Laboratorium 3a – Metody wykonywania zadań w tle (na przykładzie komunikacji sieciowej)

W aplikacjach na Androida kluczowe jest, aby długotrwałe operacje (np. pobieranie danych z sieci, operacje na bazie danych, złożone obliczenia) wykonywać poza głównym wątkiem interfejsu użytkownika (UI thread). Zapobiega to blokowaniu UI i powstawaniu błędów ANR (Application Not Responding). W nowszych wersjach systemu zamiast komunikatu ANR zgłaszany jest wyjątek i aplikacja jest przerywana (chodzi o wymuszenie na programistach poprawienia aplikacji).

Klasa Log – debugowanie aplikacji

W trakcie pracy nad programem, szczególnie wielowątkowym przydatna jest możliwość wyprowadzania pewnych informacji, tak aby programista mógł kontrolować działanie programu. Klasa Log pozwala na proste wysyłania komunikatów debugowania do *tzw. LogCat*. Klasa posiada wiele metod statycznych. Najważniejsze z nich to.:

- `v()` - wysyłanie szczegółowych informacji (od verbose - gadatliwy),
- `d()` - wysyłanie informacji służących do debugowania
- `i()` - wysyłanie zwykłych informacji
- `w()` - wysyłanie ostrzeżeń (od warning - ostrzeżenie)

Posiadają one wiele wersji. Parametrami najprostszych wersji są: etykieta tekstowa (identyfikuje źródło komunikatu) oraz komunikat tekstowy.

Wykonywanie krótkich zadań w tle

Jednym ze standardowych i efektywnych sposobów zarządzania wątkami w tle jest użycie **pul wątków**. Użycie klas `Executors` i `ExecutorService` to zalecany sposób obsługi zadań w tle w nowoczesnych aplikacjach Android, zapewniający responsywność interfejsu użytkownika i efektywne wykorzystanie zasobów systemowych.

Pula Wątków (`ExecutorService`)

Pula wątków to zarządzana kolekcja wątków roboczych. Zamiast tworzyć nowy wątek dla każdego zadania (co jest operacją kosztowną pod względem zasobów i czasu), pula wątków ponownie wykorzystuje istniejące, nieaktywne wątki do wykonania nowych zadań. Gdy zadanie zostaje przekazane do puli, jeden z dostępnych wątków je podejmuje. Jeśli wszystkie wątki są zajęte, zadanie może czekać w kolejce. Główne zalety pul wątków to:

1. **Wydajność:** Redukcja narzutu związanego z tworzeniem i niszczeniem wątków.
2. **Zarządzanie zasobami:** Ograniczenie liczby jednocześnie działających wątków, co zapobiega wyczerpaniu zasobów systemowych.
3. **Uproszczenie zarządzania:** Łatwiejsze zarządzanie cyklem życia wątków.

Interfejsem reprezentującym pulę wątków w Javie jest `java.util.concurrent.ExecutorService`.

Klasa `Executors`

Klasa `java.util.concurrent.Executors` to klasa narzędziowa (utility class) i fabryka (factory class), która dostarcza statycznych metod do tworzenia różnych typów instancji `ExecutorService` (czyli pul wątków) oraz innych powiązanych obiektów. Upraszcza ona proces konfiguracji i tworzenia pul wątków. Najczęściej używane metody fabrykujące to:

- `Executors.newFixedThreadPool(int nThreads)`: Tworzy pulę wątków o stałej, określonej liczbie wątków. Jeśli wszystkie wątki są zajęte, nowe zadania oczekują w kolejce. Dobra dla zadań intensywnie wykorzystujących CPU.
- `Executors.newCachedThreadPool()`: Tworzy pulę, która dynamicznie tworzy nowe wątki w

miarę potrzeb i usuwa wątki, które były nieaktywne przez pewien czas (domyślnie 60 sekund). Odpowiednia dla dużej liczby krótkotrwałych zadań asynchronicznych. Może jednak prowadzić do stworzenia bardzo dużej liczby wątków, jeśli napływa wiele zadań naraz.

