

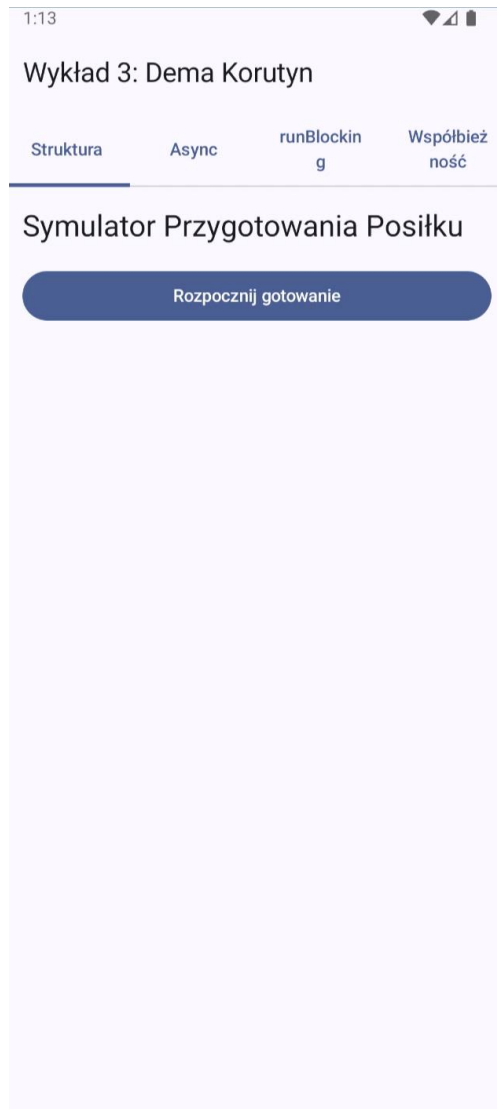


PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

WYKŁAD 3

- Coroutines

Na poprzednim wykładzie dowiedzieliśmy się, że korutyny „**żyją**” w **CoroutineScope**. **Ustrukturyzowana Współbieżność** to zasada, która mówi, że **cykl życia** korutyny jest **nierozzerwalnie związany** z jej zakresem (scope).



Ustrukturyzowana Współbieżność

Tworzy listę, która przechowuje logi z operacji

Tworzy **bezpieczny** zakres korutyn, **powiązany** z **cyklem życia** tego komponentu.

```
fun MealPrepSimulatorScreen() {  
    val logs = remember { mutableStateListOf<String>() }  
    val scope = rememberCoroutineScope()  
  
    Column(modifier = Modifier.fillMaxSize().padding( all = 16.dp)) {  
        Text( text = "Symulator Przygotowania Posiłku", style = MaterialTheme.typography.l  
        Spacer(Modifier.height( height = 16.dp))  
        Button(onClick = {  
            logs.clear()  
            scope.launch {  
                logs.add("Szef kuchni (rodzic): Zaczynamy!")  
  
                val jobMieso = launch {  
                    delay( timeMillis = 2000)  
                    logs.add("Kucharz 1: Mięso usmażone (2s).")  
                }  
  
                val jobWarzywa = launch {  
                    delay( timeMillis = 3000)  
                    logs.add("Kucharz 2: Warzywa gotowe (3s).")  
                }  
  
                logs.add("Szef kuchni: Zadania zlecone, scope czeka na zakończenie...")  
  
                jobWarzywa.join()  
                jobMieso.join()  
  
                logs.add("Szef kuchni: Wszyscy skończyli! Można podawać danie.")  
            }  
        }, modifier = Modifier.fillMaxWidth()) {  
            Text( text = "Rozpocznij gotowanie")  
        }  
        Spacer(Modifier.height( height = 16.dp))  
        LazyColumn {...}  
    }  
}
```

Ustrukturyzowana Współbieżność

Tworzy listę, która przechowuje logi z operacji

Tworzy **bezpieczny** zakres korutyn, **powiązany** z **cyklem życia** tego komponentu.

