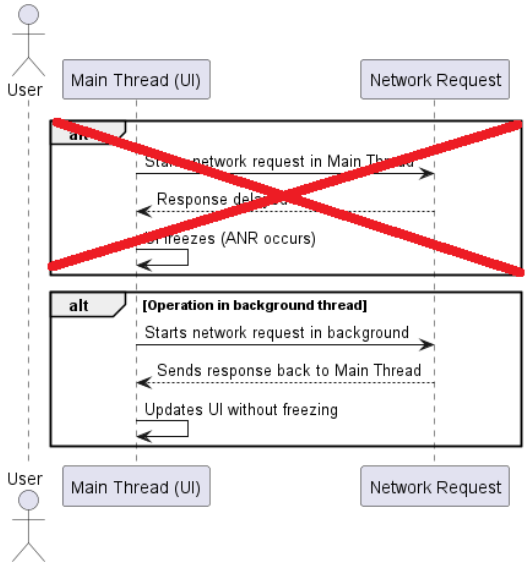
Sieć, wątki.

W Androidzie główny wątek (Main Thread/ UI Thread) obsługuje aktualizacje interfejsu użytkownika takie jak rysowanie elementów czy obsługa kliknięć.

Wykonanie operacji sieciowej w głównym wątku powoduje zatrzymanie obsługi UI, co prowadzi do zamrożenia aplikacji (tzw. Application Not Responding - ANR).

łączenie operacji sieciowych w głównym wątku generuje wyjątek **NetworkOnMainThreadException**.

Operacje sieciowe powinny być wykonywane w tle !



Rozwiązanie:

AsyncTask (starsze rozwiązanie).

Threads (manualne zarządzanie wątkami).

Coroutines (nowe rozwiązanie).

Do pobrania książek z przykładu z poprzednich zajęć (projekt Gutenberg: https://www.gutenberg.org/ebooks/search/?sort_order=downloads)
można użyć biblioteki okhttp3
implementation ("com.squareup.okhttp3:okhttp:4.12.0")

Biblioteka zapewnia:

- Wysyłanie i odbieranie żądań HTTP/HTTPS.
- Zarządzanie połączeniami.
- Obsługa nagłówków, czasu oczekiwania, ciasteczek, itd.
- Przetwarzanie odpowiedzi, w tym strumieniowanie danych.

Przed użyciem sieci w aplikacji należy dodać odpowiednie uprawnienie w pliku manifestu:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Funkcja, która będzie pobierała zawartość książki po podaniu url:

```
import okhttp3.OkHttpClient
import okhttp3.Request
import java.io.IOException

fun getBookOKHttp(url: String): String {
    val client = OkHttpClient()
    val request = Request.Builder()
        .url(url)
        .build()

    return try {
        client.newCall(request).execute().use { response ->
            if (!response.isSuccessful) {
                throw IOException("error: ${response.code}")
            }
            response.body?.string().orEmpty()
        }
    } catch (exc: Exception) {
        "Exception: ${exc.message}"
    }
}
```

…

`val client = OkHttpClient()`- Tworzy klienta http.

```
val request = Request.Builder()
        .url(url)
        .build()
```
- ustawia url i buduje żądanie

`client.newCall(request).execute()` tworzy nowy obiekt wywołania (ang. Call) na podstawie wcześniej przygotowanego żądania HTTP (Request).
`client.newCall(request).execute().use`.-wysyła żądanie i czeka na odpowiedź (`.use` automatycznie zamknie zasób-odpowiedź response)

`response.body?.string().orEmpty()`- -pobiera body odpowiedzi, zamienia na String i zwraca(lub zwraca pusty)

Wątki

Wystarczy podać odpowiedni url oraz uruchomić pobieranie w nowym wątku, po pobraniu kontent książki zostanie zaktualizowany a wraz z nim LazyColumn:

```
@Composable
fun BookComposable(modifier: Modifier) {
    val url = "https://www.gutenberg.org/cache/epub/11/pg11.txt"
    var content by remember { mutableStateOf("...") }

    Column(modifier=modifier){
    Button(
            onClick = {
        Thread{
            content = getBookOKHttp(url)
        }.start()
    }) {
```

```
        Text(text = "Download book with Thread")
    }

    LazyColumn {
        items(content.lines()){
            line -> Text(text = line)
        }
    }
    }
}
```

Utworzenie i wykonanie wątku polega na użyciu klasy Thread, Runnable przekazujemy przez lambdę:

```
        Thread{
            content = getBookOKHttp(url)
        }.start()
```

Minusy:

-Każdy wątek zajmuje dużo pamięci, co jest szczególnie istotne w urządzeniach mobilnych z ograniczonymi zasobami.

