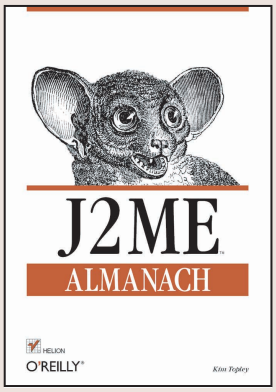
J2ME. Almanach



Autor: Kim Topley
Tłumaczenie: Paweł Gonera (rozdz. 0-5),
Michał Dadan (rozdz. 6-18)
ISBN: 83-7361-118-5
Tytuł oryginału: J2ME in a Nutshell
Format: B5, stron: 544

„J2ME. Almanach" to niezastąpione, podręczne kompendium wiedzy dla programistów korzystających z Java 2 Micro Edition (J2ME). J2ME to nowa rodzina specyfikacji powstałych w firmie SUN, opisujących uproszczoną, skondensowaną wersję platformy Java 2, która może być używana do tworzenia aplikacji działających na urządzeniach o bardzo ograniczonych zasobach, takich jak telefony komórkowe, palmtopy czy dwukierunkowe pagery.

W książce znajdziesz:

- Wprowadzenie do platformy J2ME i środowisk programistycznych, takich jak Java Wireless Toolkit
- Szczegółowy opis możliwości i wymagań CLDC, MIDP i MIDletów
- Dogłębne omówienie interfejsu użytkownika stosowanego w MIDletach oraz wiele praktycznych wskazówek dotyczących wykorzystania MIDP UI API
- Prezentację sposobów w jaki używać Generic Connection Framework API w celu korzystania z bezprzewodowego Internetu, a także API dla przechowywania danych (MDIP Record Management System – RMS)

W książce zawarto znany z innych pozycji serii Almanach wydawnictwa O'Reilly obszerny alfabetyczny spis elementów języka, obejmujący klasy z wielu pakietów J2ME, w tym java.lang, java.io, java.util, java.microediton.io, java.microediton.lcdui, java.microediton.midlet i java.microediton.rms. Gdy zaczniesz pracować z J2ME, „J2ME. Almanach" na stałe zagości na Twoim biurku i będzie podręcznym źródłem informacji.

Tysiące osób codziennie kupują nowe urządzenia wyposażone w możliwość uruchamiania aplikacji Javy. Jeśli chcesz, by były to także Twoje aplikacje, ta książka dostarczy Ci całej wiedzy potrzebnej do ich stworzenia.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

8

Narzędzia uruchamiane z linii poleceń

Programiści J2ME mają do wyboru wiele graficznych środowisk programistycznych, umożliwiających tworzenie i debugowanie aplikacji. O niektórych z nich już wspominaliśmy (lub wspomnimy w rozdziale 9.). W pewnych sytuacjach trzeba jednak posługiwać się niskopoziomowymi narzędziami, niedostępnymi z poziomu IDE. W tym rozdziale omówione zostały te narzędzia wywoływane z linii poleceń, które są najczęściej wykorzystywane przez programistów.

cvm: maszyna wirtualna konfiguracji CDC (Connected Device Configuration)

Dostępność

Wzorcowa implementacja CDC, wzorcowa implementacja profilu Foundation

Składnia

```
cvm [opcje] [właściwości] plikklasy [argumenty]
```

Opis

CVM jest maszyną wirtualną spełniającą wymagania określone w specyfikacji CDC. Ma ona wszystkie cechy, jakie powinna mieć maszyna wirtualna Java 2, a ponadto zawiera garbage collector zoptymalizowany do pracy w środowiskach o niewielkich zasobach

pamięciowych. W celu skrócenia czasu potrzebnego na uruchomienie maszyny wirtualnej i ograniczenia jej wymagań odnośnie pamięci, zazwyczaj już na etapie jej kompilowania łączy się z nią podstawowe klasy języka Java, tak samo jak w przypadku maszyny wirtualnej KVM (patrz rozdział 2.).

CVM dostarczana jest w postaci kodu źródłowego i stanowi część wzorcowych implementacji CDC i profilu Foundation, dostępnych na platformy Linux oraz VxWorks.

