine Scope

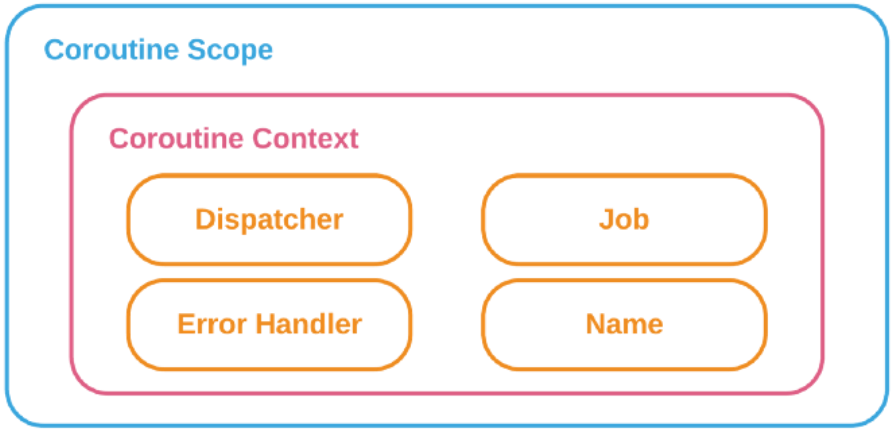


https://write.agrevolution.in/kotlin-coroutines-part-3-coroutine-context-bd5543389190

CoroutineContext to **zbiór zasad i konfiguracji**, które definiują, jak i gdzie dana korutyna ma być wykonana. Określa on jej zachowanie, na przykład na którym wątku powinna działać i jak zarządzać jej cyklem życia.

CoroutineContext jest **zestawem instrukcji, narzędzi i przydziałem do miejsca pracy**, który każdy pracownik otrzymuje.

Coroutine Scope



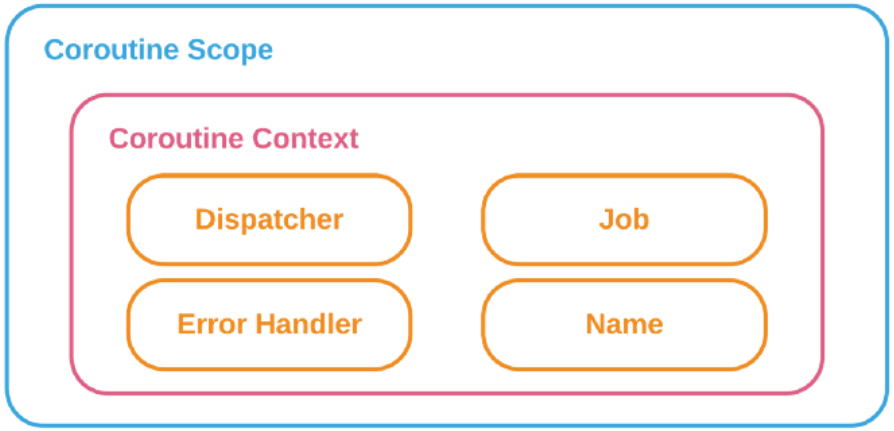
https://write.agrevolution.in/kotlin-coroutines-part-3-coroutine-context-bd5543389190

CoroutineContext to **zbiór zasad i konfiguracji**, które definiują, jak i gdzie dana korutyna ma być wykonana. Określa on jej zachowanie, na przykład na którym wątku powinna działać i jak zarządzać jej cyklem życia.

CoroutineContext jest **zestawem instrukcji, narzędzi i przydziałem do miejsca pracy**, który każdy pracownik otrzymuje.

Każda korutyna uruchomiona **w danym zakresie** (scope) **dziedziczy jego kontekst**, czyli jego "**regulamin pracy**".

