O'REILLY®

Android

Aplikacje wielowątkowe
Techniki przetwarzania

WYKORZYSTAJ W PEŁNI POTENCJAŁ WĄTKÓW!

HELION

Anders Göransson

Tytuł oryginału: Efficient Android Threading

Tłumaczenie: Lech Lachowski

ISBN: 978-83-246-9614-7

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/andraw
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/andraw.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Komunikacja wątków

W aplikacjach wielowątkowych zadania mogą działać równolegle i współpracować w celu uzyskania rezultatów. Dlatego wątki muszą być w stanie komunikować się, aby umożliwić prawdziwe asynchroniczne przetwarzanie. W systemie Android znaczenie komunikacji wątków zostało zaakcentowane w charakterystycznym dla platformy mechanizmie `Handler/Looper` (procedura obsługi/procedura zapętlania). Na tym mechanizmie oraz tradycyjnych technikach Javy skupimy się w tym rozdziale. Rozdział obejmuje:

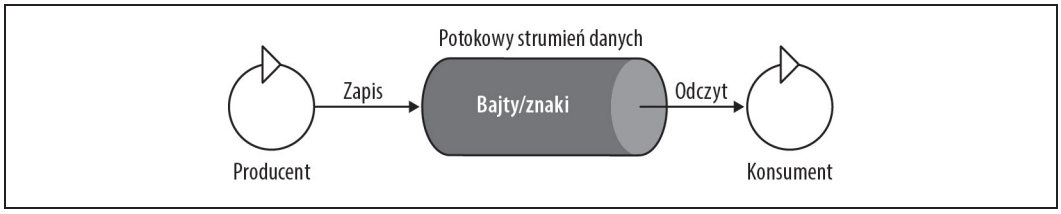
- przekazywanie danych przez jednokierunkowe potoki danych;
- komunikację za pomocą pamięci współdzielonej;
- implementację wzorca konsument – producent za pomocą interfejsu `BlockingQueue`;
- operacje na kolejkach komunikatów;
- wysyłanie zadania z powrotem do wątku interfejsu użytkownika.

Potoki

Potoki są częścią paczki `java.io`. Oznacza to, że są one ogólną funkcjonalnością Javy, a nie są charakterystyczne dla platformy Android. Dla dwóch wątków w ramach tego samego procesu potok stanowi sposób połączenia i ustanowienia jednokierunkowego kanału danych. Wątek producenta zapisuje dane do potoku, natomiast wątek konsumenta odczytuje dane z potoku.

Potok Java jest porównywalny do operatora potoku systemów Unix i Linux (znak specjalny | powłoki), który jest używany do przekierowania wyjścia z jednego polecenia na wejście do innego polecenia. Operator potoku działa w systemie Linux przez granice procesów, ale potoki Java działają pomiędzy wątkami w maszynie wirtualnej, np. w ramach procesu.

Sam potok jest buforem alokowanym w pamięci, dostępnym tylko dla dwóch połączonych wątków. Żadne inne wątki nie mogą uzyskać dostępu do danych. Dlatego zapewnione jest bezpieczeństwo wątków, które zostało omówione w rozdziale 2., w podrozdziale „Bezpieczeństwo wątków". Potok jest także jednokierunkowy, co pozwala tylko jednemu wątkowi zapisywać dane, a drugiemu odczytywać (patrz rysunek 4.1).



Rysunek 4.1. Komunikacja wątków z wykorzystaniem potoków

Potoki są zazwyczaj używane, gdy masz dwa długo wykonywane zadania i musisz w sposób ciągły odciążać jedno, przerzucając dane do drugiego. Potoki ułatwiają oddzielać zadania do kilku wątków, zamiast obsługiwać wiele zadań za pomocą tylko jednego wątku. Gdy jedno zadanie wyprodukuje wynik w wątku, wynik ten jest potokowany do następnego wątku, który dalej przetwarza dane. Zysk pochodzi z czystej separacji kodu i współbieżnego wykonywania. Potoki mogą być stosowane między wątkami roboczymi i do odciążania pracy wątku interfejsu użytkownika, który należy oszczędzać, aby zachować responsywność w doświadczeniu użytkownika.

Potok może przesyłać dane binarne lub znakowe. Transfer danych binarnych jest reprezentowany przez klasy `PipedOutputStream` (u producenta) i `PipedInputStream` (u konsumenta). Natomiast transfer danych znakowych jest reprezentowany przez klasy `PipedWriter` (u producenta) i `PipedReader` (u konsumenta). Niezależnie od typu transferu danych oba te potoki mają podobną funkcjonalność. Życie potoku zaczyna się, gdy wątek zapisujący lub odczytujący ustanawia połączenie, a kończy, gdy połączenie jest zamykane.

Podstawowe zastosowanie potoków

Podstawowy cykl życia potoku można podsumować w trzech etapach: konfiguracja, transfer danych (który może być powtarzany tak długo, jak te dwa wątki chcą wymieniać dane) oraz odłączenie. Poniższe przykłady zostały przygotowane z wykorzystaniem klas `PipedWriter` i `PipedReader`, ale te same etapy dotyczą klas `PipedOutputStream` i `PipedInputStream`.

1. Konfigurowanie połączenia:

   ```
   PipedReader r = new PipedReader();
   PipedWriter w = new PipedWriter();
   w.connect(r);
   ```

   W tym przypadku połączenie jest nawiązywane przez wątek zapisujący łączący się z wątkiem odczytującym. Połączenie może równie dobrze zostać ustanowione przez wątek odczytujący. Potok domyślnie konfiguruje również kilka konstruktorów. Domyślny rozmiar bufora to 1024, ale można go konfigurować od strony konsumenta potoku, tak jak pokazano poniżej:

   ```
   int BUFFER_SIZE_IN_CHARS = 1024 * 4;
   PipedReader r = new PipedReader(BUFFER_SIZE_IN_CHARS);
   PipedWriter w = new PipedWriter(r);
   ```

2. Przekazanie czytnika do wątku przetwarzania:

   ```
   Thread t = new MyReaderThread(r);
   t.start();
   ```

   Po uruchomieniu wątek odczytujący jest gotowy do odbioru danych z wątku zapisującego.

3. Transfer danych:

```
// Wątek producenta: zapisuje pojedynczy znak lub tablicę znaków.
w.write('A');

// Wątek konsumenta: odczytuje dane.
int result = r.read();
```

Komunikacja stosuje się do wzorca konsument – producent z mechanizmem blokowania. Jeśli potok jest pełny, metoda `write()` będzie blokować, aż odpowiednia ilość danych zostanie odczytana, a w konsekwencji usunięta z potoku, aby zwolnić miejsce dla danych, które próbuje dodać wątek zapisujący. Metoda `read()` blokuje za każdym razem, gdy nie ma danych do odczytu z potoku. Warto zauważyć, że metoda `read()` zwraca znak jako wartość całkowitą, aby upewnić się, że dostępna jest wystarczająca ilość miejsca do obsługi różnego kodowania o różnych rozmiarach. Możesz zrzucić wartość całkowitą z powrotem do znaku.

W praktyce najlepsze podejście będzie wyglądać następująco:

```
// Wątek producenta: spłukuje potok po wykonaniu zapisu.
w.write('A');
w.flush();

// Wątek konsumenta: odczytuje dane w pętli.
int i;
while((i = reader.read()) != -1){
    char c = (char) i;
    // Obsługuje odebrane dane.
}
```

Wywołanie metody `flush()` po zapisie do potoku informuje wątek konsumenta, że dostępne są nowe dane. Jest to przydatne z punktu widzenia wydajności, ponieważ gdy bufor jest pusty, `PipedReader` używa wywołania blokującego do metody `wait()` z jednosekundowym limitem czasu. Dlatego jeśli wywołanie metody `flush()` zostanie pominięte, wątek konsumenta może opóźnić odczyt danych o maksymalnie jedną sekundę. Wywołując metodę `flush()`, producent skraca czas oczekiwania w wątku konsumenta i umożliwia natychmiastowe kontynuowanie przetwarzania danych.

4. Zamknięcie połączenia.

Po zakończeniu fazy komunikacji potok powinien zostać odłączony:

```
// Wątek producenta: zamykanie modułu zapisu.
w.close();

// Wątek konsumenta: zamykanie modułu odczytu.
r.close();
```

Jeśli moduły zapisujący i odczytujący są połączone, wystarczy tylko zamknąć jeden z nich. Jeśli moduł zapisujący zostanie zamknięty, potok jest odłączany, ale dane znajdujące się w buforze wciąż można odczytać. Jeśli zamknięty zostanie moduł odczytujący, bufor jest czyszczony.