- `Executors.newSingleThreadExecutor()`: Tworzy pulę zawierającą tylko jeden wątek. Gwarantuje to, że zadania będą wykonywane sekwencyjnie, jedno po drugim, w kolejności ich dodania. Przydatne, gdy kolejność wykonania jest istotna lub gdy potrzebujemy synchronizacji dostępu do zasobu bez jawnych blokad.

Użycie puli wątków polega na:

1. Stworzeniu instancji `ExecutorService` używając jednej z metod klasy `Executors`.
2. Przekazania zadania do wykonania (jako obiekty implementujące `Runnable` lub `Callable`) za pomocą metod `execute(Runnable)` lub `submit(Runnable/Callable)`.
3. Po zakończeniu pracy z pulą wątków, należy pamiętać o jej zamknięciu za pomocą `shutdown()` (czeka na zakończenie bieżących zadań) lub `shutdownNow()` (próbuje natychmiast zatrzymać wszystkie zadania), aby zwolnić zasoby.

Interfejs `Future` – Reprezentacja Przyszłego Wyniku

Kiedy przekazujemy zadanie typu `Callable<V>` do wykonania asynchronicznego za pomocą metod `submit()` interfejsu `ExecutorService`, metody te nie zwracają bezpośrednio wyniku zadania (ponieważ zadanie może jeszcze nie być zakończone). Zamiast tego zwracają obiekt implementujący interfejs `java.util.concurrent.Future`. `Future` dostarcza standardowego mechanizmu do zarządzania wynikiem operacji asynchronicznych, umożliwiając pobranie wyniku, sprawdzenie statusu zadania lub jego anulowanie. Jest uchwytem lub obietnicą dostarczenia wartości, która może być dostępna w przyszłości, gdy zadanie w tle zakończy swoje działanie.

Kluczowe metody `Future`:

Interfejs `Future` udostępnia następujące metody:

1. `V get()` - **czeka (blokuje bieżący wątek)**, aż zadanie powiązane z tym `Future` zakończy swoje wykonanie, a następnie zwraca jego wynik ((a) jeśli zadanie zakończyło się pomyślnie, zwraca obliczoną wartość typu `V`, (b) jeśli zadanie zostało anulowane, zgłasza `CancellationException`, (c) jeśli zadanie zakończyło się wyjątkiem, `get()` opakowuje ten wyjątek w `ExecutionException` i zgłasza go, (d) jeśli bieżący wątek zostanie przerwany podczas oczekiwania, `get()` zgłasza `InterruptedException`, (e) **nigdy nie należy wywoływać `get()` bez limitu czasowego w głównym wątku UI – blokuje wątek**
2. `V get(long timeout, TimeUnit unit)`: Działa podobnie do `get()`, ale czeka tylko przez określony czas. Jeśli wynik nie będzie dostępny w zadanym czasie, metoda zgłasza `TimeoutException`. **Nie zaleca się jej używania w wątku UI**.
3. `boolean isDone()`: Zwraca `true`, jeśli zadanie zostało zakończone (normalnie, przez wyjątek lub przez anulowanie), w przeciwnym razie `false`. Metoda **nie blokuje** i jest bezpieczna do wywołania w dowolnym momencie, aby sprawdzić status zadania.
4. `boolean cancel(boolean mayInterruptIfRunning)`: Próbuje anulować wykonanie zadania. Zwraca `true`, jeśli próba anulowania się powiodła (np. zadanie nie zostało jeszcze rozpoczęte lub udało się je przerwać), `false` w przeciwnym wypadku (np. zadanie już się zakończyło). Parametr `mayInterruptIfRunning` wskazuje, czy wątek wykonujący zadanie powinien zostać przerwany (`Thread.interrupt()`), jeśli zadanie już działa.
5. `boolean isCancelled()`: Zwraca `true`, jeśli zadanie zostało anulowane przed jego normalnym zakończeniem.

Klasa `Handler` – Komunikacja z Wątkiem UI

Jak już wspomniano, operacje na interfejsie użytkownika (np. zmiana tekstu w `TextView`, aktualizacja `ProgressBar`, wyświetlenie `Toast`) muszą być wykonywane **wyłącznie** w głównym wątku aplikacji/wątku UI. Próba modyfikacji UI z innego wątku (np. utworzonego przez `ExecutorService`) zakończy się błędem. Aby przekazać wynik operacji wykonanej w tle (np. pobrane dane) z powrotem do wątku UI, aby bezpiecznie zaktualizować interfejs można wykorzystać klasę `android.os.Handler`.