-Synchronizacja danych między wątkami wymaga użycia mechanizmów takich jak synchronized, Handler, czy Lock.

-Wątki mogą prowadzić do wycieków pamięci, jeśli nie zostaną odpowiednio zatrzymane podczas zmiany stanu aktywności

-Wątki wykonują kod w sposób blokujący, co może prowadzić do niskiej wydajności.

Coroutines

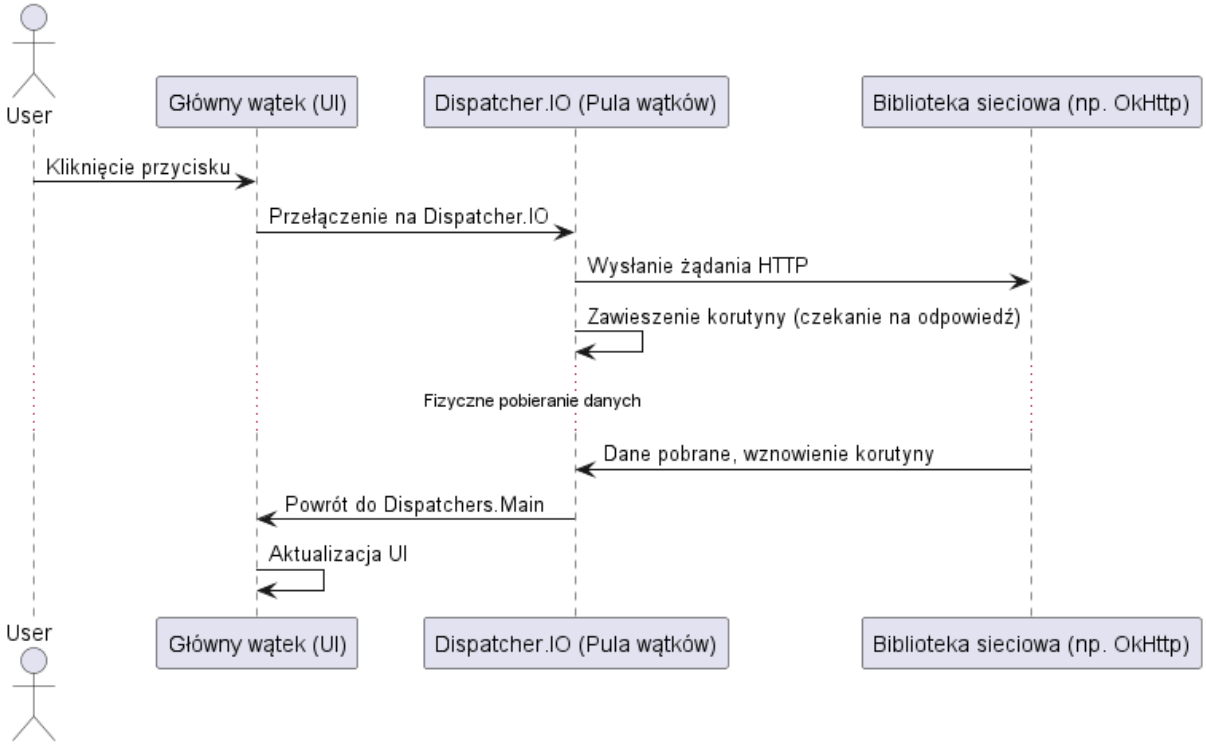
Korutyny działają podobnie do wątków, umożliwiając współbieżne wykonywanie zadań. Nie są jednak na stałe przypisane do jednego wątku. Korutyna może rozpocząć swoje działanie w jednym wątku, następnie zostać zawieszona, a po pewnym czasie wznowić pracę w innym wątku.