Przykład: przetwarzanie tekstu w wątku roboczym

Następny przykład ilustruje sposób, w jaki potoki mogą przetwarzać tekst, który użytkownik wprowadza w kontrolce `EditText`. Aby zachować responsywność wątku interfejsu użytkownika, każdy znak wprowadzony przez użytkownika jest przekazywany do wątku roboczego, który przypuszczalnie obsługuje pewne czasochłonne przetwarzanie:

```
public class PipeExampleActivity extends Activity {

    private static final String TAG = "PipeExampleActivity";
    private EditText editText;

    PipedReader r;
    PipedWriter w;

    private Thread workerThread;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        r = new PipedReader();
        w = new PipedWriter();

        try {
            w.connect(r);
        } catch (IOException e) {
            e.printStackTrace();
        }

        setContentView(R.layout.activity_pipe);
        editText = (EditText) findViewById(R.id.edit_text);
        editText.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence charSequence, int start,
                                          int count, int after) {
            }

            @Override
            public void onTextChanged(CharSequence charSequence, int start,
                                      int before, int count) {
                try {
                    // Obsługuje tylko dodawanie znaków.
                    if(count > before) {
                        // Zapisuje do potoku ostatni wprowadzony znak.
                        w.write(charSequence.subSequence(before, count).
                                toString());
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            @Override
            public void afterTextChanged(Editable editable) {
            }
        });

        workerThread = new Thread(new TextHandlerTask(r));
        workerThread.start();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        workerThread.interrupt();
        try {
            r.close();
            w.close();
        } catch (IOException e) {
        }
    }
```

```
    private static class TextHandlerTask implements Runnable {
        private final PipedReader reader;

        public TextHandlerTask(PipedReader reader){
            this.reader = reader;
        }
        @Override
        public void run() {
            while(Thread.currentThread().isInterrupted()){
                try {
                    int i;
                    while((i = reader.read()) != -1){
                        char c = (char) i;
                        //TUTAJ DODAJ LOGIKĘ PRZETWARZANIA TEKSTU.
                        Log.d(TAG, "char = " + c);
                    }

                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Gdy zostanie utworzona aktywność `PipeExampleActivity`, wyświetlane jest pole `EditText`, które posiada nasłuchiwacz (`TextWatcher`) zmian w zawartości. Za każdym razem, gdy w polu `EditText` dodawany jest nowy znak, jest on zapisywany do potoku i odczytywany w klasie `TextHandlerTask`. Zadaniem konsumenta jest wykonywanie nieskończonej pętli, która odczytuje znak z potoku tak długo, jak jest coś do odczytania. Pętla wewnętrzna `while` będzie blokować przy wywołaniu metody `read()`, jeśli potok jest pusty.

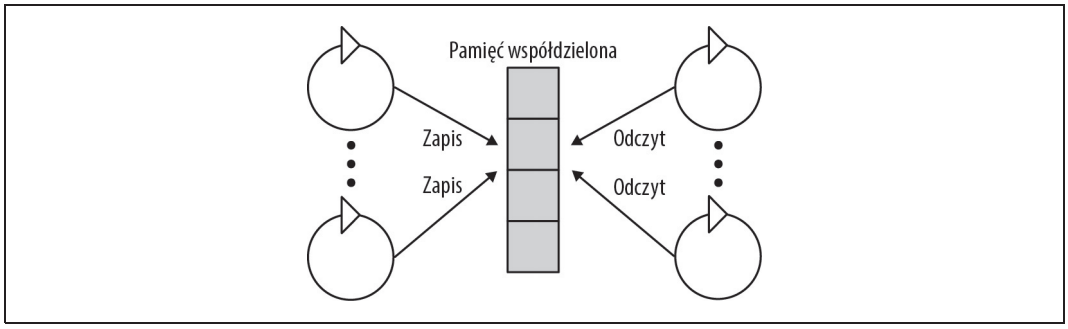
Bądź ostrożny, stosując potoki z wątkiem interfejsu użytkownika, ze względu na możliwość blokowania wywołań, jeśli potok jest pełny (producent blokuje na wywołaniu swojej metody `write()`) lub pusty (konsument blokuje na wywołaniu swojej metody `read()`).

Pamięć współdzielona

Pamięć współdzielona (używająca obszaru pamięci znanego w programowaniu jako **sterta**) jest popularnym sposobem przekazywania informacji pomiędzy wątkami. Wszystkie wątki w aplikacji mogą uzyskać dostęp do tej samej przestrzeni adresowej w procesie. Dlatego jeśli jeden wątek zapisuje wartość w zmiennej w pamięci wspólnej, wartość ta może być odczytana przez wszystkie inne wątki, tak jak pokazano na rysunku 4.2.

Jeśli wątek przechowuje dane jako zmienną lokalną, żaden inny wątek nie może ich zobaczyć. Przechowując dane w pamięci współdzielonej, wątek może używać zmiennych do komunikacji z innymi wątkami i dzielenia się z nimi pracą. Obiekty są przechowywane w pamięci współdzielonej, jeśli znajdują się w jednym z następujących zakresów:

- zmienne członków instancji,
- zmienne członków klasy,
- obiekty zadeklarowane w metodach.



Rysunek 4.2. Komunikacja wątków za pomocą pamięci współdzielonej

Referencja obiektu jest przechowywana lokalnie w stosie wątku, ale sam obiekt jest przechowywany w pamięci współdzielonej. Obiekt jest dostępny z wielu wątków, jeśli tylko metoda publikuje referencję poza zakresem metody, np. przez przekazanie tej referencji do metody innego obiektu. Wątki komunikują się poprzez pamięć współdzieloną, definiując pola instancji i klasy, które są dostępne z wielu wątków.

Sygnalizacja

Podczas gdy wątki komunikują się poprzez zmienne stanu w pamięci współdzielonej, mogą odpytywać wartość stanu, aby pobierać jego zmiany. Bardziej wydajny jest jednak wbudowany w bibliotece języka Java mechanizm sygnalizacji, który pozwala wątkowo powiadomić inne wątki o zmianach swojego stanu. Mechanizm sygnalizacji różni się w zależności od typu synchronizacji (patrz tabela 4.1).

Tabela 4.1. Sygnalizacja wątków

	synchronized	**ReentrantLock**	**ReentrantReadWriteLock**
Wywołanie blokujące, oczekiwanie na stan	`Object.wait()`	`Condition.await()`	`Condition.await()`
	`Object.wait(timeout)`	`Condition.await(timeout)`	`Condition.await(timeout)`
Sygnał dla zablokowanych wątków	`Object.notify()`	`Condition.signal()`	`Condition.signal()`
	`Object.notifyAll()`	`Condition.signalAll()`	`Condition.signalAll()`

Gdy wątek nie może kontynuować wykonywania, dopóki inny wątek nie osiągnie określonego stanu, wywołuje metodę `wait()/wait(timeout)` lub jej odpowiednik `await()/await(timeout)`, w zależności od zastosowanej synchronizacji. Parametry limitu czasu `timeout` wskazują, jak długo wątek wywołujący powinien czekać przed kontynuowaniem wykonywania.

Gdy inny wątek zmienia swój stan, sygnalizuje tę zmianę wywołaniem metody `notify()/notifyAll()` lub jej odpowiednika `signal()/signalAll()`. Po sygnale wątek oczekujący kontynuuje wykonywanie. W ten sposób wywołania obsługują dwa różne wzorce projektowe, które korzystają z warunków: wersja `notify()` lub `signal()` budzi jeden wybrany losowo wątek, natomiast wersja `notifyAll()` lub `signalAll()` budzi wszystkie wątki oczekujące na sygnał.

Ponieważ sygnał może być odbierany przez wiele wątków i jeden z nich może wejść do sekcji krytycznej, zanim pozostałe się obudzą, odbieranie sygnału nie gwarantuje, że zostanie osiągnięty prawidłowy stan. Wątek oczekujący powinien stosować wzorzec projektowy, w którym

przed dalszym wykonywaniem sprawdza, czy żądany warunek został spełniony. Jeśli np. współdzielony stan jest chroniony synchronizacją na blokadzie wewnętrznej, sprawdź stan przed wywołaniem metody `wait()`:

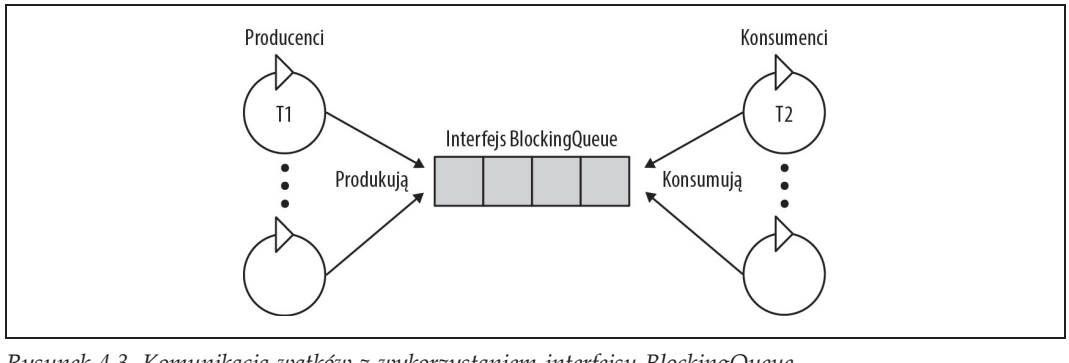
```
synchronized(this) {
    while(isConditionFulfilled == false) {
        wait();
    }
    // Kiedy wykonywanie osiąga ten punkt, stan jest poprawny.
}
```

Ten wzorzec sprawdza, czy predykat warunku został spełniony. Jeśli nie, wątek blokuje przez wywołanie metody `wait()`. Gdy inny wątek powiadamia monitor i wątek oczekujący budzi się, sprawdza ponownie, czy warunek ten został spełniony, a jeśli nie, blokuje ponownie, czekając na nowy sygnał.

Bardzo typowym przypadkiem użycia w systemie Android jest tworzenie wątku roboczego z wątku interfejsu użytkownika i pozwolenie wątkowi roboczemu na wyprodukowanie wyniku, który będzie używany przez pewien element interfejsu użytkownika. W ten sposób wątek interfejsu użytkownika musi czekać na wynik. Jednak wątek interfejsu użytkownika nie powinien czekać na sygnał z wątku działającego w tle, ponieważ może go to blokować. Zamiast tego należy użyć omówionego w dalszej części rozdziału mechanizmu przekazywania komunikatów systemu Android.

Interfejs BlockingQueue

Sygnalizacja wątków jest niskopoziomowym, wysoce konfigurowalnym mechanizmem, który może zostać przystosowany do wielu przypadków użycia, ale może być również traktowany jako technika najbardziej podatna na błędy. Dlatego platforma Java buduje wysokopoziomowe abstrakcje na bazie mechanizmu sygnalizacji wątków, aby rozwiązać problem jednokierunkowego przekazywania arbitralnych obiektów między wątkami. Ta abstrakcja jest często nazywana „rozwiązaniem problemu synchronizacji producent – konsument". Na ten problem składają się przypadki użycia, w których mogą być wątki produkujące zawartość (wątki producenta) i wątki konsumujące zawartość (wątki konsumenta). Producenci przekazują komunikaty do konsumentów w celu przetworzenia. Pośrednikiem między wątkami jest kolejka z funkcją blokowania, tj. `java.util.concurrent.BlockingQueue` (patrz rysunek 4.3).



Rysunek 4.3. Komunikacja wątków z wykorzystaniem interfejsu BlockingQueue

Interfejs `BlockingQueue` pełni rolę koordynatora pomiędzy wątkami producenta i konsumenta, zawijając razem implementację listy i sygnalizację. Lista zawiera konfigurowalną liczbę elementów, które wątki producenta wypełniają dowolnymi komunikatami danych. Po drugiej stronie wątki konsumenta wyodrębniają komunikaty w kolejności, w jakiej zostały umieszczone w kolejce, a następnie przetwarzają je. Koordynacja między producentami i konsumentami jest niezbędna, jeśli utracą synchronizację, np. jeśli producenci przekazują więcej komunikatów, niż konsumenci są w stanie obsłużyć. Dlatego interfejs `BlockingQueue` wykorzystuje warunki wątków, aby zapewnić, że producenci nie będą mogli kolejkować nowych komunikatów, jeśli lista `BlockingQueue` jest pełna, oraz że konsumenci będą wiedzieli, kiedy dostępne są komunikaty do pobrania. Synchronizację między wątkami można osiągnąć za pomocą sygnalizacji wątków, tak jak opisano w rozdziale 2., w punkcie „Przykład: konsument i producent". Jednak interfejs `BlockingQueue` zarówno blokuje wątki, jak i sygnalizuje ważne zmiany stanu, czyli lista nie jest pełna oraz lista nie jest pusta.

Wzorzec konsument – producent zaimplementowany za pomocą klasy `LinkedBlockingQueue` można łatwo wdrożyć przez dodawanie komunikatów do kolejki metodą `put()` i usuwanie ich metodą `take()`. Metoda `put()` blokuje podmiot wywołujący, jeśli kolejka jest pełna, a metoda `take()` blokuje podmiot wywołujący, jeśli kolejka jest pusta:

```
public class ConsumerProducer {

    private final int LIMIT = 10;
    private BlockingQueue<Integer> blockingQueue =
        new LinkedBlockingQueue<Integer>(LIMIT);

    public void produce() throws InterruptedException {
        int value = 0;

        while (true) {
            blockingQueue.put(value++);
        }
    }

    public void consume() throws InterruptedException {
        while (true) {
            int value = blockingQueue.take();
        }
    }
}
```

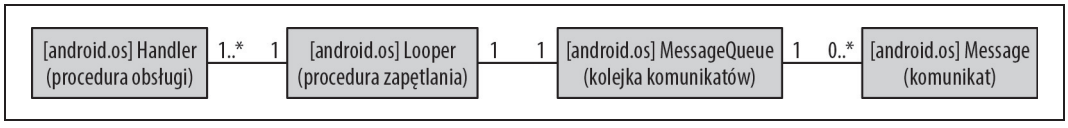
Przesyłanie komunikatów w systemie Android

Dotychczas omawiane opcje komunikacji wątków są regularnymi opcjami języka Java, dostępnymi w każdej aplikacji Java. Te mechanizmy (potoki, pamięć współdzielona oraz kolejki blokujące) mają zastosowanie do aplikacji systemu Android, ale stwarzają problemy w działaniu wątku interfejsu użytkownika z powodu ich „skłonności" do blokowania. Responsywność wątku interfejsu użytkownika jest zagrożona podczas korzystania z mechanizmów z funkcją blokowania, ponieważ okazjonalnie mogą one zawieszać wątek.

Najbardziej powszechnym przypadkiem użycia komunikacji wątków w systemie Android jest komunikacja między wątkiem interfejsu użytkownika i wątkami roboczymi. Dlatego platforma Android definiuje własny mechanizm przesyłania komunikatów, służący do komunikacji pomiędzy wątkami. Wątek interfejsu użytkownika może odciążyć długie zadania przez wy-

syłanie komunikatów z danymi do przetworzenia w wątkach tła. Mechanizm przesyłania komunikatów jest nieblokującym wzorcem konsument – producent, w którym ani wątek producenta, ani wątek konsumenta nie będą blokować podczas przekazywania komunikatu.

Mechanizm obsługi komunikatów jest fundamentalny w platformie Android, a interfejs API znajduje się w paczce *android.os* z zestawem klas przedstawionych na rysunku 4.4, które implementują tę funkcjonalność.



Rysunek 4.4. Przegląd interfejsu API

Klasa `android.os.Looper`
: Dyspozytor komunikatów powiązany z jedynym wątkiem konsumenta.

Klasa `android.os.Handler`
: Procesor komunikatów wątku konsumenta oraz interfejs dla wątku producenta służący do umieszczania komunikatów w kolejce. Obiekt `Looper` może mieć wiele powiązanych procedur obsługi, ale wszystkie umieszczają komunikaty w tej samej kolejce.

Klasa `android.os.MessageQueue`
: Nieograniczona lista powiązana komunikatów, które mają być przetworzone w wątku konsumenta. Każdy obiekt `Looper` (i `Thread`) ma co najwyżej jedną kolejkę `MessageQueue`.

Klasa `android.os.Message`
: Komunikat do wykonania w wątku konsumenta.

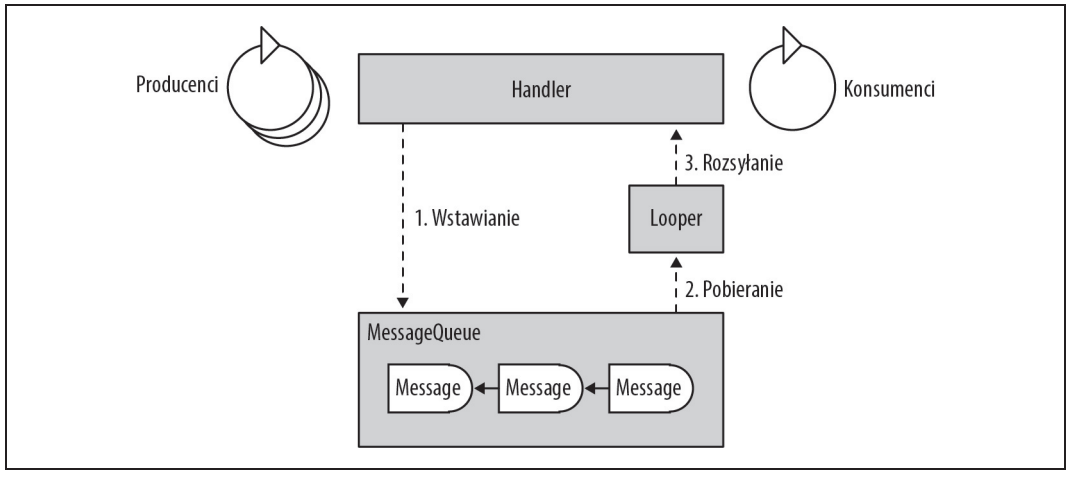
Komunikaty są wstawiane przez wątki producenta, a przetwarzane przez wątki konsumenta, tak jak pokazano na rysunku 4.5.

1. Wstawianie: wątek producenta umieszcza komunikat w kolejce za pomocą obiektu `Handler` podłączonego do wątku konsumenta, tak jak opisano w punkcie „Handler” w dalszej części podrozdziału.
2. Pobieranie: obiekt `Looper` (opisany w punkcie „Looper” w dalszej części podrozdziału) działa w wątku konsumenta i pobiera komunikaty z kolejki w porządku sekwencyjnym.
3. Rozsyłanie: procedury obsługi są odpowiedzialne za przetwarzanie komunikatów w wątku konsumenta. Wątek może mieć wiele instancji `Handler` dla przetwarzania komunikatów. `Looper` zapewnia, że komunikaty są rozsyłane do właściwej instancji `Handler`.

Przykład: podstawowe przesyłanie komunikatów

Zanim wnikliwie przeanalizujemy komponenty, przyjrzyjmy się podstawowemu przykładowi przesyłania komunikatów, aby zapoznać się z konfiguracją kodu.

Poniższy kod implementuje prawdopodobnie jeden z najbardziej powszechnych przypadków użycia. Użytkownik naciska na ekranie przycisk, który może wywołać długą operację, taką jak operacja sieciowa. Aby uniknąć zablokowania renderowania interfejsu użytkownika, długa



Rysunek 4.5. Przegląd mechanizmu przesyłania komunikatów pomiędzy wieloma wątkami producenta i jednym wątkiem konsumenta. Każdy komunikat odnosi się do kolejnego komunikatu w kolejce, co na rysunku zostało oznaczone strzałką skierowaną w lewo

operacja — reprezentowana tutaj przez atrapę metody `doLongRunningOperation()` — musi być wykonana w wątku roboczym. Dlatego konfiguracja ogranicza się jedynie do jednego wątku producenta (wątek interfejsu użytkownika) i jednego wątku konsumenta (`LooperThread`).

Nasz kod konfiguruje kolejkę komunikatów. Obsługuje klikanie przycisku jak zwykle w wywołaniu zwrotnym metody `Click()`, które wykonuje w wątku interfejsu użytkownika. W naszej implementacji wywołanie zwrotne wstawia atrapę komunikatu do kolejki komunikatów. Dla zwięzłości układy i komponenty interfejsu użytkownika zostały w kodzie pominięte:

```
public class LooperActivity extends Activity {

    LooperThread mLooperThread;

    private static class LooperThread extends Thread { ❶

        public Handler mHandler;

        public void run() {
            Looper.prepare(); ❷
            mHandler = new Handler() { ❸

                public void handleMessage(Message msg) { ❹
                    if(msg.what == 0) {
                        doLongRunningOperation();
                    }
                }
            };
            Looper.loop(); ❺
            }
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mLooperThread = new LooperThread(); ❻
        mLooperThread.start();
    }

    public void onClick(View v) {
```

```
            if (mLooperThread.mHandler != null) { ❼
                Message msg = mLooperThread.mHandler.obtainMessage(0);  ❽
                            mLooperThread.mHandler.sendMessage(msg); ❾
            }
        }

        private void doLongRunningOperation() {
            // Dodaj tutaj długotrwałą operację.
        }

        protected void onDestroy() {
            mLooperThread.mHandler.getLooper().quit(); ❿
        }
    }
```

❶ Definicja wątku roboczego działającego jako konsument kolejki komunikatów.

❷ Powiązanie obiektu `Looper` (i pośrednio kolejki `MessageQueue`) z wątkiem.

❸ Konfiguracja procedury obsługi `Handler`, która będzie używana przez producenta do umieszczania komunikatów w kolejce. Tutaj używamy domyślnego konstruktora, który będzie wiązał do obiektu `Looper` bieżącego wątku. Dlatego ten obiekt `Handler` może być utworzony tylko po metodzie `Looper.prepare()`, bo inaczej nie będzie się miał z czym wiązać.

❹ Wywołanie zwrotne, które jest uruchamiane, gdy komunikat zostanie przesłany do wątku roboczego. Sprawdza parametr `what`, a następnie wykonuje długą operację.

❺ Rozpoczęcie rozsyłania komunikatów z kolejki komunikatów do wątku konsumenta. Jest to wywołanie blokujące, więc wątek roboczy nie zakończy wykonywania.

❻ Uruchomienie wątku roboczego, aby był gotowy do przetwarzania komunikatów.

❼ Istnieje warunek wyścigu między konfiguracją obiektu `mHandler` w wątku tła i tym wykorzystaniem w wątku interfejsu użytkownika. Dlatego należy przeprowadzić walidację, czy `mHandler` jest dostępny.

❽ Inicjowanie obiektu `Message` z argumentem `what` ustawionym arbitralnie na `0`.

❾ Umieszczanie komunikatu w kolejce.

❿ Zakończenie wątku tła. Wywołanie metody `Looper.quit()` zatrzymuje rozsyłanie komunikatów i zwalnia metodę `Looper.loop()` z blokowania, więc metoda `run` może zakończyć wykonywanie, co prowadzi do zamknięcia wątku.

Klasy stosowane w przesyłaniu komunikatów

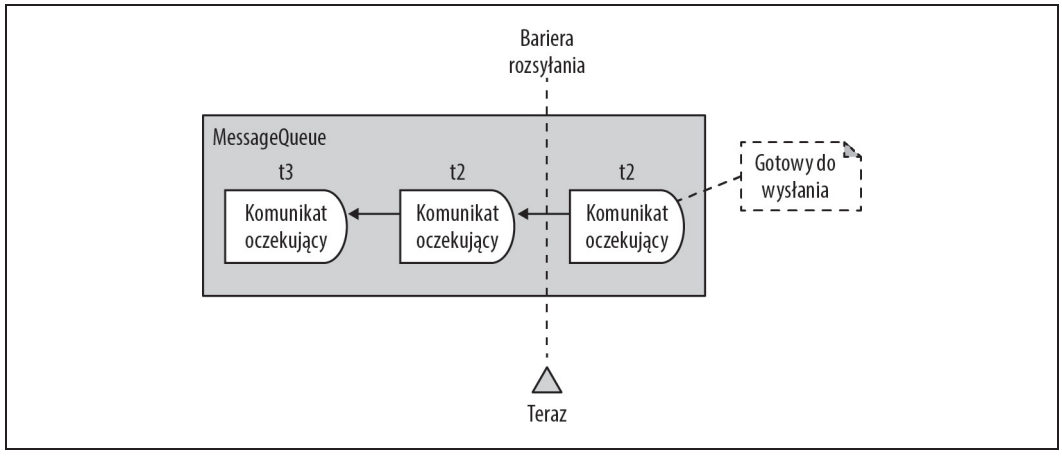
Przyjrzyjmy się teraz szczegółowo poszczególnym komponentom przesyłania komunikatów i ich zastosowaniu.

Klasa MessageQueue

Kolejka komunikatów jest reprezentowana przez klasę `android.os.MessageQueue`. Jest ona zbudowana z powiązanych komunikatów stanowiących nieograniczoną jednokierunkową listę powiązaną. Wątki producenta umieszczają w niej komunikaty, które będą później rozsyłane do konsumenta. Komunikaty są sortowane na podstawie znaczników czasu. Komunikat

oczekujący o najniższej wartości znacznika czasu jest pierwszy w kolejce do rozesłania do konsumenta. Jednak komunikat jest wysyłany tylko wtedy, gdy wartość znacznika czasu jest mniejsza niż bieżący czas. Jeśli nie, wysyłka zostanie wstrzymana do momentu, aż bieżący czas przekroczy wartość znacznika czasu.

Na rysunku 4.6 przedstawiono kolejkę komunikatów z trzema oczekującymi komunikatami posortowanymi według znaczników czasu, gdzie *t1* < *t2* < *t3*. Tylko jeden komunikat przeszedł barierę rozsyłania, którą jest aktualny czas. Kwalifikujące się do wysłania komunikaty mają wartość znacznika czasu mniejszą niż bieżący czas (na rysunku oznaczony jako „Teraz").



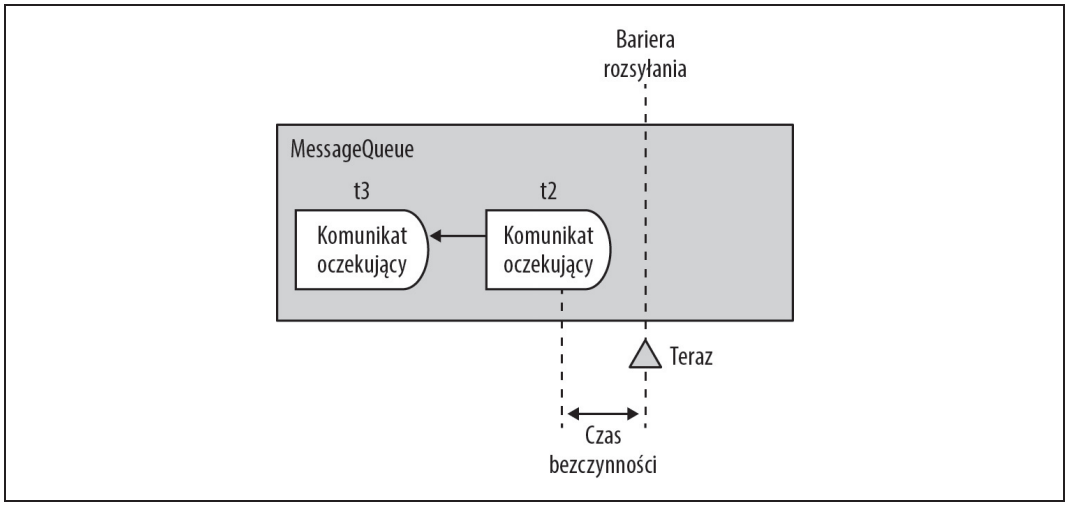
Rysunek 4.6. Komunikaty oczekujące w kolejce. Skrajny prawy komunikat jest pierwszy w kolejce do przetworzenia. Strzałki komunikatów oznaczają referencje do następnego komunikatu w kolejce

Jeśli żaden komunikat nie przeszedł bariery rozsyłania, kiedy `Looper` jest gotowy do pobrania następnego komunikatu, wątek konsumenta blokuje. Wykonywanie zostanie wznowione, gdy tylko komunikat przejdzie przez barierę rozsyłania.

Producenci mogą umieszczać nowe komunikaty w kolejce w dowolnym czasie i na dowolnym miejscu w kolejce. Pozycja umieszczania w kolejce jest oparta na wartości znacznika czasu. Jeśli nowy komunikat ma najniższą wartość znacznika czasu w porównaniu z komunikatami oczekującymi w kolejce, będzie zajmować pierwsze miejsce w kolejce, które kwalifikuje go do rozesłania. Umieszczanie komunikatu w kolejce jest zawsze zgodne z kolejnością sortowania według znacznika czasu. Umieszczanie komunikatu zostało omówione szczegółowo w punkcie „Handler" w dalszej części rozdziału.

Klasa MessageQueue.IdleHandler

Jeśli nie ma żadnych komunikatów do przetworzenia, wątek konsumenta pozostaje przez pewien czas w bezczynności. Na rysunku 4.7 zilustrowano szczelinę czasową, w której wątek konsumenta jest w stanie bezczynności. Domyślnie w czasie bezczynności wątek konsumenta po prostu czeka na nowe komunikaty. Jednak zamiast oczekiwać, w tych okresach bezczynności wątek może być wykorzystywany do wykonywania innych zadań. Ta funkcja może być wykorzystana do umożliwienia odłożenia wykonywania niekrytycznych zadań do momentu, aż żadne inne komunikaty nie będą konkurować o czas wykonywania.



Rysunek 4.7. Jeśli żaden komunikat nie przeszedł przez barierę rozsyłania, tworzy się okno czasowe, które może być wykorzystane do wykonywania zadań, dopóki nie będzie wymagane wykonanie następnego komunikatu oczekującego

Gdy komunikat oczekujący zostanie wysłany i żaden inny komunikat nie przeszedł przez barierę rozsyłania, tworzy się okno czasowe, w którym wątek konsumenta może być wykorzystany do wykonywania innych zadań. Aplikacja zdobywa to okno czasowe za pomocą interfejsu `android.os.MessageQueue.IdleHandler`. Jest to nasłuchiwacz, który generuje wywołania zwrotne, gdy wątek konsumenta jest w stanie bezczynności. Nasłuchiwacz jest dołączany do kolejki `MessageQueue` i odłączany od niej za pomocą następujących wywołań:

```
// Uzyskanie kolejki komunikatów bieżącego wątku.
MessageQueue mq = Looper.myQueue();
// Tworzenie i rejestrowanie nasłuchiwacza bezczynności.
MessageQueue.IdleHandler idleHandler = new MessageQueue.IdleHandler();
mq.addIdleHandler(idleHandler)
// Wyrejestrowanie nasłuchiwacza bezczynności.
mq.removeIdleHandler(idleHandler)
```

Interfejs procedury obsługi bezczynności (ang. *idle handler*) składa się z tylko jednego wywołania zwrotnego metody:

```
interface IdleHandler {
    boolean queueIdle();
}
```

Gdy kolejka komunikatów wykrywa czas bezczynności dla wątku konsumenta, wywołuje metodę `queueIdle()` na wszystkich zarejestrowanych instancjach `IdleHandler`. To do aplikacji należy odpowiedzialne zaimplementowanie wywołania zwrotnego. Zwykle należy unikać długotrwałych zadań, ponieważ podczas wykonywania będą one opóźniać komunikaty oczekujące.

Implementacja metody `queueIdle()` musi zwracać poniższe wartości logiczne:

Wartość `true` (prawda)
: Interfejs `IdleHandler` jest utrzymywany w stanie aktywności. Kontynuuje odbieranie wywołań zwrotnych dla kolejnych okien czasowych bezczynności.

Wartość `false` (fałsz)

Interfejs `IdleHandler` jest nieaktywny. Nie będzie więcej odbierał wywołań zwrotnych dla kolejnych okien czasowych bezczynności. To jest to samo co usunięcie nasłuchiwacza poprzez metodę `MessageQueue.removeIdleHandler()`.

Przykład: użycie interfejsu IdleHandler do zakończenia niewykorzystywanego wątku

Wszystkie zarejestrowane interfejsy `IdleHandler` dla kolejek `MessageQueue` są wywoływane, gdy wątek znajduje się w oknie czasowej bezczynności, gdzie czeka na nowe komunikaty do przetworzenia. Okna bezczynności mogą wystąpić przed pierwszym komunikatem, pomiędzy komunikatami oraz po ostatnim komunikacie. Jeśli wiele producentów zawartości będzie przetwarzać dane sekwencyjnie w wątku konsumenta, interfejs `IdleHandler` może być wykorzystywany do zakończenia wątku konsumenta, gdy wszystkie komunikaty zostały przetworzone, aby niewykorzystywany wątek nie pozostawał w pamięci. Przy zastosowaniu interfejsu `IdleHandler` nie jest konieczne śledzenie ostatniego umieszczonego komunikatu, aby wiedzieć, kiedy wątek może zostać zakończony.

Ten przypadek użycia ma zastosowanie tylko wtedy, gdy wątki produkujące umieszczają komunikaty w kolejce `MessageQueue` bez opóźnienia, tak aby wątek konsumenta nigdy nie znajdował się w stanie bezczynności, dopóki nie zostanie wstawiony ostatni komunikat.

Metoda `ConsumeAndQuitThread` przedstawia strukturę konsumowania wątku z obiektami `Looper` i `MessageQueue`, która kończy wątek, gdy nie ma więcej komunikatów do przetworzenia:

```
public class ConsumeAndQuitThread extends Thread
    implements MessageQueue.IdleHandler {

    private static final String THREAD_NAME = "ConsumeAndQuitThread";

    public Handler mConsumerHandler;
    private boolean mIsFirstIdle = true;

    public ConsumeAndQuitThread() {
        super(THREAD_NAME);
    }

    @Override
    public void run() {
        Looper.prepare();
        mConsumerHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                    // Konsumowanie danych.
            }
        };
        Looper.myQueue().addIdleHandler(this); ❶
        Looper.loop();
    }

    @Override
    public boolean queueIdle() {
        if (mIsFirstIdle) { ❷
            mIsFirstIdle = false;
            return true; ❸
        }
```

```
            mConsumerHandler.getLooper().quit(); ❹
            return false;
        }

        public void enqueueData(int i) {
            mConsumerHandler.sendEmptyMessage(i);
        }
    }
```

❶ Rejestrowanie interfejsu `IdleHandler` w wątku tła, kiedy zostaje on uruchomiony, a `Looper` jest przygotowany, więc skonfigurowana zostaje kolejka `MessageQueue`.

❷ Pierwsze wywołanie metody `queueIdle` powinno przejść, ponieważ występuje przed odebraniem pierwszego komunikatu.

❸ Zwraca wartość `true` przy pierwszym wywołaniu, aby interfejs `IdleHandler` pozostał zarejestrowany.

❹ Zakończenie wątku.

Umieszczanie komunikatów jest przeprowadzane z wielu wątków jednocześnie, z symulacją losowości czasu wstawiania:

```
final ConsumeAndQuitThread consumeAndQuitThread = new ConsumeAndQuitThread();
consumeAndQuitThread.start();

for (int i = 0; i < 10; i++) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                SystemClock.sleep(new Random().nextInt(10));
                consumeAndQuitThread.enqueueData(i);
            }
        }
    }).start();
```

Komunikaty

Każdy element w `MessageQueue` pochodzi z klasy `android.os.Message`. Jest to obiekt kontenera przenoszący element danych lub zadanie, ale nigdy obie te rzeczy. Dane są przetwarzane przez wątek konsumenta, podczas gdy zadanie jest po prostu wykonywane, gdy zostanie usunięte z kolejki i nie ma innego przetwarzania do przeprowadzenia.

Komunikat zna swój procesor odbiorczy (tj. `Handler`) i może sam się kolejkować za pomocą metody `Message.sendToTarget()`:

```
Message m = Message.obtain(handler, runnable);
m.sendToTarget();
```

Jak zobaczysz w punkcie „Handler” w dalszej części podrozdziału, procedura obsługi jest najczęściej używana do kolejkowania komunikatów, ponieważ oferuje większą elastyczność w odniesieniu do umieszczania komunikatów w kolejce.

Komunikat danych

Zestaw danych zawiera wiele parametrów, które mogą być przekazywane do wątku konsumenta, tak jak to przedstawiono w tabeli 4.2.

Tabela 4.2. Parametry komunikatu

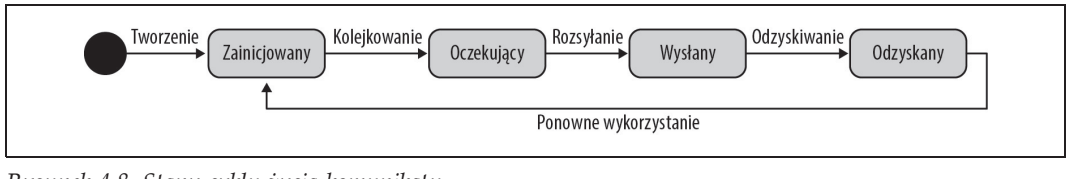
Nazwa parametru	Typ	Zastosowanie
`what`	`int`	Identyfikator komunikatu. Przekazuje intencję komunikatu.
`arg1, arg2`	`int`	Proste wartości danych służące do obsługi powszechnego przypadku użycia, jakim jest przekazywanie liczb całkowitych. Jeśli do konsumenta mają być przekazane maksymalnie dwie wartości całkowite, te parametry są bardziej efektywne niż alokowanie typu `Bundle`, co zostało objaśnione poniżej przy parametrze `data`.
`obj`	`Object`	Obiekt arbitralny. Jeśli obiekt jest przekazywany do wątku w innym procesie, musi implementować interfejs `Parcelable`.
`data`	`Bundle`	Kontener arbitralnych wartości danych.
`replyTo`	`Messenger`	Referencja do obiektu `Handler` w innym procesie. Umożliwia wymianę komunikatów między procesami, tak jak zostało to opisane w rozdziale 5., w punkcie „Komunikacja dwukierunkowa".
`callback`	`Runnable`	Zadanie, które ma być wykonane w wątku. Jest to pole wewnętrznej instancji, które przechowuje obiekt `Runnable` z metod `Handler.post`, tak jak zostało to opisane w punkcie „Handler" w dalszej części podrozdziału.

Komunikat zadania

Zadanie jest reprezentowane przez obiekt `java.lang.Runnable`, który ma być wykonany w wątku konsumenta. Komunikaty zadań nie mogą zawierać żadnych danych poza samym zadaniem.

Kolejka `MessageQueue` może zawierać dowolną kombinację komunikatów danych i zadań. Wątek konsumenta przetwarza je w sposób sekwencyjny, niezależnie od rodzaju. Jeśli komunikat jest komunikatem danych, konsument przetwarza te dane. Komunikaty zadań są obsługiwane przez umożliwienie wykonania obiektu `Runnable` w wątku konsumenta, ale wątek konsumenta nie otrzymuje komunikatu, który ma być przetworzony w metodzie `Handler.handleMessage(Message)`, jak ma to miejsce w przypadku komunikatów danych.

Cykl życia komunikatu jest prosty: producent tworzy komunikat, który ostatecznie jest przetwarzany przez konsumenta. Ten opis jest wystarczający dla większości przypadków użycia, ale kiedy pojawia się problem, głębsze zrozumienie obsługi komunikatów jest bezcenne. Rzućmy okiem na to, co faktycznie dzieje się z komunikatem podczas jego cyklu życia, który można podzielić na cztery główne stany, tak jak przedstawiono na rysunku 4.8. Środowisko uruchomieniowe przechowuje obiekty komunikatów w puli całej aplikacji, aby umożliwić ponowne wykorzystywanie poprzednich komunikatów. Pozwala to uniknąć narzutu tworzenia nowych instancji dla każdego przekazania. Czas wykonywania obiektu komunikatu jest zwykle bardzo krótki, a w jednostce czasu przetwarzanych jest wiele komunikatów.



Rysunek 4.8. Stany cyklu życia komunikatu

Transfery stanu są częściowo kontrolowane przez aplikację, a częściowo przez platformę. Należy zauważyć, że stany nie są obserwowalne, a aplikacja nie może śledzić przejść z jednego stanu do innego (chociaż istnieją sposoby śledzenia ruchu komunikatów, które zostały

opisane w punkcie „Obserwowanie kolejki komunikatów" w dalszej części podrozdziału). Dlatego aplikacja nie powinna dokonywać żadnych założeń na temat bieżącego stanu podczas obsługiwania komunikatu.

Zainicjowany

W stanie zainicjowanym utworzony został obiekt komunikatu ze stanem mutowalnym, a jeśli jest to komunikat danych, został wypełniony danymi. Aplikacja jest odpowiedzialna za tworzenie obiektu komunikatu przy użyciu jednego z poniższych wywołań, które biorą obiekt z puli obiektów:

- bezpośrednie konstruowanie obiektu:

```
Message m  = new Message();
```

- metody fabryki:
  - pusty komunikat:

```
Message m = Message.obtain();
```

  - komunikat danych:

```
Message m = Message.obtain(Handler h);
Message m = Message.obtain(Handler h, int what);
Message m = Message.obtain(Handler h, int what, Object o);
Message m = Message.obtain(Handler h, int what, int arg1, int arg2);
Message m = Message.obtain(Handler h, int what, int arg1, int arg2, Object o);
```

  - komunikat zadania:

```
Message m = Message.obtain(Handler h, Runnable task);
```

  - konstruktor kopiowania:

```
Message m = Message.obtain(Message originalMsg);
```

Oczekujący

Komunikat został umieszczony w kolejce przez wątek producenta i czeka na wysłanie do wątku konsumenta.

Wysłany

W tym stanie `Looper` pobrał i usunął komunikat z kolejki. Komunikat został wysłany do wątku konsumenta i jest aktualnie przetwarzany. Nie ma interfejsu API aplikacji dla tej operacji, ponieważ rozsyłanie jest kontrolowane przez `Looper` bez wpływu na aplikację. Kiedy `Looper` wysyła komunikat, sprawdza informacje o doręczeniu komunikatu i dostarcza komunikat do właściwego odbiorcy. Po wysłaniu komunikat jest wykonywany w wątku konsumenta.

Odzyskany

W tym momencie w cyklu życia stan komunikatu zostaje wyczyszczony i instancja zostaje zwrócona do puli komunikatów. `Looper` obsługuje recykling komunikatu po zakończeniu jego wykonywania w wątku konsumenta. Recykling komunikatów jest obsługiwany przez środowisko uruchomieniowe i nie powinien być przeprowadzany bezpośrednio przez aplikację.

Gdy komunikat zostaje umieszczony w kolejce, jego zawartość nie powinna być zmieniana. Teoretycznie można zmieniać zawartość przed wysłaniem komunikatu. Jednak ponieważ stan nie jest obserwowalny, komunikaty mogą być przetwarzane przez wątek konsumenta w czasie, kiedy producent próbuje zmienić dane, co wpływa na bezpieczeństwo wątku. Byłoby jeszcze gorzej, jeśli komunikat zostałby odzyskany, ponieważ potem zostałby zwrócony do puli komunikatów i ewentualnie wykorzystany przez innego producenta do przekazywania danych w innej kolejce.

Looper

Klasa `android.os.Looper` obsługuje wysyłkę komunikatów z kolejki do powiązanej procedury obsługi. Wszystkie komunikaty, które przeszły przez barierę rozsyłania (tak jak pokazano na rysunku 4.6), kwalifikują się do wysłania przez `Looper`. Dopóki kolejka ma komunikaty kwalifikujące się do rozesłania, `Looper` zapewnia, że wątek konsumenta będzie odbierał komunikaty. Gdy nie ma komunikatów, które przeszły przez barierę rozsyłania, wątek konsumenta będzie blokował, aż jakiś komunikat przejdzie barierę rozsyłania.

Wątek konsumenta nie wchodzi w bezpośrednią interakcję z kolejką komunikatów, aby pobierać komunikaty. Zamiast tego kolejka komunikatów jest dodawana do wątku, kiedy zostaje dołączony `Looper`. `Looper` zarządza kolejką komunikatów i ułatwia rozsyłanie komunikatów do wątku konsumenta.

Domyślnie tylko wątek interfejsu użytkownika posiada obiekt `Looper`. Wątki utworzone w aplikacji muszą bezpośrednio powiązać `Looper`. Gdy dla wątku tworzony jest `Looper`, jest on podłączany do kolejki komunikatów. `Looper` działa jako pośrednik między kolejką a wątkiem. Konfiguracja odbywa się w metodzie `run` wątku:

```
class ConsumerThread extends Thread {
    @Override
    public void run() {
        Looper.prepare(); ❶

        // Pominięte tworzenie obiektu Handler.

        Looper.loop(); ❷
```

❶ Pierwszym krokiem jest utworzenie obiektu `Looper`, co odbywa się za pomocą statycznej metody `prepare()`. Spowoduje to utworzenie kolejki komunikatów i powiązanie jej z bieżącym wątkiem. W tym momencie kolejka komunikatów jest gotowa do umieszczania w niej komunikatów, ale nie zostały one jeszcze rozesłane do wątku konsumenta.

❷ Rozpoczęcie obsługi komunikatów w kolejce komunikatów. Jest to metoda blokująca, która zapewnia, że metoda `run()` nie zostanie zakończona. Podczas gdy metoda `run()` blokuje, `Looper` wysyła komunikaty do wątku konsumenta w celu przetworzenia.

Wątek może mieć tylko jeden powiązany obiekt `Looper`. Jeśli aplikacja spróbuje skonfigurować drugi, wystąpi błąd wykonywania aplikacji (ang. *runtime error*). W konsekwencji wątek może mieć tylko jedną kolejkę komunikatów, co oznacza, że komunikaty wysyłane przez wiele wątków producenta są przetwarzane sekwencyjnie w wątku konsumenta. Dlatego aktualnie wykonywany komunikat odłoży wykonywanie kolejnych komunikatów do czasu, aż zostanie przetworzony. Komunikaty z długim czasem wykonywania nie będą użyte, jeżeli mogą opóźnić inne ważne zadania w kolejce.

Zakończenie obiektu Looper

Zatrzymanie przetwarzania komunikatów przez `Looper` odbywa się za pomocą metod `quit` lub `quitSafely`. Metoda `quit()` zatrzymuje wysyłanie przez `Looper` kolejnych komunikatów z kolejki. Wszystkie komunikaty oczekujące w kolejce, w tym te, które przeszły przez barierę rozsyłania, zostaną porzucone. Z drugiej strony metoda `quitSafely` porzuca tylko komunikaty, które nie przeszły przez barierę rozsyłania. Oczekujące komunikaty, które zostały zakwalifikowane do wysyłki, zostaną przetworzone przed zatrzymaniem obiektu `Looper`.

Metodę `quitSafely` dodano w interfejsie API poziomu 18 (Jelly Bean 4.3). Dotychczasowe poziomy API obsługiwały tylko metodę `quit`.

Zakończenie obiektu `Looper` nie kończy wątku. Następuje jedynie wyjście z metody `Looper.loop()` i wątek może wznowić działanie w metodzie, która wywołała metodę `loop`. Jednak nie można uruchomić starego lub nowego obiektu `Looper`, więc wątek nie może już kolejkować ani obsługiwać komunikatów. Jeśli wywołasz metodę `Looper.prepare()`, wystąpi wyjątek `RuntimeException`, ponieważ wątek ma już dołączony `Looper`. Jeśli wywołasz metodę `Looper.loop()`, będzie ona blokować, ale żadne komunikaty nie będą rozsyłane z kolejki.

Looper wątku interfejsu użytkownika

Wątek interfejsu użytkownika jest jedynym wątkiem domyślnie powiązanym z obiektem `Looper`. Jest to regularny wątek, jak każdy inny wątek utworzony przez samą aplikację, ale `Looper` jest kojarzony z tym wątkiem[1], zanim zostaną zainicjowane komponenty aplikacji.

Istnieje kilka praktycznych różnic między obiektem `Looper` wątku interfejsu użytkownika a obiektami `Looper` innych wątków aplikacji:

- Jest dostępny z każdego miejsca poprzez metodę `Looper.getMainLooper()`.
- Nie może być zakończony. Metoda `Looper.quit()` wyrzuca wyjątek `RuntimeException`.
- Środowisko uruchomieniowe kojarzy `Looper` z wątkiem interfejsu użytkownika poprzez metodę `Looper.prepareMainLooper()`. Może to być wykonane tylko raz w każdej aplikacji. Dlatego próba dołączenia głównego obiektu `Looper` do innego wątku spowoduje wystąpienie wyjątku.

Handler

Dotychczas koncentrowaliśmy się na wewnętrznych mechanizmach komunikacji wątków w systemie Android, ale aplikacja przeważnie wchodzi w interakcje z klasą `android.os.Handler`. Jest to dwustronny interfejs API, który obsługuje umieszczanie komunikatów w kolejce oraz przetwarzanie komunikatów. Jak pokazano na rysunku 4.5, jest wywoływany zarówno przez wątki producenta, jak i konsumenta i zwykle jest stosowany do:

- tworzenia komunikatów,
- umieszczania komunikatów w kolejce,

[1] Wątek interfejsu użytkownika jest zarządzany przez wewnętrzną klasę platformy `android.app.ActivityThread`.

- przetwarzania komunikatów w wątku konsumenta,
- zarządzania komunikatami w kolejce.

Konfiguracja

W trakcie wykonywania swoich obowiązków `Handler` współdziała z obiektem `Looper`, kolejką komunikatów oraz komunikatami. Jak przedstawiono na rysunku 4.4, jedyną bezpośrednią relacją instancji jest relacja z obiektem `Looper`, który służy do łączenia się z kolejką `Message ↪Queue`. Bez obiektu `Looper` procedury obsługi nie mogą funkcjonować. Nie mogą się łączyć z kolejką, aby wstawiać komunikaty, a tym samym nie będą otrzymywać żadnych komunikatów do przetworzenia. Dlatego instancja `Handler` jest wiązana z instancją `Looper` podczas konstruowania:

- Konstruktory bez wyraźnie wskazanego obiektu `Looper` wiążą do obiektu `Looper` bieżącego wątku:

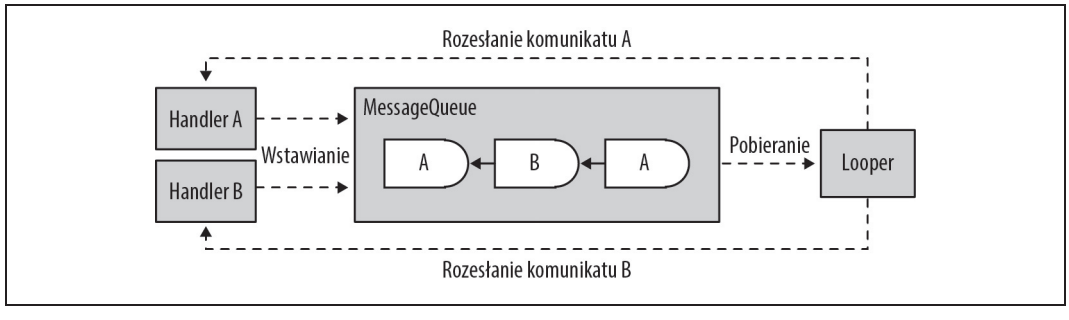
```
new Handler();
new Handler(Handler.Callback)
```

- Konstruktory z wyraźnym wskazaniem obiektu `Looper` wiążą się z tym obiektem `Looper`:

```
new Handler(Looper);
new Handler(Looper, Handler.Callback);
```

Jeśli konstruktory bez wyraźnego obiektu `Looper` są wywoływane w wątku bez obiektu `Looper` (czyli nie było wywołania metody `Looper.prepare()`), nie ma nic, do czego procedury obsługi mogłyby wiązać, co prowadzi do wystąpienia wyjątku `RuntimeException`. Kiedy procedura obsługi zostanie związana z obiektem `Looper`, wiązanie jest ostateczne.

Wątek może mieć wiele procedur obsługi. Pochodzące z nich komunikaty współistnieją w kolejce, ale są rozsyłane do właściwej instancji `Handler`, tak jak pokazano na rysunku 4.9.



Rysunek 4.9. Wiele procedur obsługi wykorzystujących jeden Looper. Procedura obsługi wstawiająca komunikat jest tą samą procedurą obsługi, która przetwarza ten komunikat

Wiele procedur obsługi nie umożliwia współbieżnego wykonywania. Komunikaty znajdują się nadal w tej samej kolejce i są przetwarzane sekwencyjnie.

Tworzenie komunikatu

Dla uproszczenia klasa `Handler` oferuje funkcje zawijające (ang. *wrapper functions*) dla metod fabryki (przedstawionych w podpunkcie „Zainicjowany" we wcześniejszej części podrozdziału) w celu tworzenia obiektów klasy `Message`:

```
Message obtainMessage(int what, int arg1, int arg2)
Message obtainMessage()
Message obtainMessage(int what, int arg1, int arg2, Object obj)
Message obtainMessage(int what)
Message obtainMessage(int what, Object obj)
```

Uzyskany z instancji `Handler` komunikat jest pobierany z puli komunikatów i pośrednio podłączany do instancji `Handler`, która go zażądała. To połączenie umożliwia instancji `Looper` rozesłanie każdego komunikatu do właściwej instancji `Handler`.

Wstawianie komunikatów

`Handler` umieszcza komunikaty w kolejce komunikatów na wiele sposobów, w zależności od typu komunikatu. Komunikaty zadania są wstawiane przez metody z przedrostkiem `post`, natomiast metody wstawiania danych mają przedrostek `send`:

- dodawanie zadania do kolejki komunikatów:

```
boolean post(Runnable r)f
boolean postAtFrontOfQueue(Runnable r)
boolean postAtTime(Runnable r, Object token, long uptimeMillis)
boolean postAtTime(Runnable r, long uptimeMillis)
boolean postDelayed(Runnable r, long delayMillis)
```

- dodawanie obiektu danych do kolejki komunikatów:

```
boolean sendMessage(Message msg)
boolean sendMessageAtFrontOfQueue(Message msg)
boolean sendMessageAtTime(Message msg, long uptimeMillis)
boolean sendMessageDelayed(Message msg, long delayMillis)
```

- dodawanie prostego obiektu danych do kolejki komunikatów:

```
boolean sendEmptyMessage(int what)
boolean sendEmptyMessageAtTime(int what, long uptimeMillis)
boolean sendEmptyMessageDelayed(int what, long delayMillis)
```

Wszystkie metody wstawiania umieszczają nowy obiekt `Message` w kolejce, nawet jeśli aplikacja nie tworzy obiektu `Message` bezpośrednio. Obiekty, takie jak `Runnable` w zadaniu `post` oraz `what` w zadaniu `send`, są zawijane w obiekty `Message`, ponieważ są to jedyne typy danych dozwolone w kolejce.

Każdy komunikat umieszczany w kolejce ma parametr czasu wskazujący, kiedy komunikat będzie się kwalifikował do wysłania do wątku konsumenta. Sortowanie opiera się na tym parametrze czasu i jest to jedyny sposób, w jaki aplikacja może wpływać na kolejność rozsyłania:

Parametr `default`
: Natychmiast kwalifikujący się do wysłania.

Parametr `at_front`
: Komunikat kwalifikuje się do wysłania w czasie `0`. Dlatego będzie następnym wysyłanym komunikatem, chyba że inny komunikat zostanie wstawiony z przodu, zanim ten zostanie przetworzony.

Parametr `delay`
: Określa czas, po jakim ten komunikat będzie się kwalifikował do wysłania.

Parametr `uptime`
: Czas bezwzględny, w którym komunikat będzie się kwalifikował do wysłania.

Chociaż można wyraźnie zdefiniować parametry `delay` i `uptime`, czas wymagany do przetworzenia każdego komunikatu jest nadal nieokreślony. Zależy to od tego, czy istniejące komunikaty muszą być przetworzone najpierw, oraz od algorytmu szeregowania systemu operacyjnego.

Umieszczanie komunikatów w kolejce jest podatne na błędy. Niektóre typowe błędy, które mogą wystąpić, zostały wymienione w tabeli 4.3.

Tabela 4.3. Błędy wstawiania komunikatów

Błąd	Komunikat błędu	Typowy problem aplikacji
Komunikat nie posiada obiektu `Handler`.	`RuntimeException`	Komunikat został utworzony z metody `Message.obtain()` bez określenia instancji `Handler`.
Komunikat został już wysłany i jest przetwarzany.	`RuntimeException`	Ta sama instancja komunikatu została wstawiona dwukrotnie.
Nastąpiło wyjście z obiektu `Looper`.	`Return false`	Komunikat został wstawiony po wywołaniu metody `Looper.quit()`.

Metoda `dispatchMessage` klasy `Handler` jest wykorzystywana przez `Looper` do rozsyłania komunikatów do wątku konsumenta. Jeśli zostanie użyta przez aplikację bezpośrednio, komunikat zostanie przetworzony natychmiast w wątku wywołującym, a nie w wątku konsumenta.

Przykład: dwukierunkowe przesyłanie komunikatów

Aktywność `HandlerExampleActivity` symuluje długotrwałą operację, która rozpoczyna się, gdy użytkownik kliknie przycisk. Długotrwałe zadanie jest wykonywane w wątku działającym w tle. W międzyczasie interfejs użytkownika wyświetla pasek postępu, który jest usuwany, gdy wątek tła prześle wynik z powrotem do wątku interfejsu użytkownika.

Najpierw należy skonfigurować komponent `Activity`:

```
public class HandlerExampleActivity extends Activity {

    private final static int SHOW_PROGRESS_BAR = 1;
    private final static int HIDE_PROGRESS_BAR = 0;
    private BackgroundThread mBackgroundThread;

    private TextView mText;
    private Button mButton;
    private ProgressBar mProgressBar;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler_example);

        mBackgroundThread = new BackgroundThread();
        mBackgroundThread.start(); ❶
        mText = (TextView) findViewById(R.id.text);
```

```
        mProgressBar = (ProgressBar) findViewById(R.id.progress);
        mButton = (Button) findViewById(R.id.button);
        mButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mBackgroundThread.doWork(); ❷
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mBackgroundThread.exit(); ❸
    }

    // ...reszta komponentu Activity jest definiowana poniżej.

}
```

❶ Wątek tła z kolejką komunikatów jest uruchamiany, gdy tworzona jest aktywność `Handler➥ExampleActivity`. Obsługuje ona zadania z wątku interfejsu użytkownika.

❷ Kiedy użytkownik kliknie przycisk, nowe zadanie jest wysyłane do wątku działającego w tle. Ponieważ zadania będą wykonywane sekwencyjnie w wątku tła, wielokrotne kliknięcie przycisku może prowadzić do kolejkowania zadań, zanim zostaną one przetworzone.

❸ Wątek tła zostaje zatrzymany, gdy aktywność `HandlerExampleActivity` jest niszczona.

`BackgroundThread` służy do odciążenia zadań z wątku interfejsu użytkownika. Działa (i może odbierać komunikaty) w czasie życia aktywności `HandlerExampleActivity`. Nie udostępnia swojego wewnętrznego obiektu `Handler`. Zamiast tego zawija wszystkie przypadki uzyskania dostępu do instancji `Handler` w publiczne metody `doWork` i `exit`:

```
private class BackgroundThread extends Thread {

    private Handler mBackgroundHandler;

    public void run() { ❶
        Looper.prepare();
        mBackgroundHandler = new Handler(); ❷
        Looper.loop();
    }

    public void doWork() {
        mBackgroundHandler.post(new Runnable() { ❸
            @Override
            public void run() {
                Message uiMsg = mUiHandler.obtainMessage(
                    SHOW_PROGRESS_BAR, 0, 0, null); ❹

                mUiHandler.sendMessage(uiMsg); ❺

                Random r = new Random();
                int randomInt = r.nextInt(5000);
                SystemClock.sleep(randomInt); ❻

                uiMsg = mUiHandler.obtainMessage(
                    HIDE_PROGRESS_BAR, randomInt, 0, null); ❼
                    mUiHandler.sendMessage(uiMsg); ❽
            }
        });
```