Uruchamia **główną** korutynę.

```
fun MealPrepSimulatorScreen() {  
    val logs = remember { mutableStateListOf<String>() }  
    val scope = rememberCoroutineScope()  
  
    Column(modifier = Modifier.fillMaxSize().padding( all = 16.dp)) {  
        Text( text = "Symulator Przygotowania Posiłku", style = MaterialTheme.typography.l  
        Spacer(Modifier.height( height = 16.dp))  
        Button(onClick = {  
            logs.clear()  
            scope.launch {  
                logs.add("Szef kuchni (rodzic): Zaczynamy!")  
  
                val jobMieso = launch {  
                    delay( timeMillis = 2000)  
                    logs.add("Kucharz 1: Mięso usmażone (2s).")  
                }  
  
                val jobWarzywa = launch {  
                    delay( timeMillis = 3000)  
                    logs.add("Kucharz 2: Warzywa gotowe (3s).")  
                }  
  
                logs.add("Szef kuchni: Zadania zlecone, scope czeka na zakończenie...")  
  
                jobWarzywa.join()  
                jobMieso.join()  
  
                logs.add("Szef kuchni: Wszyscy skończyli! Można podawać danie.")  
            }  
        }, modifier = Modifier.fillMaxWidth()) {  
            Text( text = "Rozpocznij gotowanie")  
        }  
        Spacer(Modifier.height( height = 16.dp))  
        LazyColumn {...}  
    }  
}
```

Ustrukturyzowana Współbieżność

Tworzy listę, która przechowuje logi z operacji

Tworzy **bezpieczny** zakres korutyn, **powiązany** z **cyklem życia** tego komponentu.

Uruchamia **główną** korutynę.

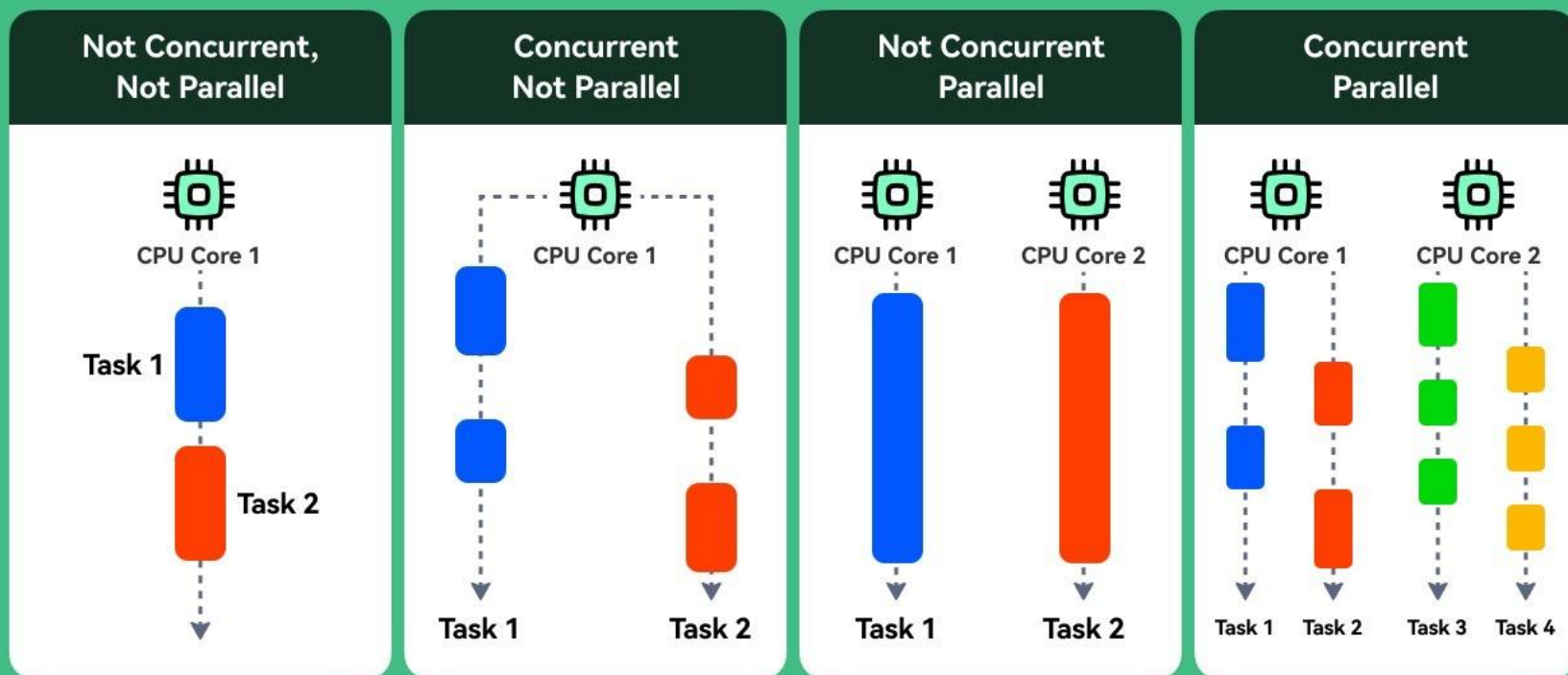
Uruchamiane są **współbieżne** korutyny. Obie korutyny startują niemal w tym samym momencie i działają **współbieżnie**.

```
fun MealPrepSimulatorScreen() {  
    val logs = remember { mutableStateListOf<String>() }  
    val scope = rememberCoroutineScope()  
  
    Column(modifier = Modifier.fillMaxSize().padding( all = 16.dp)) {  
        Text( text = "Symulator Przygotowania Posiłku", style = MaterialTheme.typography.l  
        Spacer(Modifier.height( height = 16.dp))  
        Button(onClick = {  
            logs.clear()  
            scope.launch {  
                logs.add("Szef kuchni (rodzic): Zaczynamy!")  
  
                val jobMieso = launch {  
                    delay( timeMillis = 2000)  
                    logs.add("Kucharz 1: Mięso usmażone (2s).")  
                }  
  
                val jobWarzywa = launch {  
                    delay( timeMillis = 3000)  
                    logs.add("Kucharz 2: Warzywa gotowe (3s).")  
                }  
  
                logs.add("Szef kuchni: Zadania zlecone, scope czeka na zakończenie...")  
  
                jobWarzywa.join()  
                jobMieso.join()  
  
                logs.add("Szef kuchni: Wszyscy skończyli! Można podawać danie.")  
            }  
        }, modifier = Modifier.fillMaxWidth()) {  
            Text( text = "Rozpocznij gotowanie")  
        }  
        Spacer(Modifier.height( height = 16.dp))  
        LazyColumn {...}  
    }  
}
```

