

## Zawartość profilu

Mobile Information Device Profile rozwijany był jako JSR o numerze 37. Profil ten zbudowany jest na podstawie konfiguracji CLDC i rozszerza ją o trzy pakiety w hierarchii `javax.microedition`. Każdy pakiet rozszerza funkcjonalność J2ME w konkretnej dziedzinie. Poniższe zestawienie przedstawia trzy dodane pakiety wraz z opisem ich przeznaczenia.

TABELA2:

Pakiet	Opis
<code>javax.microedition.lcdui</code>	Zawiera klasy pozwalające tworzyć interfejs użytkownika (GUI). Zawiera zarówno klasę <code>Display</code> , będącą logiczną reprezentacją rzeczywistego ekranu, jak i całą hierarchię klas dziedziczących z <code>Displayable</code> , a dzięki temu mogących być wyświetlanych na ekranie. Dzięki nim będziemy mogli budować zarówno tzw. interfejs wysokiego, jak i niskiego poziomu.
<code>javax.microedition.midlet</code>	Zawiera tylko dwie klasy: <code>MIDlet</code> oraz <code>MIDletStateChangeException</code> . Pierwsza klasa definiuje cykl życia MIDletu, i główna klasa każdego MIDletu <i>musi</i> z niej dziedziczyć definiując trzy metody wywoływane przy przejściach pomiędzy stanami życia MIDletu.
<code>javax.microedition.rms</code>	Zawiera klasy pozwalające zarządzać pamięcią stałą telefonu. Tzw. <i>Record Store</i> pozwala przechowywać informacje nawet jeśli MIDlet jest wyłączony, tzn. pomiędzy uruchomieniami. Informacje są w pamięci stałej przechowywane w postaci rekordów.

Dodatkowo MIDP w wersji 1.0 rozszerza konfigurację CLDC dodając do pakietu `java.util` dwie klasy: `Timer` i abstrakcyjną `TimerTask` oraz do pakietu `javax.microedition.io` interfejs `HttpConnection`.

Pierwsze rozszerzenie dostarcza mechanizm wywoływania określonych zadań w regularnych odstępach czasu, drugi natomiast udostępnia połączenie oparte na protokole HTTP.

Dodając w swej specyfikacji nowe możliwości, do tych zawartych już w CLDC, profil MIDP zwiększa wymagania w stosunku do urządzeń, które go implementują. Poniżej wylistowane są minimalne wymagania sprzętowe stawiane przez specyfikację MIDP:

1. PAMIĘĆ. MIDP wymaga 128KB dostępnej pamięci RAM dla samej siebie oraz co

najmniej 32KB pamięci przeznaczonej na stertę (pamięć przeznaczoną dla obiektów uruchamianej aplikacji). Dodatkowo wymagane jest 8KB pamięci stałej dla przechowywania stałych informacji (mowa tutaj o pamięci, w której MIDlety mogą zapisywać informacje, które chcą przechować pomiędzy uruchomieniami - tzw. RecordStore),

2. EKTRAN. Wyświetlacz wbudowany w urządzenie powinien mieć przynajmniej 96 pikseli szerokości i 54 wysokości. Każdy piksel powinien być w przybliżeniu kwadratem. Jednocześnie ekran powinien wyświetlać co najmniej dwa kolory,
3. KLAWIATURA. Minimalne wymagania dot. urządzenia wejścia mówią o klawiaturze pozwalającej wprowadzić cyfry od 0 do 9, klawiszach sterowania i przycisku wyboru. Oczywiście producenci urządzeń mogą emulować obecność tych przycisków. I tak częste jest wykorzystywanie zwykłej klawiatury jako klawiszy sterowania (4 - w lewo, 6 - w prawo, 2 - do góry, 8 - w dół) w telefonach komórkowych.
4. NETWORKING. MIDP wymaga od producentów implementacji protokołu HTTP 1.1. Mogą to oni osiągnąć w różny sposób, w zależności od możliwości urządzenia - poprzez bramę WAP (w telefonach komórkowych) lub przez protokół TCP/IP (to w przypadku urządzeń PDA z modemem).