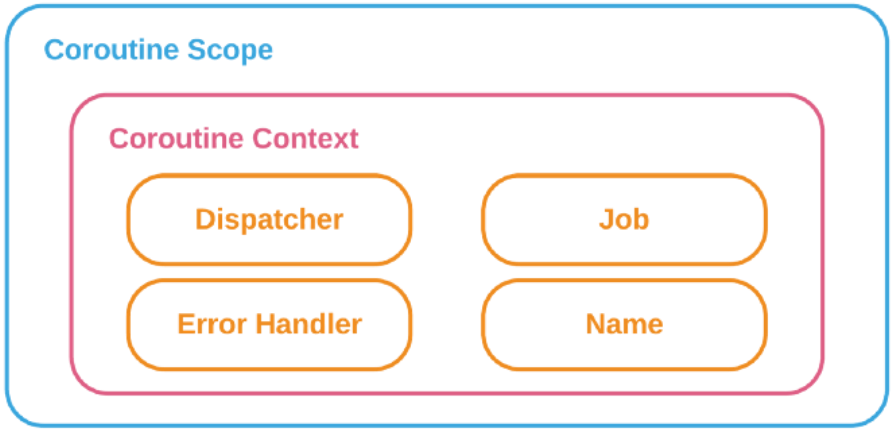
Coroutine Scope



https://write.agrevolution.in/kotlin-coroutines-part-3-coroutine-context-bd5543389190

Job: Reprezentuje **cykl życia** i **stan zadania** korutyny. To jej "**karta identyfikacyjna**". CoroutineScope używa Job do śledzenia korutyny – anulowania, sprawdzania, czy jest aktywna itp.

Coroutine Scope

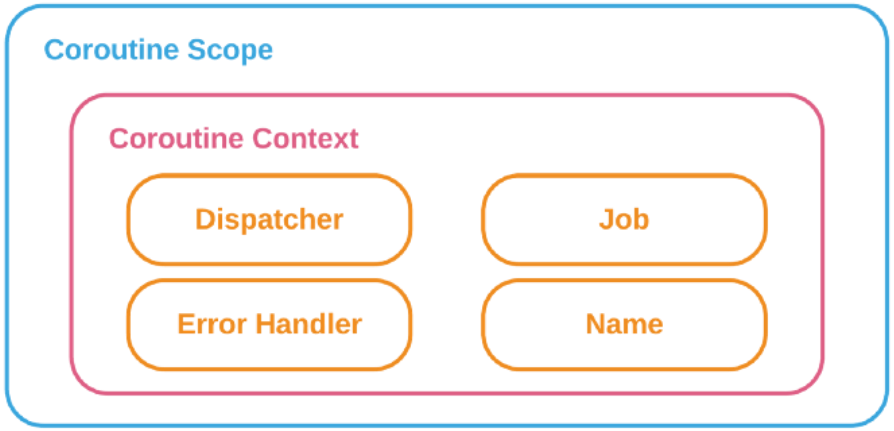


https://write.agrevolution.in/kotlin-coroutines-part-3-coroutine-context-bd5543389190

Job: Reprezentuje **cykl życia** i **stan zadania** korutyny. To jej "**karta identyfikacyjna**". CoroutineScope używa Job do śledzenia korutyny – anulowania, sprawdzania, czy jest aktywna itp.

CoroutineDispatcher: Określa, na którym **wątku** (lub **puli wątków**) korutyna ma wykonać swoją pracę. To jest jej "**przydział do miejsca pracy**".

Coroutine Scope



https://write.agrevolution.in/kotlin-coroutines-part-3-coroutine-context-bd5543389190