Opcje

`-version`
Wyświetla informacje na temat wersji maszyny wirtualnej, po czym kończy jej działanie. Po wywołaniu tego polecenia na ekranie zazwyczaj pojawia się coś takiego:

```
java version „J2ME Foundation 1.0”
Java(TM) 2, Micro Edition (build 1.0fcs-ar)
CVM (build .0fcs-ar, native threads)
```

`-showversion`
Wyświetla te same informacje co `-version`, przy czym maszyna wirtualna nie jest zatrzymywana. Dzięki tej opcji informacje o wersji VM można zwrócić przed uruchomieniem aplikacji.

`-fullversion`
Powoduje zwrócenie mniejszej liczby informacji niż opcja `-version`. Zazwyczaj wygląda to tak:

```
java full version „1.0fcs-ar”
```

Po wyświetleniu tych danych maszyna wirtualna jest zatrzymywana.

`-Xbootclasspath:list`
`-Xbootclasspath=list`
Lista katalogów, plików JAR oraz plików ZIP, w których maszyna wirtualna będzie szukać klas uruchomieniowych, które nie zostały z nią zlinkowane na etapie kompilacji. Poszczególne elementy na liście powinny być oddzielane dwukropkami (Linux) lub średnikami (VxWorks).

`-Xbootclasspath/p:list`
`-Xbootclasspath/p=list`
Umieszcza wskazane katalogi, pliki JAR oraz ZIP na początku istniejącej listy klas uruchomieniowych. Poszczególne elementy na liście powinny być oddzielane dwukropkami (Linux) lub średnikami (VxWorks).

`-Xbootclasspath/a:list`
`-Xbootclasspath/a=list`
Dołącza wskazane katalogi, pliki JAR oraz ZIP na końcu istniejącej listy klas uruchomieniowych. Poszczególne elementy na liście powinny być oddzielane dwukropkami (Linux) lub średnikami (VxWorks).

`-Xss`*`rozmiar`*

Ustawia rozmiar natywnego stosu wątków na *`rozmiar`* bajtów. Aby określić rozmiar w kilobajtach lub megabajtach, dołącz do wprowadzanej liczby literę `k`, `K`, `m` lub `M`. Zauważ, że między znakami `-Xss` a wprowadzaną liczbą nie może być spacji. Tak więc opcja `-Xss1m` jest zapisana poprawnie, ale `-Xss 1m` już nie.

`-Xms`*`rozmiar`*

Ustawia rozmiar sterty (ang. *heap*). Zmiennej rozmiar nadaje się wartość w taki sam sposób, jak w przypadku opcji `-Xss`. Wartość ta może zostać zaokrąglona przez maszynę wirtualną do takiej, jaka będzie dla niej wygodna.

`-Xgc:`*`opcje_dla_gc`*

Przekazuje opcje dla garbage collectora. Zakres możliwych do stosowania opcji zależy od konkretnej implementacji `gc`.

`-Xverify:`*`typ`*

Określa zakres weryfikacji klas. Jeżeli nie określono *`typu`* lub wprowadzono wartość `all`, wówczas wszystkie wczytywane klasy podlegają weryfikacji. Chcąc poddawać weryfikacji jedynie klasy wgrywane zdalnie, wprowadź wartość `remote`, natomiast chcąc wyłączyć mechanizm weryfikacji, wpisz `none`. Jeżeli ta opcja zostanie pominięta, weryfikowane będą jedynie zdalnie wczytywane klasy.

`-Xdebug`

Uaktywnia w maszynie wirtualnej obsługę debugowania JPDA. Tę opcję wykorzystuje się wraz z `-Xrunjdwp` i można z niej korzystać tylko wtedy, jeżeli maszyna wirtualna została skompilowana w taki sposób, aby obsługiwała JVMDI. Więcej informacji na ten temat znajdziesz w jednym z następnych podrozdziałów, zatytułowanym „Debugowanie”..