## Cykl życia MIDletu

Programy uruchamiane w przenośnych urządzeniach nie są zwykłymi aplikacjami. Tak jak główna klasa apletu musi rozszerzać klasę `JApplet`, serwletu HTTP - `HttpServlet`, tak główna klasa MIDletu musi rozszerzać abstrakcyjną klasę `MIDlet` będącą częścią pakietu `javax.microedition.midlet`. Musi też implementować trzy metody definiujące cykl życia MIDletu. Szkielet MIDletu będzie więc wyglądał następująco:

```
public class MyFirstMIDlet extends MIDlet {

    public void startApp() throws MIDletStateChangeException {
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    /** Opcjonalny konstruktor:
     */
    public MyFirstMIDlet() {
    }

}
```

MIDlet musi mieć również publiczny bezargumentowy konstruktor. Nie możemy więc zadeklarować konstruktora w następujący sposób:

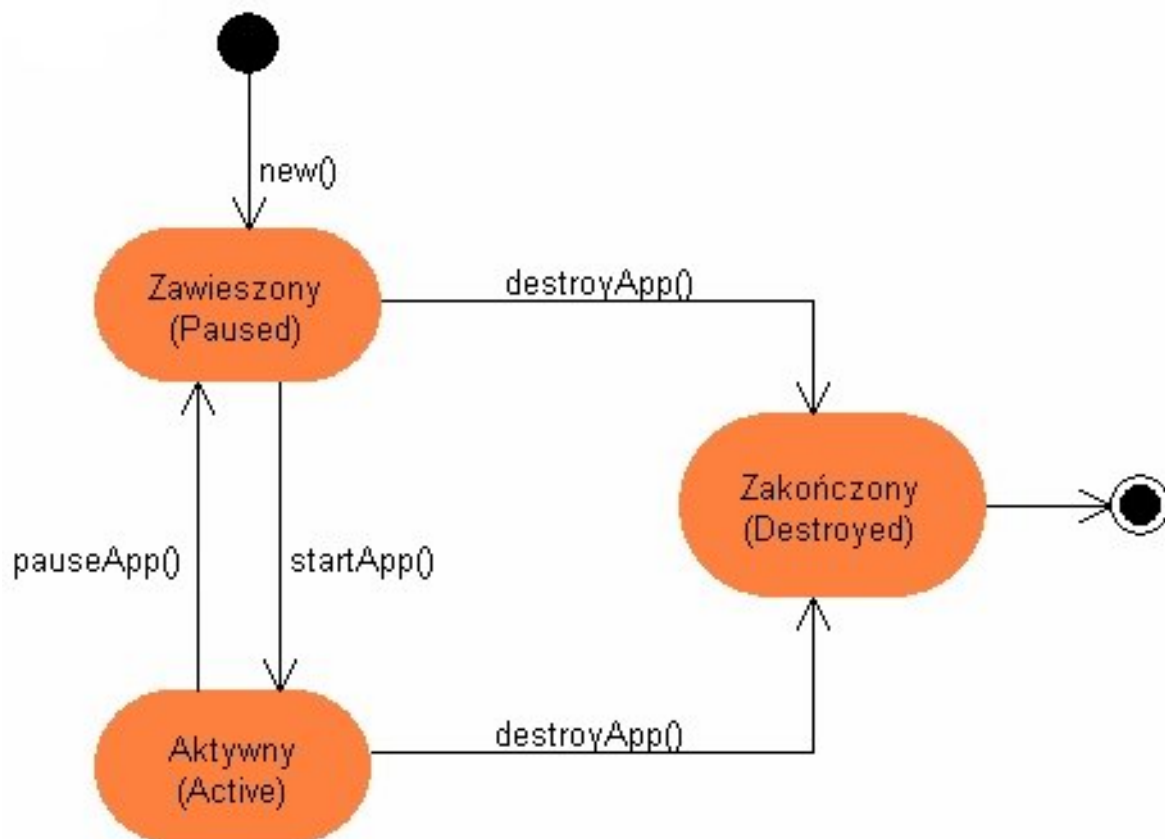
```
private NazwaKlasy() {}
```

Jeśli w przypadku zwykłej aplikacji metoda wywoływana po inicjalizacji obiektów była statyczną metodą `main()`, to tutaj tę rolę pełni metoda `startApp()`:

```
protected void startApp() throws MIDletStateChangeException
```