```
    }

    public void exit() { ❾
        mBackgroundHandler.getLooper().quit();
    }
}
```

❶ Skojarzenie obiektu `Looper` z wątkiem.

❷ `Handler` przetwarza tylko obiekty `Runnable`. Dlatego nie jest wymagana implementacja `Handler.handleMessage`.

❸ Wysyłanie długiego zadania do wykonania w tle.

❹ Tworzenie obiektu `Message` zawierającego tylko argument `what` z poleceniem `SHOW_PROGRESS_BAR` dla wątku interfejsu użytkownika, aby mógł pokazać pasek postępu.

❺ Wysyłanie komunikatu startowego do wątku interfejsu użytkownika.

❻ Symulowanie długiego zadania o losowej długości, które produkuje dane `randomInt`.

❼ Tworzenie obiektu `Message` z rezultatem `randomInt`, który jest przekazywany w parametrze `arg1`. Parametr `what` zawiera polecenie `HIDE_PROGRESS_BAR` w celu usunięcia paska postępu.

❽ Komunikat z rezultatem końcowym, który informuje wątek interfejsu użytkownika, że zadanie zostało zakończone, oraz dostarcza rezultat.

❾ Zamknięcie instancji `Looper`, aby można było zakończyć wątek.

Wątek interfejsu użytkownika definiuje własny `Handler`, który może odbierać polecenia do kontrolowania paska postępu i aktualizacji interfejsu użytkownika wynikami z wątku tła:

```
private final Handler mUiHandler = new Handler() {
    public void handleMessage(Message msg) {

        switch(msg.what) {
            case SHOW_PROGRESS_BAR: ❶
                mProgressBar.setVisibility(View.VISIBLE);
                break;
            case HIDE_PROGRESS_BAR: ❷
                mText.setText(String.valueOf(msg.arg1));
                mProgressBar.setVisibility(View.INVISIBLE);
                break;
        }
    }
};
```

❶ Pokazuje pasek postępu.

❷ Ukrywa pasek postępu i aktualizuje kontrolkę `TextView` uzyskanym wynikiem.

Przetwarzanie komunikatów

Komunikaty rozsyłane przez `Looper` są przetwarzane przez obiekt `Handler` w wątku konsumenta. Typ komunikatu określa przetwarzanie:

Komunikaty zadań
: Komunikaty zadań zawierają tylko obiekty `Runnable` i żadnych danych. Dlatego przetwarzanie do wykonywania jest zdefiniowane w metodzie `run` obiektu `Runnable`, która

jest wykonywana automatycznie na wątku konsumenta bez wywoływania metody `Handler.handleMessage()`.

Komunikaty danych

Gdy komunikat zawiera dane, obiekt `Handler` jest odbiornikiem danych i jest odpowiedzialny za ich przetwarzanie. Wątek konsumenta przetwarza, nadpisując metodę `Handler.handleMessage(Message msg)`. Można to zrobić na dwa sposoby, które zostały opisane poniżej.

Jednym ze sposobów na zdefiniowanie metody `handleMessage` jest zrobienie tego w ramach tworzenia obiektu `Handler`. Metoda ta powinna być zdefiniowana od razu, kiedy tylko dostępna będzie kolejka (po wywołaniu metody `Looper.prepare()`), ale przed rozpoczęciem pobierania komunikatów (przed wywołaniem metody `Looper.loop()`).

Poniżej zamieszczono szablon służący do konfigurowania obsługi komunikatów danych:

```
class ConsumerThread extends Thread {
    Handler mHandler;
    @Override
    public void run() {
        Looper.prepare();
        mHandler = new Handler() {
            public void handleMessage(Message msg) {
                // Tutaj przetwarzany jest komunikat danych.
            }
        };)
        Looper.loop();
    }
}
```

W tym kodzie `Handler` jest zdefiniowany jako anonimowa klasa wewnętrzna, ale może również być zdefiniowany jako klasa regularna lub klasa wewnętrzna.

Wygodną alternatywą dla rozszerzania klasy `Handler` jest użycie interfejsu `Handler.Callback` definiującego metodę `handleMessage` z dodatkowym parametrem zwrotnym, który nie jest umieszczany w metodzie `Handler.handleMessage()`:

```
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

Przy zastosowaniu interfejsu `Callback` rozszerzanie klasy `Handler` nie jest konieczne. Zamiast tego implementacja interfejsu `Callback` może być przekazana do konstruktora `Handler`, który następnie odbierze rozesłane do przetworzenia komunikaty:

```
public class HandlerCallbackActivity extends Activity implements Handler.Callback {
    Handler mUiHandler;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mUiHandler = new Handler(this);
    }

    @Override
    public boolean handleMessage(Message message) {
        // Przetwarzanie komunikatów.
        return true;
    }
}
```

Metoda `Callback.handleMessage` powinna zwrócić wartość `true` (prawda), jeśli komunikat został obsłużony, co gwarantuje, że nie będzie wykonywane żadne dalsze przetwarzanie tego komunikatu. Jeśli jednak zwracana jest wartość `false` (fałsz), komunikat jest przekazywany do metody `Handler.handleMessage` w celu dalszego przetwarzania. Należy zwrócić uwagę, że interfejs `Callback` nie nadpisuje metody `Handler.handleMessage`. Zamiast tego dodaje preprocesor, który jest wywoływany przed metodą klasy `Handler`. Preprocesor interfejsu `Callback` może przechwytywać i zmieniać komunikaty, zanim otrzyma je `Handler`. Poniższy kod przedstawia zasadę przechwytywania komunikatów za pomocą interfejsu `Callback`:

```
public class HandlerCallbackActivity extends Activity implements Handler.Callback { ❶

    @Override
    public boolean handleMessage(Message msg) { ❷
        switch (msg.what) {
            case 1:
                msg.what = 11;
                return true;
            default:
                msg.what = 22;
                return false;
        }
    }

    // Wywoływane po kliknięciu przycisku.

    public void onHandlerCallback(View v) {
        Handler handler = new Handler(this) {
            @Override
            public void handleMessage(Message msg) {
                // Przetwarzanie komunikatu. ❸
            }
        };
        handler.sendEmptyMessage(1); ❹
        handler.sendEmptyMessage(2); ❺
    }
}
```

❶ Aktywność `HandlerCallbackActivity` implementuje interfejs `Callback` do przechwytywania komunikatów.

❷ Implementacja interfejsu `Callback` przechwytuje komunikaty. Jeśli `msg.what` wynosi `1`, zwraca wartość `true` — komunikat jest obsługiwany. W przeciwnym razie wartość `msg.what` jest zmieniana na `22` i zwracana jest wartość `false` — komunikat nie jest obsługiwany, więc jest przekazywany do implementacji metody `handleMessage` instancji `Handler`.

❸ Przetwarzanie komunikatu w drugiej instancji `Handler`.

❹ Wstawianie komunikatu z parametrem `msg.what == 1`. Komunikat jest przechwytywany przez interfejs `Callback` i zwracana jest wartość `true`.

❺ Wstawianie komunikatu z parametrem `msg.what == 2`. Komunikat jest zmieniany przez interfejs `Callback` i przekazywany do instancji `Handler`, która wyświetla wiadomość `Secondary Handler - msg = 22`.

Usuwanie komunikatów z kolejki

Po dodaniu komunikatu do kolejki producent może wywołać metodę klasy `Handler`, aby usunąć komunikat. Jest to możliwe do momentu, kiedy komunikat zostanie usunięty z kolejki przez `Looper`. Czasami aplikacja może chcieć wyczyścić kolejkę komunikatów poprzez usunięcie wszystkich komunikatów, co jest możliwe, ale najczęściej pożądane jest podejście bardziej wybiórcze: aplikacja chce wybierać tylko część komunikatów. W tym celu musi być w stanie zidentyfikować właściwy komunikat. Dlatego mogą być one identyfikowane na podstawie określonych właściwości, tak jak pokazano w tabeli 4.4.

Tabela 4.4. Identyfikatory komunikatów

Typ identyfikatora	Opis	Komunikaty, do których identyfikator ma zastosowanie
`Handler`	Odbiornik komunikatów	Komunikaty zadań i danych
`Object`	Znacznik komunikatów	Komunikaty zadań i danych
`Integer`	Parametr `what` komunikatu	Komunikaty danych
`Runnable`	Zadanie do wykonania	Komunikaty zadań

Identyfikator `Handler` jest obowiązkowy dla każdego komunikatu, ponieważ komunikat zawsze wie, do której instancji `Handler` zostanie wysłany. Wymóg ten pośrednio ogranicza każdy `Handler` do usuwania jedynie komunikatów należących do niego. Dany `Handler` nie może usuwać z kolejki komunikatów wstawionych przez inny `Handler`.

W klasie `Handler` do zarządzania kolejką komunikatów dostępne są następujące metody:

- usuwanie zadania z kolejki komunikatów:

```
removeCallbacks(Runnable r)
removeCallbacks(Runnable r, Object token)
```

- usuwanie komunikatu danych z kolejki komunikatów:

```
removeMessages(int what)
removeMessages(int what, Object object)
```

- usuwanie zadań i komunikatów danych z kolejki komunikatów:

```
removeCallbacksAndMessages(Object token)
```

Identyfikator `Object` jest używany w komunikatach danych i zadań. Dlatego może być przypisany do komunikatów jako rodzaj znacznika, co pozwala później usunąć powiązane komunikaty, które zostały oznaczone tym samym znacznikiem `Object`.

Poniższy przykładowy fragment kodu umieszcza w kolejce dwa komunikaty w celu umożliwienia ich usunięcia w późniejszym czasie na podstawie znacznika:

```
Object tag = new Object(); ❶

Handler handler = new Handler()
    public void handleMessage(Message msg) {
        // Przetwarzanie komunikatu.
        Log.d("Przykład", "Przetwarzanie komunikatu");
    }
};

Message message = handler.obtainMessage(0, tag); ❷
handler.sendMessage(message);
```

```
handler.postAtTime(new Runnable() { ❸
    public void run() {
        // Pozostawiono puste dla zwięzłości.
    }
}, tag, SystemClock.uptimeMillis());

handler.removeCallbacksAndMessages(tag); ❹
```

❶ Identyfikator znacznika komunikatu wspólny dla komunikatów danych i zadań.

❷ Obiekt w instancji `Message` jest wykorzystywany jako kontener danych i domyślnie zdefiniowany znacznik komunikatu.

❸ Przesłanie komunikatu zadania z bezpośrednio zdefiniowanym znacznikiem komunikatu.

❹ Usunięcie wszystkich komunikatów z określonym znacznikiem.

Jak wskazano wcześniej, nie ma sposobu, aby dowiedzieć się, czy komunikat został rozesłany i obsłużony, zanim nie zostanie przeprowadzone wywołanie w celu jego usunięcia. Po wysłaniu komunikatu wątek producenta, który umieścił go w kolejce, nie może powstrzymać wykonywania jego zadania ani przetwarzania jego danych.

Obserwowanie kolejki komunikatów

Możliwe jest obserwowanie komunikatów oczekujących oraz ich rozsyłania z obiektu `Looper` do powiązanych instancji `Handler`. Platforma Android oferuje dwa mechanizmy obserwacji. Przyjrzyjmy się im na przykładach.

W pierwszym przykładzie pokazano, jak można zarejestrować bieżący zrzut komunikatów oczekujących w kolejce.