`-Xrun`*`nazwa`*`:`*`opcje`*

Wczytuje do maszyny wirtualnej bibliotekę kodu natywnego o podanej *`nazwie`*. Ten argument może wystąpić więcej niż jeden raz, co pozwala wczytać większą liczbę bibliotek. Na każdej platformie ciąg *`nazwa`* zamieniany jest na nazwę biblioteki w inny sposób. W Linuksie stosowany jest wzorzec *nazwabibl.so*, natomiast w systemie VxWorks — *nazwabibl.o*. Jeżeli jednak maszyna wirtualna została skompilowana w taki sposób, aby umożliwiała debugowanie aplikacji, stosowane są odpowiednio wzorce *nazwabibl_g.so* oraz *nazwabibl _g.o*. Między `-Xrun` a nazwą biblioteki nie mogą pojawić się żadne spacje. Tak więc aby wczytać bibliotekę *libjdwp.so* (lub *libjdwp.o*), trzeba wprowadzić argument `-Xrunjdwp`. Zmienne systemowe `sun.boot.library.path` oraz `java.library.path` określają lokalizacje, które będą przeszukiwane w poszukiwaniu bibliotek.

Po nazwie biblioteki może opcjonalnie wystąpić grupa opcji, oddzielonych od niej dwukropkiem. Format i znaczenie tych opcji zależą od biblioteki. Jeżeli zawiera ona funkcję `JVM_onLoad()`, to jest ona wywoływana w chwili wczytywania biblioteki, a jednemu z jej argumentów przypisywany jest ciąg znaków, zawierający wprowadzone opcje. W jednym z kolejnych podrozdziałów, zatytułowanym „Debugowanie”, znajdziesz przykład wykorzystania tego mechanizmu.

`-Xtrace:`*`wartość`*

Włącza w maszynie wirtualnej niskopoziomowe śledzenie przebiegu wykonywania programu. Argument *`wartość`* określa, co dokładnie ma być śledzone. Powstaje on poprzez dodanie do siebie wybranych wartości z poniższej tabeli. Ta opcja jest dostępna tylko wtedy, jeżeli maszyna wirtualna została skompilowana w taki sposób, aby umożliwiała debugowanie.

Wartość	Co będzie śledzone:
`0x0000001`	Wykonywanie kodu pośredniego (ang. *byte-code*)
`0x0000002`	Wykonywanie metod
`0x0000004`	Wewnętrzny stan pętli interpretera w chwili wywołania każdej metody i powrotu z niej
`0x0000008`	Szyba ścieżka synchronizacji
`0x0000010`	Wolna ścieżka synchronizacji
`0x0000020`	Operacje blokowania i odblokowywania muteksów
`0x0000040`	Spójne zmiany stanu
`0x0000080`	Początek i koniec operacji oczyszczania pamięci
`0x0000100`	Skanowanie zasobów głownych przez garbage collector
`0x0000200`	Skanowanie obiektów na stercie przez garbage collector
`0x0000400`	Alokacja obiektów
`0x0000800`	Wewnętrzne dane garbage collectora
`0x0001000`	Przejścia między bezpiecznymi a niebezpiecznymi stanami garbage collectora
`0x0002000`	Wykonywanie statycznych inicjalizatorów klas
`0x0004000`	Obsługa wyjątków języka Java
`0x0008000`	Inicjalizacja i niszczenie sterty, globalna inicjalizacja stanu oraz bezpieczne wyjścia
`0x0010000`	Bariery odczytu i zapisu dla garbage collectora
`0x0020000`	Tworzenie map garbage collectora dla stosów języka Java
`0x0040000`	Wczytywanie klas
`0x0080000`	Sprawdzanie klas w wewnętrznych tablicach maszyny wirtualnej
`0x0100000`	Operacje systemowe na typach
`0x0200000`	Weryfikacja klas
`0x0400000`	Obsługa słabych referencji
`0x0800000`	Usuwanie klas z maszyny wirtualnej
`0x1000000`	Łączenie klas

Zmienne systemowe

Za pomocą argumentów mających następującą postać:

```
-Dnazwa=wartość
```

można nadawać `wartości` zmiennym o określonej `nazwie`, przechowywanym w tablicy zmiennych systemowych. Można je później odczytywać w kodzie programu:

```
String value = System.getProperty(„name”);
```

Wiele z nich ma specjalne znaczenie dla maszyny wirtualnej i podstawowych bibliotek klas. W szczególności poniższe zmienne systemowe wpływają na sposób wczytywania klas języka Java i bibliotek natywnego kodu:

`java.class.path`
: Interpretowana jako lista katalogów, plików JAR oraz plików ZIP, z których będą wczytywane klasy aplikacji. Poszczególne wpisy na liście powinny być od siebie oddzielane dwukropkami (Linux) lub średnikami (VxWorks). Ta zmienna CVM jest odpowiednikiem zmiennej środowiskowej `CLASSPATH` występującej w J2SE, przy czym jej wartość musi być określona ręcznie, ponieważ CVM nie sprawdza zawartości zmiennej `CLASSPATH`. W jednym z kolejnych podrozdziałów, o nazwie „Przykłady”, znajdziesz przykład wykorzystania tej zmiennej systemowej w praktyce.