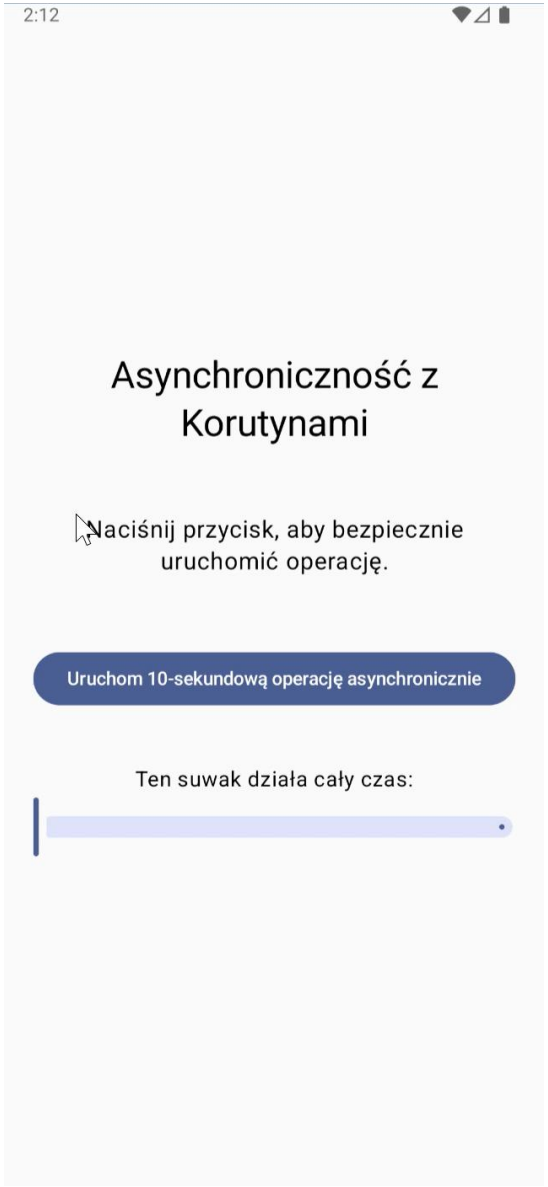
Job: Reprezentuje **cykl życia** i **stan zadania** korutyny. To jej "**karta identyfikacyjna**". CoroutineScope używa Job do śledzenia korutyny – anulowania, sprawdzania, czy jest aktywna itp.

CoroutineDispatcher: Określa, na którym **wątku** (lub **puli wątków**) korutyna ma wykonać swoją pracę. To jest jej "**przydział do miejsca pracy**".

- **Dispatchers.Main:** Wątek główny Androida, jedyny, na którym można bezpiecznie modyfikować UI.
- **Dispatchers.IO:** Pula wątków zoptymalizowana pod operacje wejścia-wyjścia (I/O), takie jak odczyt z sieci czy bazy danych.
- **Dispatchers.Default:** Pula wątków do zadań intensywnie obciążających procesor (CPU), np. sortowanie dużej listy.

Uniwersytet Wrocławski

Asynchroniczność



Uniwersytet Wrocławski

Asynchroniczność

```
suspend fun fetchDataFromServer(): String {
    println("Korutyna: Zaczynam pobieranie danych na wątku: ${Thread.currentThread().name}")
    delay( timeMillis = 10000)
    println("Korutyna: Zakończono pobieranie danych.")
    return "Dane pobrane pomyślnie! ✅"
}
```

Uniwersytet Wrocławski

Asynchroniczność

Używamy rememberCoroutineScope(), aby uzyskać **zakres powiązany** z **cyklem życia** naszego **komponentu**. Funkcja jest ***Lifecycle-aware*** – automatycznie anuluje wszystkie uruchomione w nim korutyny, gdy ten komponent zniknie z ekranu.
Inaczej mówiąc, jest to mechanizm **automatycznego** sprzątania, który zapobiega wyciekom pamięci i błędom.

```
Composable
n CoroutineSolutionScreen() {
    var statusText by remember { mutableStateOf( value = "Naciśnij przycisk,

    val scope = rememberCoroutineScope()

    Column(...) {
        Text(...)
        Spacer(modifier = Modifier.height( height = 50.dp))

        Text(...)
        Spacer(modifier = Modifier.height( height = 20.dp))

        Button(
            onClick = {
                statusText = "Operacja rozpoczęta w tle..."


                scope.launch {
                    val result = fetchDataFromServer()

                    statusText = result
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text( text = "Uruchom 10-sekundową operację asynchronicznie")
        }
        Spacer(modifier = Modifier.height( height = 40.dp))
```

Asynchroniczność

Używamy rememberCoroutineScope(), aby uzyskać **zakres powiązany** z **cyklem życia** naszego **komponentu**. Funkcja jest ***Lifecycle-aware*** – automatycznie anuluje wszystkie uruchomione w nim korutyny, gdy ten komponent zniknie z ekranu.
Inaczej mówiąc, jest to mechanizm **automatycznego** sprzątania, który zapobiega wyciekom pamięci i błędom.