Współbieżność vs Równoległość

Współbieżność polega na **zdolności systemu** do obsługi **wielu zadań jednocześnie**, co niekoniecznie oznacza ich wykonywanie w tym samym fizycznym momencie. Chodzi o **zarządzanie wieloma zadaniami** i przełączanie się między nimi tak, aby wszystkie posuwały się do przodu.

Concurrency is **NOT** Parallelism



Ustrukturyzowana Współbieżność

Tworzy listę, która przechowuje logi z operacji

Tworzy **bezpieczny** zakres korutyn, **powiązany** z **cyklem życia** tego komponentu.

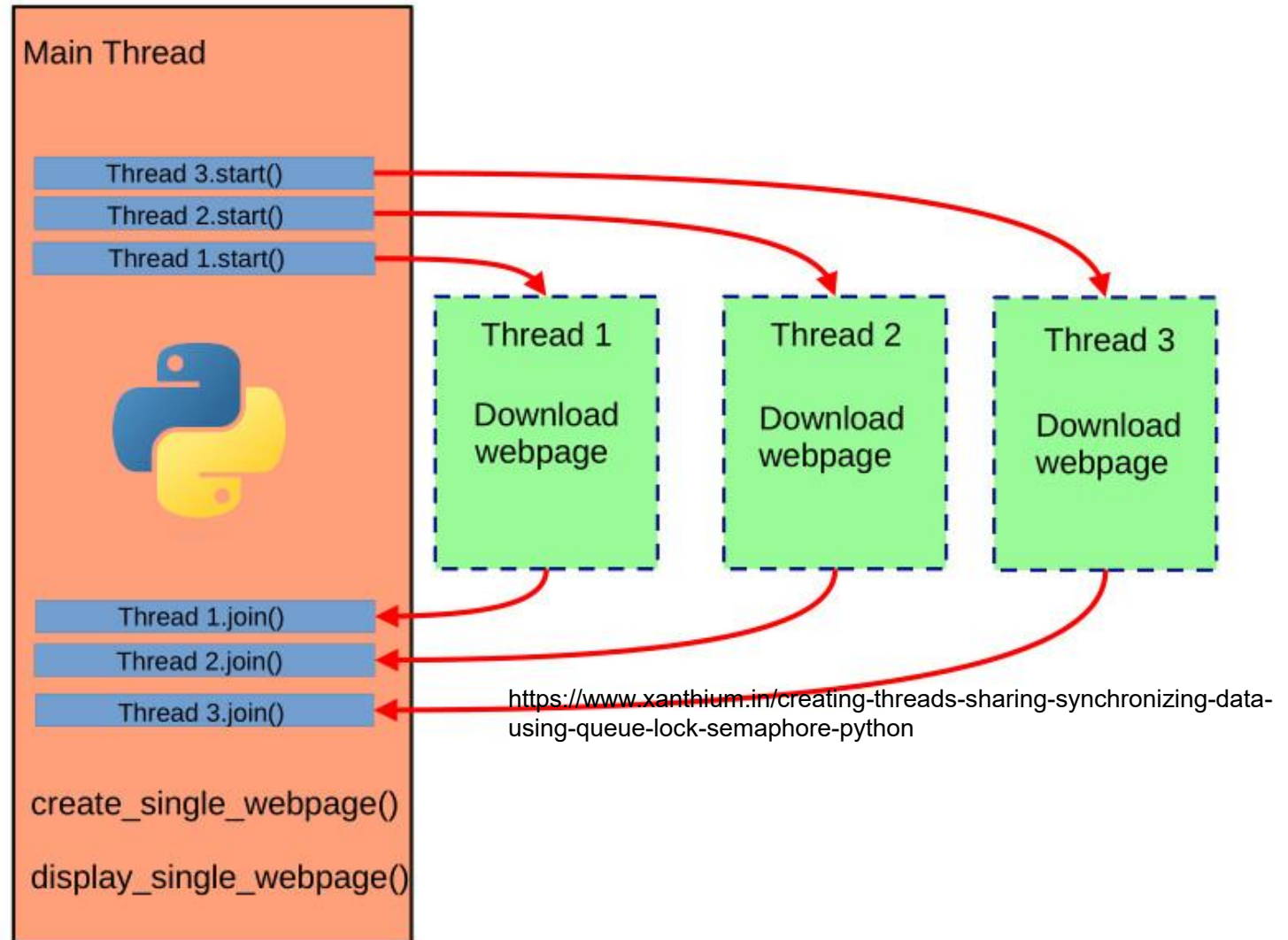
Uruchamia **główną** korutynę.

Uruchamiane są **współbieżne** korutyny. Obie korutyny startują niemal w tym samym momencie i działają **współbieżnie**.

Aby wykonać akcję **po zakończeniu wszystkich zadań-dzieci**, musimy na nie **jawnie poczekać**. Służy do tego funkcja `join()`.

```
fun MealPrepSimulatorScreen() {  
    val logs = remember { mutableStateListOf<String>() }  
    val scope = rememberCoroutineScope()  
  
    Column(modifier = Modifier.fillMaxSize().padding( all = 16.dp)) {  
        Text( text = "Symulator Przygotowania Posiłku", style = MaterialTheme.typography.l  
        Spacer(Modifier.height( height = 16.dp))  
        Button(onClick = {  
            logs.clear()  
            scope.launch {  
                logs.add("Szef kuchni (rodzic): Zaczynamy!")  
  
                val jobMieso = launch {  
                    delay( timeMillis = 2000)  
                    logs.add("Kucharz 1: Mięso usmażone (2s).")  
                }  
  
                val jobWarzywa = launch {  
                    delay( timeMillis = 3000)  
                    logs.add("Kucharz 2: Warzywa gotowe (3s).")  
                }  
  
                logs.add("Szef kuchni: Zadania zlecone, scope czeka na zakończenie...")  
  
                jobWarzywa.join()  
                jobMieso.join()  
  
                logs.add("Szef kuchni: Wszyscy skończyli! Można podawać danie.")  
            }  
        }, modifier = Modifier.fillMaxWidth()) {  
            Text( text = "Rozpocznij gotowanie")  
        }  
        Spacer(Modifier.height( height = 16.dp))  
        LazyColumn {...}  
    }  
}
```


Download multiple webpages and join them as single page



Ustrukturyzowana Współbieżność

Tworzy listę, która przechowuje logi z operacji

Tworzy **bezpieczny** zakres korutyn, **powiązany** z **cyklem życia** tego komponentu.