`sun.boot.class.path`
: Ta zmienna pełni taką samą rolę jak `java.class.path`, z tym że określa ona położenie klas systemowych (uruchomieniowych). Jej wartość można określić w wygodny sposób, wprowadzając jedną z opcji `-Xbootclasspath`.

`java.library.path`
: Lista katalogów oddzielonych od siebie dwukropkami (Linux) lub średnikami (VxWorks), które będą przeszukiwane w poszukiwaniu bibliotek z natywnym kodem, wykorzystywanych przez JNI.

`sun.boot.class.path`
: Określa kolejność, w jakiej będą przeszukiwane biblioteki w poszukiwaniu klas uruchomieniowych. Jest to systemowy odpowiednik `java.library.path`.

Debugowanie

Jeżeli maszyna wirtualna CVM została skompilowana z włączoną obsługą JVMDI, istnieje możliwość debugowania programu na poziomie kodu źródłowego, podłączając do CVM debugger JPDA. Oczywiście przy założeniu, że maszyna wirtualna została uruchomiona z opcjami `-Xdebug` oraz `-Xrunjdwp`. Druga z nich sprawia, że zostanie wczytana biblioteka zawierająca implementację JDWP. Opcja `-Xrunjdwp` wymaga podania dodatkowych parametrów, dzięki którym biblioteka wie, jak ma się skonfigurować. Ogólna postać tego argumentu wygląda następująco:

```
-Xrunjdwp:parametr=wartość[,parametr=wartość....]
```

Niżej wymieniono wszystkie możliwe parametry i opisano ich znaczenie:

`transport=`*`typ`*
: Określa sposób przekazywania informacji, który powinien być wykorzystywany przez maszynę wirtualną w komunikacji z debuggerem. W chwili pisania tej książki obsługiwane były jedynie gniazda, a więc jedynym możliwym do wprowadzenia typem był `dt_socket`.

`address=`*`wartość`*
: Adres, pod którym maszyna wirtualna powinna oczekiwać na połączenia inicjowane przez debugger. Format, w jakim należy wprowadzić wartość, zależy od wartości parametru `transport`. W przypadku wartości `dt_socket` trzeba podać numer portu TCP/IP.

`server=y|n`
: Określa, czy biblioteka JDWP powinna pełnić rolę serwera (wartość `y`) czy klienta (wartość `n`). Jeżeli chcesz odbierać połączenia od debuggera JPDA, powinieneś wprowadzić wartość `y`.

`suspend=y|n`
: Jeżeli opcja ta ma wartość *`y`*, co jest przyjmowane domyślnie, to w chwili inicjalizacji kodu JVMDI maszyna wirtualna jest zatrzymywana do czasu, aż podłączy się do niej jakiś debugger. Normalnie ma to miejsce w momencie uruchamiania maszyny wirtualnej, ale inicjalizację mechanizmów debugowania można odroczyć do czasu pojawienia się *nieobsłużonego wyjątku* lub *wyjątku określonego typu*.

`strict=y|n`
: Określa, czy mechanizmy CVM odpowiedzialne za debugowanie aplikacji mają zachowywać się w pełni zgodnie ze specyfikacją JVMDI. Domyślnie opcja ta ma wartość `n` i raczej nie będziesz z niej często korzystał.

`onuncaught=y|n`
: Domyślnie debugger inicjalizowany jest przy starcie maszyny wirtualnej. Jeżeli jednak nadasz tej opcji wartość *`y`*, proces ten zostanie odroczony do chwili pojawienia się nieobsłużonego wyjątku. Pozwala to uniknąć spowolnienia pracy maszyny wirtualnej do czasu wystąpienia błędu.