`Handler` to mechanizm w systemie Android, który pozwala na:

1. **Wysyłanie zadań:** Umożliwia wysyłanie obiektów `Runnable` lub `Message` do przetworzenia w przyszłości.
2. **Kolejkowanie zadań:** Dodaje te zadania do kolejki komunikatów (`MessageQueue`) powiązanej z konkretnym wątkiem.
3. **Wykonywanie zadań:** Zadania te są ostatecznie wykonywane na tym wątku, z którym `Handler` jest skojarzony.

Wykorzystanie klasy `Handler` do Aktualizacji UI

Klasa `Handler` jest kluczowym elementem do komunikacji między wątkami w Androidzie, szczególnie przydatnym do przekazywania wyników z zadań w tle w celu aktualizacji interfejsu użytkownika w bezpieczny sposób, zgodny z regułami systemu.

Główny wątek aplikacji Android ma automatycznie utworzoną i działającą pętlę komunikatów (`Looper`) wraz z kolejką (`MessageQueue`). Można stworzyć instancję `Handler`, która będzie powiązana właśnie z tą pętlą i kolejką wątku UI. Dzięki temu, zadania wysłane za pomocą tego `Handlera` (nawet z innego wątku) zostaną ostatecznie wykonane w wątku UI.

Użycie Handlera pole na:

1. Utworzeniu `Handlera`

   ```
   Handler mainHandler = new Handler(Looper.getMainLooper());
   ```

2. Udostępnieniu `Handlera` - instancja `Handlera` musi być dostępna dla kodu wykonywanego w wątku tła
3. Wysłanie zadań z wątku działającego w tle - wewnątrz kodu działającego w tle używamy metody `post()` na instancji `Handlera` powiązanego z wątkiem UI:

   ```
   mainHandler.post(new Runnable() {
       @Override
       public void run() {
           // Ten kod wykona się w wątku UI
           textView.setText(result); // Bezpieczna aktualizacja UI
       }
   });
   ```

Szkielet klasy wykonującej krótkie zadania w tle

Poniżej zamieszczono szablon klasy pozwalającej na wykonywanie, krótkich zadań w tle.

```
// szablon klasy ShortTask
public class ShortTask {

    private static final String TAG = ShortTask.class.getSimpleName();
```

```
private final ExecutorService executorService;
private final Handler mainThreadHandler;
private Future<ResultType> future;

public FileShortTask() {
    // utworzenie puli 2 wątków
    executorService = Executors.newFixedThreadPool(2);
    // utworzenie Handlera do wysyłania zadań do wątku UI
    mainThreadHandler = new Handler(Looper.getMainLooper());
}

// wywołanie zwrotne do przekazywania wyników
public interface ResultCallback {
    void onSuccess(ResultType result);
    void onError(Throwable throwable);
    // można dodać więcej metod np. do przekazywania informacji o postępie
}

// do metody trzeba przekazać wywołanie zwrotne do odbierania wyników i wymagane
// parametry
public Future<ResultType> executeTask(ResultCallback callback, String param) {
    // anulowanie bieżącego zadania
    if (future != null && !future.isDone()) {
        future.cancel(true);
    }

    // tworzenie zadania, które odczyta wynik zadania wykonanego w tle i
    // przekaże go do głównego wątku
    Runnable completionTask = new Runnable() {
        @Override
        public void run() {
            try {
                // czekanie na wynik (jeżeli wyniku nie ma blokuje wątek
                // wywołujący)
                ResultType result = future.get();
                // przekazanie wyniku
                callback.onSuccess(result);
            } catch (CancellationException e) {
                // ewentualna reakcja na anulowanie
            } catch (Exception e) {
                // przekazanie informacji o błędach
                callback.onError(e);
            }
        }
    };

    // tworzenie zadania wykonywanego w tle (zadanie zwraca wynik)
    Callable<ResultType> asyncTask = new Callable<ResultType>() {
        @Override                // wykonywane zadania mogą powodować wyjątki
        public ResultType call() throws Exception {
            // tutaj umieścić zadanie do wykonania w tle
            // ...
            // wynik jest gotowy - wysyłamy do wykonania zadanie przekazujące wynik
            mainThreadHandler.post(completionTask);
            // zakończenie przyszłości (poprzez ustawienie wyniku)
            return result;
        }
    };

    // wysłanie zadania do wykonania przez pulę wątków
    future = executorService.submit(asyncTask);
```

```
        // zwrócenie przyszłości (może być użyta do anulowania zadania)
        return future;
    }

    // tą metodę wywołać gdy aktywność lub fragment są niszczone
    public void shutdown() {
        // anulowanie wykonywanego zadania
        if (future != null && !future.isDone()) {
            future.cancel(true); // true - przerwanie zadania
        }
        // zamknięcie puli wątków
        executorService.shutdown();
    }
}
```