Uruchamia **główną** korutynę.

Uruchamiane są **współbieżne** korutyny. Obie korutyny startują niemal w tym samym momencie i działają **współbieżnie**.

Aby wykonać akcję **po zakończeniu wszystkich zadań-dzieci**, musimy na nie **jawnie poczekać**. Służy do tego funkcja `join()`.

`launch` zwraca obiekt typu `Job`, który **reprezentuje cykl życia** korutyny. `job.join()` to funkcja `suspend`, która **zawiesza korutynę**, w której została wywołana (w tym przypadku rodzica), do czasu, aż `job` się zakończy.

```
fun MealPrepSimulatorScreen() {  
    val logs = remember { mutableStateListOf<String>() }  
    val scope = rememberCoroutineScope()  
  
    Column(modifier = Modifier.fillMaxSize().padding( all = 16.dp)) {  
        Text( text = "Symulator Przygotowania Posiłku", style = MaterialTheme.typography.h1)  
        Spacer(Modifier.height( height = 16.dp))  
        Button(onClick = {  
            logs.clear()  
            scope.launch {  
                logs.add("Szef kuchni (rodzic): Zaczynamy!")  
  
                val jobMieso = launch {  
                    delay( timeMillis = 2000)  
                    logs.add("Kucharz 1: Mięso usmażone (2s).")  
                }  
  
                val jobWarzywa = launch {  
                    delay( timeMillis = 3000)  
                    logs.add("Kucharz 2: Warzywa gotowe (3s).")  
                }  
  
                logs.add("Szef kuchni: Zadania zlecone, scope czeka na zakończenie...")  
  
                jobWarzywa.join()  
                jobMieso.join()  
  
                logs.add("Szef kuchni: Wszyscy skończyli! Można podawać danie.")  
            }  
        }, modifier = Modifier.fillMaxWidth()) {  
            Text( text = "Rozpocznij gotowanie")  
        }  
    }  
}
```

W poprzednich przykładach włączaliśmy korutyny za pomocą **launch**. W tym przykładzie zobaczymy w jaki sposób odebrać **wynik działania asynchronicznego (async)**.



Asynchroniczność

Funkcja `suspend. delay(3000)` zawiesza korutynę na 3 sekundy **bez blokowania** wątku UI, a **następnie zwraca wynik** w postaci `String`.

```
@Composable
fun PasswordGeneratorScreen() {
    var password by remember { mutableStateOf( value = "...") }
    val scope = rememberCoroutineScope()

    suspend fun generatePassword(): String {
        delay( timeMillis = 3000)
        return "Kotlin-Is-Super-Secure-123"
    }

    Column(
        modifier = Modifier.fillMaxSize().padding( all = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text( text = "Generator Tajemniczego Hasła", style = MaterialTheme.typography.h2)
        Spacer( Modifier.height( height = 20.dp) )
        Button( onClick = {
            password = "Generowanie..."
            scope.launch {
                val deferredPassword: Deferred<String> =
                    async { generatePassword() }
                password = deferredPassword.await()
            }
        }) {
            Text( text = "Wygeneruj hasło")
        }
        Spacer( Modifier.height( height = 20.dp) )
        Text( text = password, fontSize = 20.sp, fontFamily = FontFamily.Serif)
    }
}
```

Zwraca wynik

Asynchroniczność

Funkcja `suspend. delay(3000)` zawiesza korutynę na 3 sekundy **bez blokowania** wątku UI, a **następnie zwraca wynik** w postaci `String`.

Uruchamia **główną** korutynę

```
@Composable
fun PasswordGeneratorScreen() {
    var password by remember { mutableStateOf( value = "...") }
    val scope = rememberCoroutineScope()

    suspend fun generatePassword(): String {
        delay( timeMillis = 3000)
        return "Kotlin-Is-Super-Secure-123"
    }

    Column(
        modifier = Modifier.fillMaxSize().padding( all = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text( text = "Generator Tajemniczego Hasła", style = Mater
        Spacer( Modifier.height( height = 20.dp))
        Button( onClick = {
            password = "Generowanie..."
            scope.launch {
                val deferredPassword: Deferred<String> =
                    async { generatePassword() }
                password = deferredPassword.await()
            }
        }) {
            Text( text = "Wygeneruj hasło")
        }
        Spacer( Modifier.height( height = 20.dp))
        Text( text = password, fontSize = 20.sp, fontFamily = Font
    }
}
```

Zwraca wynik

Asynchroniczność

Funkcja `suspend. delay(3000)` zawiesza korutynę na 3 sekundy **bez blokowania** wątku UI, a **następnie zwraca wynik** w postaci `String`.