`onthrow=`*`wartość`*
: Ta opcja jest podobna do `onuncaught`, z tym że odracza ona inicjalizację debuggera do czasu zgłoszenia konkretnego wyjątku (nie ma przy tym znaczenia, czy zostanie on obsłużony, czy nie). Argument *`wartość`* zawiera nazwę klasy interesującego nas wyjątku. Uruchomienie debuggera nastąpi gdy tylko zostanie zgłoszony jakiś wyjątek tej konkretnej klasy. Na przykład aby zainicjalizować debugger w chwili zgłoszenia wyjątku `NullPointerException`, napisz:

```
onthrow=java.lang.NullPointerException
```

`stdalloc=y|n`
: Gdy ta opcja ma wartość `y`, stosowane są standardowe metody alokacji pamięci z biblioteki C. W przeciwnym wypadku maszyna wirtualna korzysta z własnego pakietu alokacji pamięci. Jest to zachowanie domyślne.

`launch=wartość`
: Sprawia, że po zainicjalizowaniu debuggera uruchamiana jest aplikacja, której nazwę przekazano w zmiennej `wartość`. Do aplikacji tej przekazywane są dwa parametry: nazwa (np. *dt_socket*) i adres połączenia, za pośrednictwem którego debugger komunikuje się z maszyną wirtualną. Parametr `wartość` powinien zawierać ścieżkę bezwzględną lub nazwę pliku wykonywalnego, znajdującego się w jednym z katalogów przeszukiwanych przez maszynę wirtualną.

Przykłady

```
cvm -Xbootclasspath:../lib/foundation.jar -Djava.class.path=/
home/user/project mojPakiet.mojaKlasa
```
: Wczytuje i uruchamia aplikację Javy, zaczynając od metody `main()` klasy `mojPakiet.mojaKlasa`. Klasy podstawowe, których nie wbudowano w maszynę wirtualną, wczytywane są z pliku *../lib/foundation.jar*, natomiast klasy aplikacji znajdują się w katalogu */home/user/project*.

```
cvm -Xbootclasspath:../lib/foundation.jar -Djava.class.path=/
home/user/project -Djava.library.path=/home/user/nativecode/
lib mojPakiet.mojaKlasa
```
: Uruchamia tę samą aplikację, przy czym tym razem biblioteki natywne wczytywane są z katalogu */home/user/nativecode/lib*.

Patrz również

- Dokument „The Java 2 Platform, Micro Edition Connected Device Configuration (CDC) 1.0 Porting Guide" w archiwum zawierającym wzorcową implementację CDC lub profilu Foundation
- „Usuwanie błędów z programów napisanych w Javie w CVM" w rozdziale 7. Znajduje się tam przykład pokazujący, jak uaktywnia się debugowanie JPDA.

kdp: KVM Debug Proxy

Dostępność

Wzorcowa implementacja CLDC

Składnia

```
java kdp.KVMDebugProxy [opcje]
```

Opis

kdp jest aplikacją napisaną w języku Java, pełniącą funkcję pośrednika między debuggerem zgodnym z JPDA a maszyną wirtualną, taką jak KVM. W pełni rozbudowane maszyny wirtualne, takie jak te dostarczane z J2SE, mają własne, wbudowane implementacje JDWP, pozwalające im bezpośrednio łączyć się z debuggerem. Natomiast ograniczenia zasobów, charakterystyczne dla typowych maszyn wirtualnych CLDC, nie pozwalają na pełną implementację JDWP. kdp, czyli proxy debuggera, umiejscawia się między debuggerem a maszyną wirtualną, co pozwala zdjąć „z barków" tej ostatniej niektóre szczegóły implementacyjne.

kdp zazwyczaj uruchamiane jest na komputerze typu desktop, dzięki czemu nie uszczupla ono zasobów platformy docelowej. Komunikuje się z KVM za pomocą gniazd, wykorzystując okrojoną wersję JDWP o nazwie KDWP (od *KVM Debug Wire Protocol*).

Opcje

`-classpath` *`ścieżka`*
: Określa położenie kopii plików klas, które zostaną wczytane do debugowanej maszyny wirtualnej. Wskazane lokalizacje, czyli nazwy katalogów lub plików JAR, oddzielane są od siebie separatorem charakterystycznym dla danej platformy. Na przykład w systemie Windows jest to średnik, a pod Uniksem dwukropek). Ta opcja jest wymagana tylko wtedy, gdy stosuje się opcję `-p`.