Może ona być również zadeklarowana jako publiczna. Nie ma to wpływu na działanie programu. Metoda ta reprezentuje stan działania MIDletu. To w ciele tej metody decyduje się, jak przebiegać będzie jego wykonywanie lub ewentualna zmiana stanów.

Stany, w których może znajdować się MIDlet oraz dozwolone przejścia pomiędzy nimi przedstawia poniższy schemat:



*Cykl życia MIDletu.*

Pozostałe dwie abstrakcyjne metody z klasy `MIDlet` definiują kroki, jakie powinny zostać podjęte, aby przygotować MIDlet do przejścia w odpowiedni stan. I tak na przykład przejście w stan zawieszenia wykonywania może się odbyć w dwojaki sposób. Przede wszystkim urządzenie, na którym działa MIDlet, może zawiesić jego działanie. Dzieje się tak na przykład przy przychodzącej rozmowie w przypadku telefonów komórkowych. Urządzenie wywołuje wówczas metodę `pauseApp()` i zawiesza działanie aplikacji. W ciele tej metody powinny więc nastąpić działania przygotowujące MIDlet do przejścia w stan zawieszenia. Być może powinny więc zostać zwolnione wykorzystywane zasoby lub ich część, bądź zawieszona działalność "timerów". W stan zawieszenia może również dobrowolnie wprowadzić siebie aplikacja. Dokonuje tego wywołując metodę `notifyPaused()`. Metoda ta informuje maszynę

wirtualną, iż MIDlet zwolnił zasoby i jest gotowy do przejścia w stan zawieszenia. Maszyna wirtualna może wówczas na przykład przekazać zasoby (jak np. ekran) innemu MIDletowi, który czekał na przejście w stan aktywny. W przypadku wywołania metody `notifyPaused()` platforma nie wywołuje już metody `pauseApp()`. Zakłada, że MIDlet wywołując metodę `notifyPaused()` jest już gotowy do przejścia w stan zawieszenia. Dlatego metoda `pauseApp()` często poprzedza wywołanie metody `notifyPaused()`.

Aby powrócić ze stanu zawieszenia w stan aktywny, MIDlet wywołuje bezargumentową metodę `resumeRequest()`. Ta z kolei informuje maszynę wirtualną, iż MIDlet jest gotowy do działania i jeśli tylko zasoby będą zwolnione, to zostaną mu one przydzielone. W przypadku powrotu ze stanu zawieszenia w stan aktywny, ponownie wywoływana jest metoda `startApp()`. Rodzi to oczywiście niebezpieczeństwo, jeśli nieuważny programista w ciele metody `startApp()` będzie alokował pamięć (poprzez inicjalizację obiektów), że obiekty zostaną zdublowane. Należy więc możliwą inicjalizację przeprowadzić w konstruktorze, a w metodzie `startApp()` wprowadzić flagę mówiącą, czy metoda jest wywoływana po raz pierwszy. Generalnie, w metodzie `startApp()` powinno się inicjalizować obiekty niszczone w `pauseApp()`, bądź te, które przy inicjalizacji wymagają dostępu do ekranu. Dzieje się tak dlatego, iż specyfikacja nie wymaga od maszyny wirtualnej udostępniania ekranu już w konstruktorze. Czy będzie on już tam dostępny zależy od producenta konkretnej KVM.