Uruchamia **główną** korutynę

Wewnątrz głównej korutyny, `async` uruchamia nowe, zagnieżdżone zadanie w tle w celu wywołania `generatePassword()`. Zamiast wyniku, `async` **natychmiast** zwraca obiekt `Deferred`.

```
@Composable
fun PasswordGeneratorScreen() {
    var password by remember { mutableStateOf( value = "...") }
    val scope = rememberCoroutineScope()

    suspend fun generatePassword(): String {
        delay( timeMillis = 3000)
        return "Kotlin-Is-Super-Secure-123"
    }

    Column(
        modifier = Modifier.fillMaxSize().padding( all = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text( text = "Generator Tajemniczego Hasła", style = Mater
        Spacer( Modifier.height( height = 20.dp))
        Button( onClick = {
            password = "Generowanie..."
            scope.launch {
                val deferredPassword: Deferred<String> =
                    async { generatePassword() }
                password = deferredPassword.await()
            }
        }) {
            Text( text = "Wygeneruj hasło")
        }
        Spacer( Modifier.height( height = 20.dp))
        Text( text = password, fontSize = 20.sp, fontFamily = Font
    }
}
```

Zwraca wynik

Asynchroniczność

Funkcja `suspend`. `delay(3000)` zawiesza korutynę na 3 sekundy **bez blokowania** wątku UI, a **następnie zwraca wynik** w postaci `String`.

Uruchamia **główną** korutynę

Wewnątrz głównej korutyny, `async` uruchamia nowe, zagnieżdżone zadanie w tle w celu wywołania `generatePassword()`. Zamiast wyniku, `async` **natychmiast** zwraca obiekt `Deferred`.

`Deferred` jest to obiekt, który otrzymujesz **natychmiast po wywołaniu** `async`, zanim jeszcze właściwe obliczenia się zakończą. Podobnie jak `Job`, `Deferred` również **reprezentuje cykl życia** zadania.

@Composable

```
fun PasswordGeneratorScreen() {
    var password by remember { mutableStateOf( value = "...") }
    val scope = rememberCoroutineScope()

    suspend fun generatePassword(): String {
        delay( timeMillis = 3000)
        return "Kotlin-Is-Super-Secure-123"
    }

    Column(
        modifier = Modifier.fillMaxSize().padding( all = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text( text = "Generator Tajemniczego Hasła", style = Mater
        Spacer( Modifier.height( height = 20.dp))
        Button( onClick = {
            password = "Generowanie..."
            scope.launch {
                val deferredPassword: Deferred<String> =
                    async { generatePassword() }
                password = deferredPassword.await()
            }
        }) {
            Text( text = "Wygeneruj hasło")
        }
        Spacer( Modifier.height( height = 20.dp))
        Text( text = password, fontSize = 20.sp, fontFamily = Font
    }
}
```

Zwraca wynik

Asynchroniczność

Funkcja `suspend. delay(3000)` zawiesza korutynę na 3 sekundy **bez blokowania** wątku UI, a **następnie zwraca wynik** w postaci `String`.

Uruchamia **główną** korutynę

Wewnątrz głównej korutyny, `async` uruchamia nowe, zagnieżdżone zadanie w tle w celu wywołania `generatePassword()`. Zamiast wyniku, `async` **natychmiast** zwraca obiekt `Deferred`.

`Deferred` jest to obiekt, który otrzymujesz **natychmiast po wywołaniu** `async`, zanim jeszcze właściwe obliczenia się zakończą. Podobnie jak `Job`, `Deferred` również **reprezentuje cykl życia** zadania.

Korutyna `launch` **zawiesza się** w tym miejscu, czekając na zakończenie zadania. Gdy `async` zakończy pracę i dostarczy wynik, `await()` **"odpakowuje"** go i **przypisuje do zmiennej stanu** `password`, co powoduje finalną aktualizację UI (rekompozycję).