`-cp` *`ścieżka`*
: Synonim `-classpath`.

`-l` *`lokalnyport`*
: Numer portu, na którym proxy debuggera oczekuje na połączenia inicjowane przez debugger zgodny z JPDA.

`-p`
: Jeżeli zostanie wprowadzony ten argument, proxy debuggera będzie obsługiwało operacje dotyczące klas lokalnie, zamiast przekazywać je do docelowej maszyny wirtualnej. Argument ten stosuje się w sytuacji, gdy debugujemy za pomocą KVM i chcemy przenieść ciężar przechowywania informacji o poszczególnych klasach z maszyny wirtualnej na proxy debuggera. Trzeba też wprowadzić opcję `-cp` lub `-classpath`, tak aby proxy mogło wczytać także klasy wykorzystywane przez samą maszynę wirtualną.

`-r` *`host port`*
: Nazwa lub adres IP hosta, na którym pracuje debugowana maszyna wirtualna, oraz port, na którym oczekuje ona na połączenia pochodzące od proxy debuggera. Numer portu ustawia się za pomocą argumentu `-port` maszyny KVM.

`-v` *`szczegółowość`*
: Umożliwia wyświetlanie informacji pochodzących z debuggera i określa stopień ich szczegółowości. Argument *`szczegółowość`* przyjmuje wartości z zakresu od `1` do `9`, gdzie większa wartość oznacza bardziej szczegółowe debugowanie.

Przykłady

Aby uruchomić proxy debuggera i podłączyć je do maszyny wirtualnej KVM prowadzącej nasłuch na porcie 2000 na tej samej maszynie, wczytać pliki klas ze ścieżki określonej w zmiennej środowiskowej `CP` oraz oczekiwać na połączenia zainicjowane przez debugger na porcie 3000, należy wprowadzić polecenie:

```
java kdp.KVMDebugProxy -l 3000 -p -r localhost 2000 -cp %CP%
```

Patrz również

- *kvm*
- *Specyfikacja protokołu Debug Wire Protocol* dostępna w archiwum ze wzorcową implementacją CLDC

kvm: Kilobyte Virtual Machine

Dostępność

Wzorcowa implementacja CLDC

Składnia

```
kvm [opcje] plikklasy [argumenty]
```

Opis

Polecenie `kvm` stanowi wzorcową implementację maszyny wirtualnej Javy, spełniającą wymagania specyfikacji CLDC. KVM potrafi wczytywać klasy z dowolnego miejsca w strukturze katalogów lokalnego systemu plików bądź z plików JAR. Z myślą o zmniejszeniu wymagań odnośnie pamięci i skróceniu czasu uruchamiania maszyny wirtualnej, KVM ma już zazwyczaj wkompilowane podstawowe biblioteki klas.

Polecenie `kvm` dostępne we wzorcowej implementacji CLDC nie umożliwia debugowania kodu. Dostępna jest jednak jego druga wersja, `kvm_g`. Współpracuje ona z proxy debuggera o nazwie *kdp*. Oferuje ona też zestaw dodatkowych opcji wprowadzanych z linii poleceń, dzięki którym można zażyczyć sobie, aby informacje zbierane przez debugger były kierowane do standardowego strumienia wyjściowego. Istnieje także możliwość skompilowania wersji KVM implementującej menedżera aplikacji (JAM), dzięki któremu można pobierać aplikacje z sieci i instalować je w lokalnym systemie plików. Jednak zazwyczaj się z tego nie korzysta, ponieważ w większości przypadków stosuje się bardziej zaawansowane rozwiązania, oferowane przez *emulatory* urządzeń i inne polecenia *midp*.

Opcje

Następujące opcje dostępne są dla wszystkich wersji KVM:

`-version`
: Wyświetla numer wersji wzorcowej implementacji CLDC, po czym kończy pracę maszyny wirtualnej.

`-classpath` *ścieżka*
: Określa położenie plików klas, które mają być wczytane przez maszynę wirtualną. Na tej liście mogą znaleźć się nazwy katalogów lub plików JAR, oddzielone od siebie separatorem charakterystycznym dla danej platformy. Na przykład w systemie Windows jest to średnik, a pod Uniksem dwukropek. Omawianej zmiennej można też przypisać wartość zmiennej środowiskowej `CLASSPATH`.