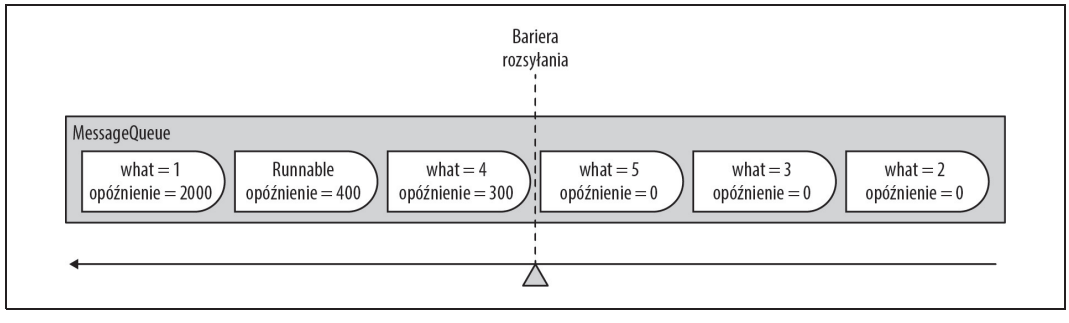
Robienie zrzutu bieżącej kolejki komunikatów

Ten przykład tworzy wątek roboczy, gdy inicjowany jest komponent `Activity`. Kiedy użytkownik naciśnie przycisk, powodując wywołanie metody `onClick`, sześć komunikatów zostanie dodanych do kolejki w różny sposób. Potem obserwujemy stan kolejki:

```
public class MQDebugActivity extends Activity {

    private static final String TAG = "EAT";
    Handler mWorkerHandler;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_mqdebug);

        Thread t = new Thread() {
            @Override
            public void run() {
                Looper.prepare();
                mWorkerHandler = new Handler() {
                    @Override
                    public void handleMessage(Message msg) {
                        Log.d(TAG, "handleMessage - what = " + msg.what);
                    }
                };
                Looper.loop();
            }
        };
```

```
            t.start();
        }

        // Wywoływana po kliknięciu przycisku, czyli z wątku interfejsu użytkownika.
        public void onClick(View v) {
            mWorkerHandler.sendEmptyMessageDelayed(1, 2000);
            mWorkerHandler.sendEmptyMessage(2);
            mWorkerHandler.obtainMessage(3, 0, 0, new Object()).sendToTarget();
            mWorkerHandler.sendEmptyMessageDelayed(4, 300);
            mWorkerHandler.postDelayed(new Runnable() {
                @Override
                public void run() {
                    Log.d(TAG, "Wykonywanie");
                }
            }, 400);
            mWorkerHandler.sendEmptyMessage(5);

            mWorkerHandler.dump(new LogPrinter(Log.DEBUG, TAG), "");
        }
    }
```

Do kolejki zostało dodanych sześć komunikatów z parametrami przedstawionymi na rysunku 4.10.



Rysunek 4.10. Komunikaty dodane do kolejki

Zaraz po dodaniu komunikatów do kolejki zrzut jest zapisywany w dzienniku. Obserwowane są tylko komunikaty oczekujące. Dlatego liczba komunikatów rzeczywiście obserwowanych zależy od tego, ile komunikatów zostało już wysłanych do procedury obsługi. Trzy z tych komunikatów zostały dodane bez opóźnienia, co sprawia, że kwalifikują się one do wysłania w momencie robienia zrzutu.

Typowy przebieg poprzedniego kodu daje następujący zapis dziennika:

```
49.397: handleMessage - what = 2
49.397: handleMessage - what = 3
49.397: handleMessage - what = 5
49.397: Handler (com.eat.MQDebugActivity$1$1) {412cb3d8} @ 5994288
49.407: Looper{412cb070}
49.407: mRun=true
49.407: mThread=Thread[Thread-111,5,main]
49.407: mQueue=android.os.MessageQueue@412cb090
49.407: Message 0: { what=4 when=+293ms }
49.407: Message 1: { what=0 when=+394ms }
49.407: Message 2: { what=1 when=+1s990ms }
49.407: (Total messages: 3)
49.707: handleMessage - what = 4
49.808: Execute
51.407: handleMessage - what = 1
```

Ze zrzutu kolejki komunikatów wynika, że w kolejce oczekują komunikaty z parametrami `what` o wartościach `0`, `1` i `4`. Są to komunikaty dodawane do kolejki z opóźnieniem wysyłki, podczas gdy pozostałe, czyli te bez opóźnienia wysyłki, najwyraźniej zostały już rozesłane. Jest to zrozumiałe, ponieważ przetwarzanie w procedurze obsługi jest bardzo krótkie — trwa tyle co wyświetlenie w dzienniku.

Zrzut pokazuje również, ile czasu pozostało do przejścia każdego komunikatu w kolejce przez barierę rozsyłania. Następny komunikat `Message 0 (what= 4)` przejdzie przez barierę za 293 ms. Komunikaty wciąż oczekują w kolejce, ale te kwalifikujące się do rozesłania będą miały negatywny wskaźnik czasu w dzienniku — np. jeśli parametr `when` jest mniejszy od zera.

Śledzenie przetwarzania kolejki komunikatów

Informacje dotyczące przetwarzania komunikatów można zapisywać w dzienniku. Rejestrowanie kolejki komunikatów jest możliwe z klasy `Looper`. Poniższe wywołanie umożliwia rejestrowanie kolejki komunikatów wątku wywołującego:

```
Looper.myLooper().setMessageLogging(new LogPrinter(Log.DEBUG, TAG));
```

Przyjrzyjmy się przykładowi śledzenia komunikatu, który jest przesyłany do wątku interfejsu użytkownika:

```
mHandler.post(new Runnable() {
    @Override
    public void run() {
        Log.d(TAG, "Wykonywanie zadania Runnable");
    }
});

mHandler.sendEmptyMessage(42);
```

Przykład przesyła dwa zdarzenia do kolejki komunikatów: najpierw zadanie `Runnable`, a następnie pusty komunikat. Jak można było oczekiwać, biorąc pod uwagę sekwencyjne wykonywanie, `Runnable` jest przetwarzany pierwszy i, co za tym idzie, pierwszy zostanie zarejestrowany:

```
>>>>> Dispatching to Handler (android.os.Handler) {4111ef40}
com.eat.MessageTracingActivity$1@41130820: 0
Wykonywanie zadania Runnable
<<<<< Finished to Handler (android.os.Handler) {4111ef40}
com.eat.MessageTracingActivity$1@41130820
```

Ślad wyświetla początek i koniec zdarzenia identyfikowanego przez trzy właściwości:

Instancja Handler
`android.os.Handler 4111ef40`

Instancja zadania
`com.eat.MessageTracingActivity$1@41130820`

Parametr what
`0` (zadania `Runnable` nie przenoszą parametru `what`)

Podobnie, ślad komunikatu z parametrem `what` ustawionym na `42` wyświetla argument komunikatu, ale nie wyświetla żadnej instancji `Runnable`:

```
>>>>> Dispatching to Handler (android.os.Handler) {4111ef40} null: 42
<<<<< Finished to Handler (android.os.Handler) {4111ef40} null
```

Połączenie tych dwóch technik robienia zrzutu kolejki komunikatów oraz śledzenia rozsyłania umożliwia aplikacji szczegółowe obserwowanie przekazywanych komunikatów.

Komunikacja z wątkiem interfejsu użytkownika

Wątek interfejsu użytkownika jest jedynym wątkiem w aplikacji domyślnie posiadającym przypisany `Looper`, który jest kojarzony z wątkiem przed uruchomieniem pierwszego komponentu systemu Android. Wątek interfejsu użytkownika może być konsumentem, do którego inne wątki mogą przekazywać komunikaty. Istotne jest, aby do wątku interfejsu użytkownika przesyłać tylko krótkotrwałe zadania. Wątek interfejsu użytkownika jest globalny dla całej aplikacji i przetwarza sekwencyjnie komunikaty komponentów oraz systemu Android. Dlatego zadania długotrwałe będą miały globalny wpływ na całą aplikację.

Komunikaty są przekazywane do wątku interfejsu użytkownika poprzez jego obiekt `Looper`, który jest dostępny w aplikacji globalnie z poziomu wszystkich wątków za pomocą metody `Looper.getMainLooper()`:

```
Runnable task = new Runnable() {...};
new Handler(Looper.getMainLooper()).post(task);
```

Niezależnie od wątku przesyłającego komunikat jest umieszczany w kolejce wątku interfejsu użytkownika. Jeśli to wątek interfejsu użytkownika przesyła komunikat sam do siebie, komunikat ten może zostać przetworzony najwcześniej po zakończeniu obsługi bieżącego komunikatu:

```
// Metoda wywołana w wątku interfejsu użytkownika.
private void postFromUiThreadToUiThread() {
    new Handler().post(new Runnable() { ... });

    // Kod na tym etapie jest częścią przetwarzanego komunikatu
    // i jest wykonywany przed przesłanym komunikatem.
}
```

Jednak komunikat zadania przesłany przez wątek interfejsu użytkownika do samego siebie może ominąć przekazywanie komunikatów i zostać wykonany natychmiast w aktualnie przetwarzanym komunikacie w wątku interfejsu użytkownika za pomocą metody złożonej `Activity.runOnUiThread(Runnable)`:

```
// Metoda wywołana w wątku interfejsu użytkownika.
private void postFromUiThreadToUiThread() {
    runOnUiThread(new Runnable() { ... });

// Kod na tym etapie jest wykonywany po danym komunikacie.
}
```

Jeśli jest ona wywoływana poza wątkiem interfejsu użytkownika, komunikat zostanie umieszczony w kolejce. Metoda `runOnUiThread` może być wykonana tylko z instancji `Activity`, ale samo zachowanie może być zaimplementowane poprzez śledzenie identyfikatora wątku interfejsu użytkownika, np. za pomocą metody złożonej `customRunOnUiThread` w podklasie `Application`. Metoda `customRunOnUiThread` umieszcza komunikat w kolejce, tak jak pokazano w poniższym przykładzie:

```
public class EatApplication extends Application {
    private long mUiThreadId;
    private Handler mUiHandler;

    @Override
    public void onCreate() {
        super.onCreate();
        mUiThreadId = Thread.currentThread().getId();
```

```
            mUiHandler = new Handler();
        }

        public void customRunOnUiThread(Runnable action) {
            if (Thread.currentThread().getId() != mUiThreadId) {
                mUiHandler.post(action);
            } else {
                action.run();
            }
        }
    }
```

Podsumowanie

Aplikacje systemu Android mają dostęp do standardowych technik komunikacji wątków Javy, które doskonale nadają się do komunikacji wątków roboczych. Rzadko jednak są odpowiednie dla przypadku użycia, kiedy jednym z wątków jest wątek interfejsu użytkownika, co jest najbardziej powszechnym przypadkiem. Przesyłanie komunikatów systemu Android jest szeroko stosowane w całych aplikacjach (bezpośrednio lub pośrednio) poprzez różne techniki zawijania, które zostały omówione w części II tej książki.

Skorowidz

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

O'REILLY®

Android

Aplikacje wielowątkowe. Techniki przetwarzania

Jeżeli chcesz tworzyć aplikacje dostarczające użytkownikom doskonałych wrażeń, szybciej rozwiązywać skomplikowane zadania lub po prostu musisz jednocześnie wykonywać różne działania — naucz się korzystać z wątków. Tworzenie programów wielowątkowych jest trudne, ale opanowanie tej sztuki pozwoli Ci osiągnąć wymierne korzyści!

Ta książka w całości jest poświęcona korzystaniu z wątków na platformie Android. Dzięki niej poznasz przeróżne sposoby asynchronicznego przetwarzania oraz ich zalety i wady. Jednak na samym początku zapoznasz się z podstawowymi informacjami dotyczącymi wielowątkowości w języku Java. Dowiesz się, w jaki sposób wątki komunikują się ze sobą i synchronizują dostęp do zasobów oraz jak nimi zarządzać. Kolejne rozdziały zawierają sporą dawkę wiedzy na temat różnych technik asynchronicznych. Zapoznanie się z ich treścią ułatwi Ci wybór techniki, która spełni Twoje wymagania, gdy sam zabierzesz się do tworzenia aplikacji wielowątkowej. Książka ta jest obowiązkową lekturą dla programistów chcących w pełni wykorzystać możliwości platformy Android.

Dzięki tej książce:

- poznasz znaczenie wątków
- zaznajomisz się z podstawami wielowątkowości w Javie
- poznasz najlepsze techniki asynchroniczne
- sprawnie opanujesz tę niełatwą dziedzinę

Przewodnik po świecie wątków platformy Android!

helion.pl
księgarnia internetowa

Nr katalogowy: 27430

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-9614-7

Cena 49,00 zł