@Composable

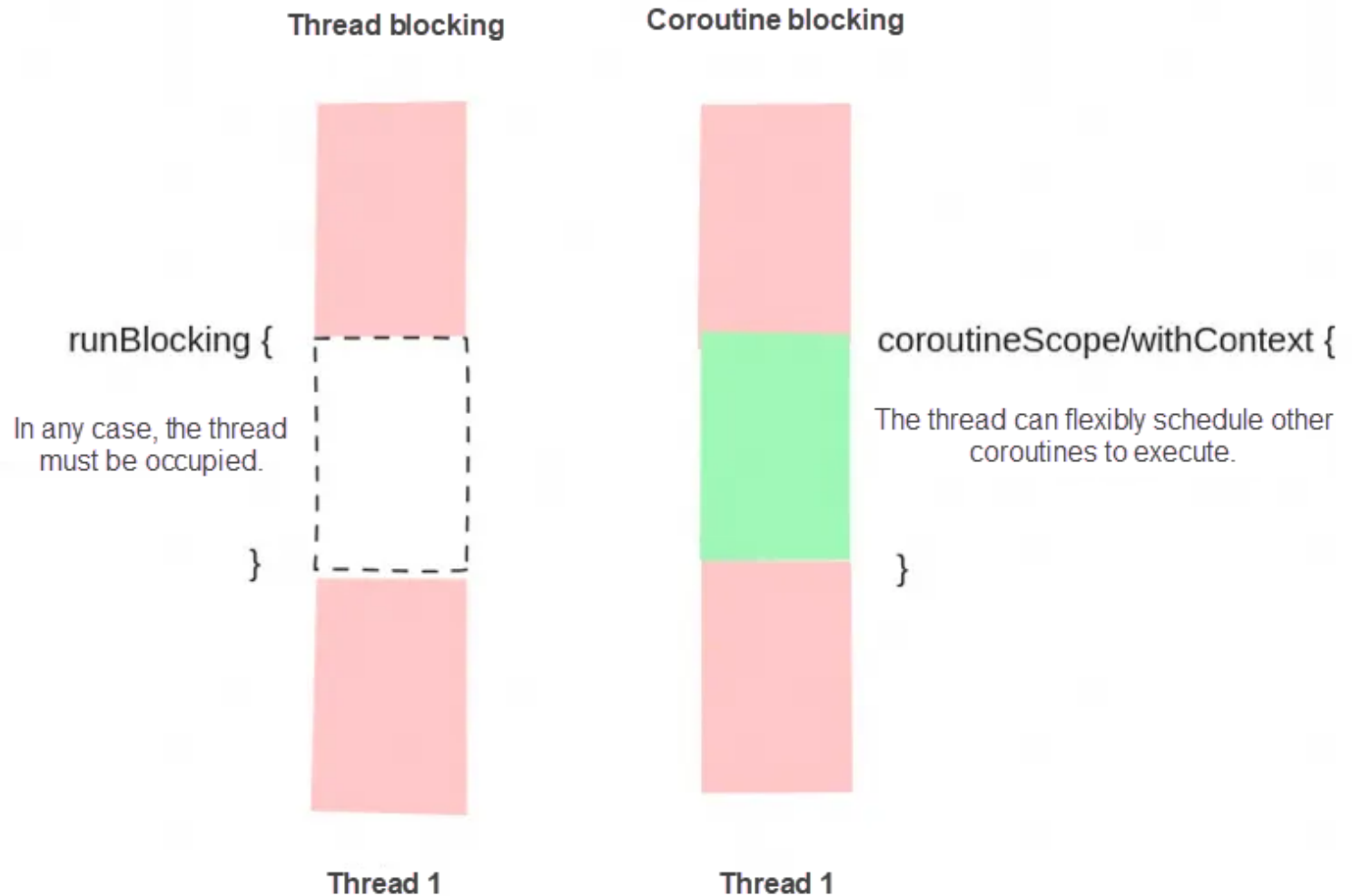
```
fun PasswordGeneratorScreen() {
    var password by remember { mutableStateOf( value = "...") }
    val scope = rememberCoroutineScope()

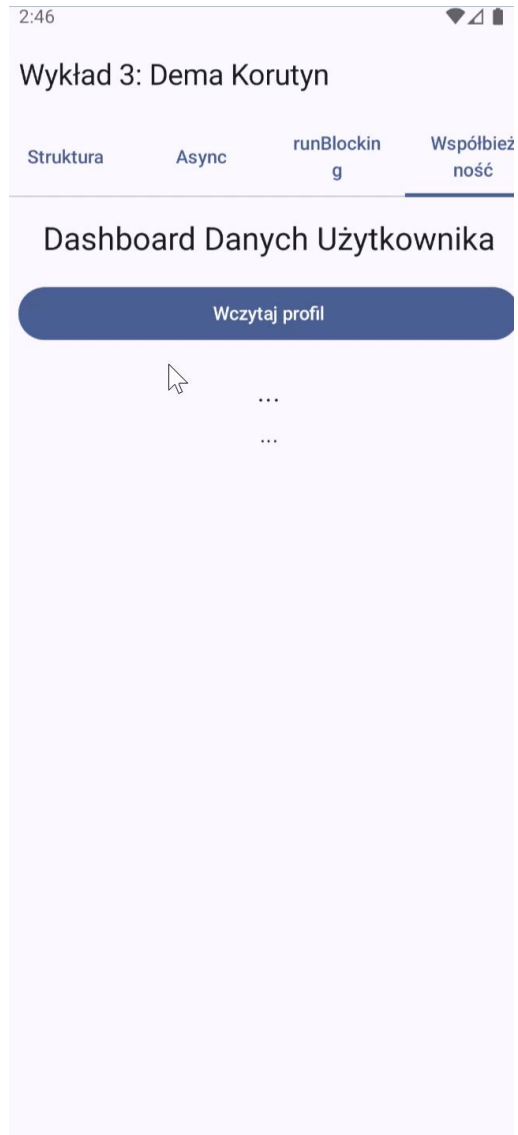
    suspend fun generatePassword(): String {
        delay( timeMillis = 3000)
        return "Kotlin-Is-Super-Secure-123"
    }

    Column(
        modifier = Modifier.fillMaxSize().padding( all = 16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text( text = "Generator Tajemniczego Hasła", style = Mater
        Spacer( Modifier.height( height = 20.dp))
        Button( onClick = {
            password = "Generowanie..."
            scope.launch {
                val deferredPassword: Deferred<String> =
                    async { generatePassword() }
                password = deferredPassword.await()
            }
        }) {
            Text( text = "Wygeneruj hasło")
        }
        Spacer( Modifier.height( height = 20.dp))
        Text( text = password, fontSize = 20.sp, fontFamily = Font
```