Użycie `Handlera` polega na:

1. Utworzeniu obiektu klasy `ShortTask` w klasie aktywności.
2. Utworzeniu wywołania zwrotnego reagującego na wyniki (i aktualizującego UI) a także na błędy.
3. Wywołaniu metody `executeTask(...)`

Komunikacja sieciowa

Istnieje wiele rodzajów komunikacji sieciowej. Za przykład posłuży tworzenie i korzystanie z protokołu HTTP. W nowszych wersjach Androida wymagane jest użycie połączeń HTTPS (szyfrowanych). Użycie połączenia nieszyfrowanego wymaga utworzenia pliku XML określającego wyjątki dla domen nieobsługujących HTTPS. Do tworzenia połączeń HTTPS można wykorzystać klasę `HttpsUrlConnection` (pochodną `HttpUrlConnection`). Typowy przebieg połączenia HTTP:

- nawiązanie połączenia za pomocą metody statycznej `URL.openConnection()` (zwraca obiekt połączenia),
- przygotowanie żądania do serwera,
- opcjonalne przygotowanie treści żądania (request body). W tym celu należy wywołać `setDoOutput(true)` (włączenie możliwości wysyłania danych) i zapisać dane do strumienia zwróconego przez `getOutputStream()`,
- odczytanie odpowiedź serwera ze strumienia zwróconego przez `getInputStream()`. Możliwe jest odczytanie dodatkowych informacji takich jak typ zawartości, ilość danych itp.,
- zamknięcie połączenia za pomocą `disconnect()`.

Poniżej przedstawiono fragment kodu nawiązującego połączenie HTTPS zadanym adresem URL. Po nawiązaniu połączenia odczytywany jest rozmiar dostępnej treści oraz jej typ. Na koniec połączenie jest zamykane. Większość operacji sieciowych może zgłaszać wyjątki.

```
HttpsURLConnection connection = null;
FileInfo fileInfo = null;
try {
    URL url = new URL(fileUrl);
    connection = (HttpsURLConnection) url.openConnection();
    connection.setRequestMethod("GET");
    // custom FileInfo class
    fileInfo = new FileInfo(connection.getContentLength(),
                            connection.getContentType());
} finally {
    if (connection != null)
        connection.disconnect();
```

```
}
```

Uprawnienia aplikacji

W Androidzie aplikacje nie mają pełnej swobody działania. Domyślnie każda aplikacja otrzymuje bardzo podstawowy zestaw uprawnień niezbędnych do działania podstawowych funkcji. Z takich uprawnień korzystały aplikacje #1 i #2. Pozostałe uprawnienia dzielą się na poziomy ochrony (ang. protection level). Uprawnienia, które mają poziom ochrony normal np. `"android.permission.INTERNET"` muszą być wymienione w manifeście aplikacji. Jeżeli nie zostaną wymienione przy próbie wykonania operacji zastrzeżonej (np. dostępu do sieci) aplikacja zgłosi wyjątek. W manifeście aplikacji muszą znajdować się znaczniki <uses-permission> określające o jakie uprawnienia prosi aplikacja (nie należy go mylić ze znacznikiem <permission> służącym do tworzenia własnych uprawnień). Uprawnienia z normalnym poziomem ochrony nie wymagają zgody użytkownika – są przyznawane automatycznie przy instalacji (pod warunkiem, że są wymienione w manifeście).