Podczas zawieszenia korutyna nie blokuje wątku, na którym została uruchomiona. Wątek wraca do puli i może być wykorzystany przez inne zadania. Kiedy zawieszenie korutyny się kończy, jej działanie jest wznawiane w dowolnym wolnym wątku z tej samej puli.

Korutyny w Kotlinie są integralną częścią języka, a nie funkcją zarządzaną przez system operacyjny. Dzięki temu system operacyjny nie musi ich monitorować ani planować ich pracy.

Podczas zawieszenia korutyny środowisko wykonawcze Kotlin automatycznie przełącza się na inną korutynę, która może kontynuować swoje działanie.

Korutyny są znacznie lżejsze niż wątki, zajmując jedynie minimalną ilość pamięci. Dzięki temu można uruchomić ich dużo więcej niż wątków, co pozwala na osiągnięcie wysokiego poziomu współbieżności przy niskich kosztach zasobów.



***implementation* ("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.9.0")**

– zależności dla Coroutine.

//

```
import androidx.compose.runtime.rememberCoroutineScope
```

```
val coroutineScope =  rememberCoroutineScope()
```

W Jetpack Compose, rememberCoroutineScope() tworzy i zwraca CoroutineScope, który jest powiązany z cyklem życia danego Composable. Gdy Composable jest usuwany z hierarchii (np. użytkownik przechodzi na inny ekran), wszystkie korutyny uruchomione w tym zakresie są anulowane automatycznie.

Analogicznie jak w wypadku wątku można uruchomić funkcję za pomocą korutyny:

```
import kotlinx.coroutines.launch
```

```
Button(
    onClick = {
        coroutineScope.launch {
            try {
                content= getBookOKHttpForCoroutine(url)
            } catch (e: Exception) {
                content = "Exception ${e.message }"
            }
        }

    }
) {Text(text = "Download book with Coroutine") }
```

Funkcja, użyta w korutynie będzie różniła się od poprzedniej użytej w wątku:

```
suspend fun getBookOKHttpForCoroutine(url: String): String = withContext(Dispatchers.IO) {
    val client = OkHttpClient()
    val request = Request.Builder()
        .url(url)
        .build()
    try {
        client.newCall(request).execute().use { response ->
            if (!response.isSuccessful) {
                throw IOException("error: ${response.code}")
            }
            response.body?.string().orEmpty()
        }
    } catch (exc: Exception) {
        "Exception: ${exc.message}"
    }
}
```

suspend - Funkcja może zawiesić swoje działanie i wstrzymać korutynę, w której jest wywoływana.

withContext(Dispatchers.IO) - Przełącza wykonanie funkcji na **Dispatcher.IO**, co oznacza, że kod będzie wykonywany w tle, w wątku zoptymalizowanym pod operacje wejścia/wyjścia.
withContext jest funkcją zawieszającą, która zwraca wynik wyrażenia zawartego w jej bloku ({}).

Dispatcher	Opis
Dispatchers.Main	Główny wątek (UI Thread)
Dispatchers.IO	Wątki zoptymalizowane dla operacji I/O
Dispatchers.Default	Wątki zoptymalizowane dla CPU
Dispatchers.Unconfined	Brak przypisania do wątku

Aby pobrać wiele książek jednocześnie można uruchomić wiele korutyn jednocześnie:

```
suspend fun awaitForAllBooks(urls: List<String>): List<String> {
    return coroutineScope {
        urls.map { url ->
            async {
                getBookOKHttpForCoroutine(url)
            }
        }.awaitAll()
    }
}
```

Zamiast jednego url, można podać wiele adresów w formie listy, również zamiast jednej odpowiedzi z zawartością odpowiedzi, można zwrócić wiele zawartości odpowiedzi.

coroutineScope - lokalny zakres korutyny, w którym zarządzane są korutyny uruchomione wewnątrz funkcji.

urls.map { url ->-Dla każdego elementu listy adresów URL tworzona jest nowa korutyna za pomocą async.

async tworzy korutynę, która zwraca wynik typu Deferred<T> – obiekt reprezentujący wynik asynchronicznej operacji.

awaitAll() - Oczekuje na zakończenie wszystkich korutyn w kolekcji.