Następnie wywołujemy **launch**. Jego **głównym celem** jest **inicjowanie operacji**, od których **nie oczekujemy** bezpośredniego **zwrotu wyniku**. Działa na zasadzie "**odpal i zapomnij**" (fire-and-forget).
Rozkazujemy nadzorcy (**scope**) uruchomienie nowego pracownika (**coroutine**) i zleć mu wykonanie **bloku kodu.**

```
Composable
n CoroutineSolutionScreen() {
    var statusText by remember { mutableStateOf( value = "Naciśnij przycisk,

    val scope = rememberCoroutineScope()

    Column(...) {
        Text(...)
        Spacer(modifier = Modifier.height( height = 50.dp))

        Text(...)
        Spacer(modifier = Modifier.height( height = 20.dp))

        Button(
            onClick = {
                statusText = "Operacja rozpoczęta w tle..."


                scope.launch {
                    val result = fetchDataFromServer()

                    statusText = result
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text( text = "Uruchom 10-sekundową operację asynchronicznie")
        }
        Spacer(modifier = Modifier.height( height = 40.dp))
```

Asynchroniczność

Używamy rememberCoroutineScope(), aby uzyskać **zakres powiązany** z **cyklem życia** naszego **komponentu**. Funkcja jest ***Lifecycle-aware*** – automatycznie anuluje wszystkie uruchomione w nim korutyny, gdy ten komponent zniknie z ekranu.
Inaczej mówiąc, jest to mechanizm **automatycznego** sprzątania, który zapobiega wyciekom pamięci i błędom.

```
Composable
n CoroutineSolutionScreen() {
    var statusText by remember { mutableStateOf( value = "Naciśnij przycisk,

    val scope = rememberCoroutineScope()

    Column(...) {
        Text(...)
        Spacer(modifier = Modifier.height( height = 50.dp))

        Text(...)
        Spacer(modifier = Modifier.height( height = 20.dp))

        Button(
            onClick = {
                statusText = "Operacja rozpoczęta w tle..."


                scope.launch {
                    val result = fetchDataFromServer()

                    statusText = result
                }
            },
            modifier =
        ) {
            Text( text =
        }
        Spacer(modifier
```

Następnie wywołujemy **launch**. Jego **głównym celem** jest **inicjowanie operacji**, od których **nie oczekujemy** bezpośredniego **zwrotu wyniku**. Działa na zasadzie "**odpal i zapomnij**" (fire-and-forget).
Rozkazujemy nadzorcuy (**scope**) uruchomienie nowego pracownika (**coroutine**) i zleć mu wykonanie **bloku kodu.**

Ponieważ jesteśmy już **wewnątrz korutyny**, **możemy** bez problemu **wywołać** naszą funkcję **suspend**. Kompilator jest zadowolony.

Asynchroniczność

Używamy rememberCoroutineScope(), aby uzyskać **zakres powiązany** z **cyklem życia** naszego **komponentu**. Funkcja jest ***Lifecycle-aware*** – automatycznie anuluje wszystkie uruchomione w nim korutyny, gdy ten komponent zniknie z ekranu.
Inaczej mówiąc, jest to mechanizm **automatycznego** sprzątania, który zapobiega wyciekom pamięci i błędom.

```
Composable
n CoroutineSolutionScreen() {
    var statusText by remember { mutableStateOf( value = "Naciśnij przycisk,

    val scope = rememberCoroutineScope()

    Column(...) {
        Text(...)
        Spacer(modifier = Modifier.height( height = 50.dp))

        Text(...)
        Spacer(modifier = M

        Button(
            onClick = {
                statusText =


                scope.launch {
                    val result = fetchDataFromServer()

                    statusText = result
                }
            },
            modifier =
        ) {
            Text( text =
        }
        Spacer(modifier
```

Następnie wywołujemy **launch**. Jego **głównym celem** jest **inicjowanie operacji**, od których **nie oczekujemy** bezpośredniego **zwrotu wyniku**. Działa na zasadzie "**odpal i zapomnij**" (fire-and-forget).
Rozkazujemy nadzorcuy (**scope**) uruchomienie nowego pracownika (**coroutine**) i zleć mu wykonanie **bloku kodu.**

launch zadba o to, by ta **aktualizacja odbyła się** w **bezpieczny** sposób na **wątku głównym**.

Ponieważ jesteśmy już **wewnątrz korutyny**, **możemy** bez problemu **wywołać** naszą funkcję **suspend**. Kompilator jest zadowolony.