Poniżej pokazano fragment manifestu aplikacji określającego, że aplikacja wymaga uprawnienia INTERNET, które pozwala na dostęp do sieci (w tym do sieci lokalnej, nie tylko do internetu).

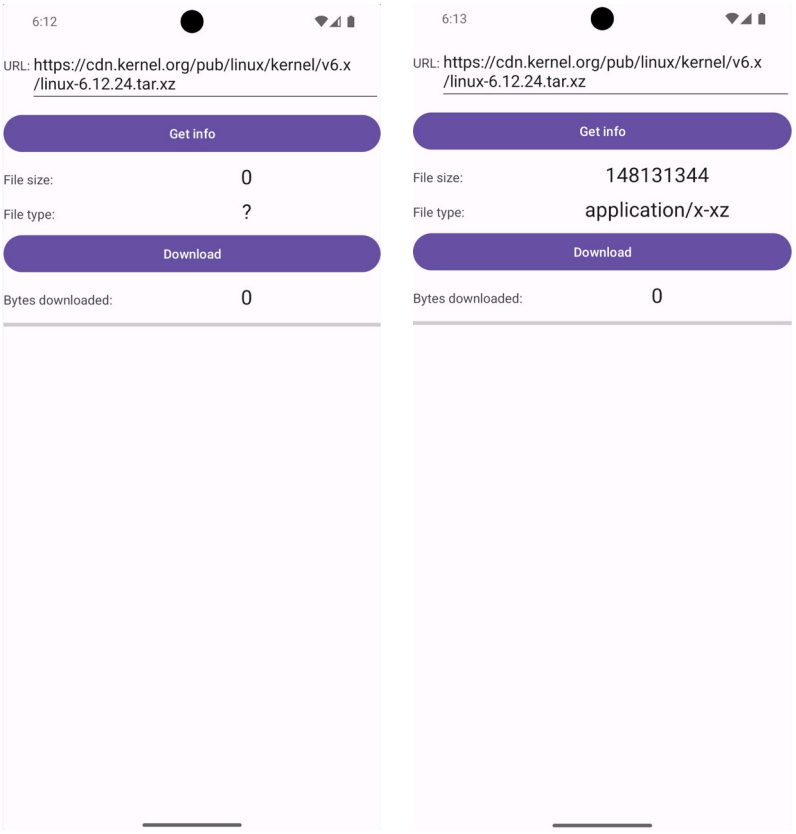
```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

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

    <!-- element application itd. -->
</manifest>
```

Istnieją również uprawnienia z poziomem ochrony „niebezpieczny” (ang. protection level dangerous). Wymagają one uzyskania zgody użytkownika w trakcie działania programu. Tymi uprawnieniami zajmiemy się w kolejnej części laboratorium

Ćwiczenie 3a



Stwórz aplikację wyglądającą podobnie do tej przedstawionej na rysunku. Aplikacja

- powinna umożliwiać wprowadzenie adresu URL w polu tekstowym (najlepiej ustawić też domyślny adres do testowania jako wartość początkową pola tekstowego);
- po naciśnięciu przycisku *Pobierz informacje* aplikacja ma uruchamiać zadanie pobierania informacji o pliku w tle (wykorzystaj mechanizmy opisane w instrukcji), które
  - nawiąże połączenie z serwerem
  - odczyta z połączenia typ i rodzaj pliku
  - wyświetli informacje o pliku (rozmiar, typ) w odpowiednich elementach
- dodaj walidację wprowadzonego adresu (powinien zaczynać się od `https://`)
- w przypadku błędu aplikacja powinna wyświetlać `Toast`

Pamiętaj, o dodaniu do manifestu informacji o tym, że aplikacja korzysta z dostępu do sieci (element `<uses-permission>` i uprawnienie `INTERNET`, w manifeście aplikacji). W przeciwnym razie próba nawiązania połączenia skończy się zgłoszeniem wyjątku.

Obsługa przycisku *Pobierz plik*, etykiety z liczbą pobranych bajtów oraz paska postępu zostanie dodana w kolejnych ćwiczeniach.