Zwraca listę wyników tych korutyn.

Przykład z 2 książkami – po pobraniu zostaną zaktualizowane 2 LazyColumn:

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth

import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Modifier
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.async
import kotlinx.coroutines.awaitAll
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.launch

@Composable
fun BookComposable(modifier: Modifier) {
   val url = "https://www.gutenberg.org/cache/epub/11/pg11.txt"
   var content by remember { mutableStateOf("...") }
   val coroutineScope =  rememberCoroutineScope()

   Column(modifier=modifier){
   Button(
         onClick = {
      Thread{
         content = getBookOKHttp(url)
      }.start()
   }) {
      Text(text = "Download book with Thread")
   }
   Button(
      onClick = {
         coroutineScope.launch {
            try {
               content= getBookOKHttpForCoroutine(url)
            } catch (e: Exception) {
               content = "Exception ${e.message }"
            }
         }

      }
   ) {Text(text = "Download book with Coroutine") }
```

```
    LazyColumn {
        items(content.lines()){
            line -> Text(text = line)
        }
    }
    }
}


@Composable
fun BookComposableAsync(modifier: Modifier) {
    val url = "https://www.gutenberg.org/cache/epub/11/pg11.txt"
    val url2 = "https://www.gutenberg.org/cache/epub/84/pg84.txt"
    var content by remember { mutableStateOf("...") }
    var content2 by remember { mutableStateOf("...") }
    val coroutineScope =  rememberCoroutineScope()

    Column(modifier=modifier){
        Button(
            onClick = {
                coroutineScope.launch {
                    try {
                        val books = awaitForAllBooks(listOf(url,url2))
                        content= books[0]
                        content2= books[1]

                    } catch (e: Exception) {
                        content = "Exception ${e.message }"
                        content2 = "Exception ${e.message }"
                    }
                }

            }
        ) {Text(text = "Download book with Coroutine ascync") }
        Card(modifier = Modifier
            .fillMaxWidth()
            .weight(1f)
            .padding(8.dp)){
        LazyColumn{
            items(content.lines()){
                    line -> Text(text = line)
            }
        }
        }
        Card(modifier = Modifier
            .fillMaxWidth()
            .weight(1f)
            .padding(8.dp)) {
            LazyColumn(
                modifier = Modifier
                    .fillMaxWidth()
                    .weight(1f)
            ) {
                items(content2.lines()) { line ->
                    Text(text = line)
                }
            }
        }
    }
```

```
}

suspend fun awaitForAllBooks(urls: List<String>): List<String> {
    return coroutineScope {
        urls.map { url ->
            async {
                getBookOKHttpForCoroutine(url)
            }
        }.awaitAll()
    }
}
```

Zadanie:
Zmodyfikować poprzednie zadanie o pobranie prawdziwych plików z książkami. Należy pobrać wszystkie pliki jednocześnie używając courutine i przesłać przez broadcast dane dotyczące wszystkich książek – tytuł, liczba słów, liczba wyrazów, najczęściej pojawiające się słowo.
Tytuł można pobrać z książek z projektu Gutenberg (pierwsza linia).
Dodatkowo wyszukując najczęściej występującego wyrazu należy przefiltrować słowa z książki pomijając słowa ze zbioru stopWords. Zbiór tych słów znajduje się w pliku Download.kt

Bez wysyłania powiadomień (notifications), bez sprawdzania czy książka jest już w tabeli.

UWAGA!! W serwisie CoroutineScope należy utworzyć za pomocą konstruktora,
ponieważ Serwis działa poza interfejsem użytkownika.

```
private val coroutineScope = CoroutineScope(Dispatchers.IO)
```

zamiast:

~~val coroutineScope = rememberCoroutineScope()~~

```
Należy pamiętać o anuluwaniu korutyn przy zatrzymaniu serwisu:
override fun onDestroy() {
    coroutineScope.cancel()
    super.onDestroy()
}
```