Zwraca wynik

Blokowanie





```
private data class ProfileUiState(  
    val isLoading: Boolean = false,  
    val userData: String = "...",  
    val userActivity: String = "...",  
    val time: String = ""  
)
```

@Composable

```
fun UserProfileDashboardScreen() {  
    var uiState by remember { mutableStateOf( value = ProfileUiState()) }  
    val scope = rememberCoroutineScope()  
  
    suspend fun fetchUserDetails(): String { delay( timeMillis = 2000)  
        return "Użytkownik: Ada" }  
    suspend fun fetchUserActivity(): String { delay( timeMillis = 3000)  
        return "Ostatnia aktywność: Dodała post" }  
  
    Column(  
        modifier = Modifier.fillMaxSize().padding( all = 16.dp),  
        horizontalAlignment = Alignment.CenterHorizontally  
    ) {  
        Text( text = "Dashboard Danych Użytkownika", style = MaterialTheme.typography.h1  
        Spacer( Modifier.height( height = 16.dp))  
        Button( onClick = {  
            scope.launch {  
                uiState = ProfileUiState( isLoading = true)  
                val startTime = System.currentTimeMillis()  
  
                val detailsDeferred = async { fetchUserDetails() }  
                val activityDeferred = async { fetchUserActivity() }  
  
                val details = detailsDeferred.await()  
                val activity = activityDeferred.await()  
  
                val totalTime = System.currentTimeMillis() - startTime  
                uiState = ProfileUiState(  
                    isLoading = false,  
                    userData = details,  
                    userActivity = activity,  
                    time = "Całkowity czas: ${totalTime}ms"  
                )  
            }  
        })  
    }  
}
```