Analogicznie, metoda `destroyApp()` będzie wywoływana w celu przygotowania MIDletu do zamknięcia. W ciele tej metody powinny więc zostać zwolnione zablokowane wcześniej zasoby, zatrzymane aktywne Timery i ew. w pamięci stałej powinny zostać zapisane informacje, które mają być dostępne przy ponownym uruchomieniu. Metoda ta przyjmuje argument `unconditional` typu `boolean` oraz wyrzuca wyjątek `MIDletStateChangeException`. Kiedy MIDlet jest zamykany przez platformę, metoda ta wywoływana jest z parametrem `true`, co oznacza, że MIDlet zostanie zamknięty bez względu na przebieg metody `destroyApp()`. Jeśli natomiast metoda ta zostanie wywołana z parametrem `false` i wyrzuci wyjątek, oznacza to, iż aplikacja nie jest gotowa do zamknięcia. Można wykorzystać tą informację do odpowiedniego przygotowania MIDletu do zamknięcia. MIDlet można również zakończyć wywołując metodę `notifyDestroyed()`. Platforma nie wywołuje wówczas metody `destroyApp()`. Przyjmuje, iż aplikacja została już przygotowana do zamknięcia.

Jeśli z kolei metoda `startApp()` wyrzuci wyjątek, to:

1. jeśli jest to `MIDletStateChangeException`, to MIDlet przechodzi w stan zawieszenia, także możliwy jest powrót do działania
2. jeśli jest to wyjątek inny niż `MIDletStateChangeException`, to aplikacja jest zamykana poprzez wywołanie `destroyApp()`.

## MIDlet Suits - zestawy MIDletów

MIDlet Suit, to zestaw MIDletów, które dostarczane są do użytkownika i instalowane z pojedynczego archiwum Javy (JAR - Java ARchive). Zestaw ten może być instalowany i usuwany jedynie jako całość (nie można usunąć pojedynczego MIDletu należącego do zestawu). Jednocześnie MIDlet Suit można stworzyć tylko poprzez spakowanie jego

składowych do pojedynczego pliku JAR.

MIDlety należące do zestawu podlegają innym regułom bezpieczeństwa dotyczącym dostępu do pamięci stałej oraz dostępu do klas Javy w maszynie wirtualnej. O pamięci stałej więcej napiszę później. Teraz ważne jest, że każdy MIDlet ma swoją własną przestrzeń w tej pamięci. I tylko on ma do niej dostęp, żadna zewnętrzna aplikacja nie może uzyskać dostępu do naszej przestrzeni w pamięci stałej. Inaczej jest z MIDletami należącymi do jednego zestawu. One współdzielą przestrzeń pamięci stałej. Czyli jeden MIDlet ma dostęp do danych zapisanych przez inny należący do tego samego zestawu.

Pozwala to na budowanie zestawu aplikacji o różnej funkcjonalności, ale korzystających z tych samych danych. Można na przykład stworzyć zestaw MIDletów do składania zamówień w magazynie, w którym pamięć stała będzie wykorzystywana do przechowywania oferty i dostępności produktów. Jeden MIDlet może uaktualniać zawarte tu dane, drugi może służyć do ich przeglądania, a trzeci do składania zamówień. Takie rozbitcie jest dobrym rozwiązaniem, gdyż pamiętamy, że pracujemy na urządzeniach o ograniczonych zasobach. Załadowanie małej aplikacji o odpowiedniej funkcjonalności jest dużo efektywniejsze niż jednej dużej służącej do wszystkiego. Poza tym rozbity w ten sposób łatwiej pielęgnować.

Jeśli natomiast urządzenie, na którym pracujemy umożliwia uruchomienie kilku MIDletów jednocześnie, to te zgromadzone w MIDlet Suit uruchamiane są w pojedynczej maszynie wirtualnej. Oznacza to, że korzystają z tych samych obiektów klas Javy. Rodzi to potrzebę synchronizowania fragmentów kodu, ale i możliwość komunikacji pomiędzy aplikacjami innej niż pamięć stała.