`-heapsize` *rozmiar*
: Ustawia rozmiar sterty (ang. *heap*), zastępując wartość domyślną dla danej implementacji. Parametr *rozmiar* może mieć wartość bezwzględną, wyrażoną w bajtach (na przykład `131072`), bądź zapisaną skrótowo, np. `512k` lub `2M`. Nie może ona być mniejsza niż `32K`, ani większa niż `64M`.

`-help`
: Wyświetla składnię polecenia i listę wszystkich dostępnych opcji, po czym kończy działanie maszyny wirtualnej.

Jeżeli maszyna wirtualna KVM została skompilowana z obsługą debugowania JPDA, dostępne są następujące, dodatkowe opcje:

`-debugger`
: Uaktywnia w maszynie wirtualnej debugowanie JDPA.

`-port` *numer*
: Określa numer portu, na którym maszyna wirtualna będzie odbierała połączenia pochodzące z proxy debuggera KVM. Przy braku jawnego wprowadzenia tej opcji stosowany jest port 2800.

`-suspend`
: Sprawia, że maszyna wirtualna wstrzymuje wykonywanie aplikacji do czasu, aż zostanie poproszona przez zdalny debugger o jej wznowienie. Jest to domyślne postępowanie w sytuacji, gdy zostanie wprowadzony argument `-debugger`.

`-nosuspend`
: Jeżeli wprowadzono argument `-debugger`, wykonywanie aplikacji nie zostanie wstrzymane, a maszyna wirtualna nie będzie czekać na to, aż podłączy się do niej debugger.

Jeżeli w maszynę wirtualną wbudowano mechanizm śledzenia wykonywania programu, dostępne są następujące dodatkowe opcje:

`-traceall`
: Uaktywnia wszystkie opisane poniżej opcje dotyczące śledzenia wykonywania programu.

`-traceallocation`
Umożliwia śledzenie przydzielania pamięci. Zapisywane są informacje o rozmiarze każdego zaalokowanego bloku oraz o ilości pozostałej wolnej pamięci.

`-tracebytecodes`
Umożliwia śledzenie wykonywania każdej instrukcji kodu pośredniego. Zwracane informacje zawierają nazwę instrukcji, wartości jej operandów oraz nazwę metody, której jest ona częścią.

`-traceclassloading`
Śledzi wczytywanie i inicjalizację klas.

`-traceclassloadingverbose`
Śledzi wczytywanie i inicjalizację klas oraz dostarcza więcej informacji zwrotnych niż `-traceclassloading`.

`-tracedebugger`
Śledzi operacje debuggera.

`-traceevents`
Uaktywnia śledzenie zdarzeń (takich jak poruszenie pisakiem) zgłaszanych przez platformę sprzętową. Zapamiętywany jest jedynie rodzaj zdarzenia.

`-traceexceptions`
Zapisuje informacje zgromadzone przez debugger za każdym razem, gdy wystąpi jakiś wyjątek.

`-traceframes`
Śledzi odkładanie ramek na stos i ich zdejmowanie, czyli zdarzenia zachodzące w chwili wywołania i wyjścia z metody.

`-tracegc`
Zapamiętuje czas, kiedy garbage collector rozpoczął i zakończył pracę, a także liczbę bajtów pamięci, którą uwało mu się uwolnić.

`-tracegcverbose`
Zwraca te same informacje co `-tracegc`, a ponadto informuje o obiektach, którym garbage collector się przygląda.

`-tracemethods`
Śledzi wejście i wyjście z każdej metody, zapisując nazwę klasy i metody.

`-tracemethodsverbose`
Śledzi wejście i wyjście z każdej metody, zapisując rodzaj wywołania (virtual, static, special, interface, etc.), nazwę klasy, nazwę metody oraz sygnaturę metody.

`-tracemonitors`
Śledzi aktywność monitora. Za pomocą monitorów kontroluje się zsynchronizowane metody i bloki kodu.

`-tracenetworking`
Śledzi aktywność sieciową.

`-tracestackchunks`
Śledzi tworzenie nowych stosów (gdy tworzony jest nowy wątek) oraz odkładanie ramek wykonywania (ang. *execution frames*) na stos i zdejmowanie ich z niego (podobnie jak `-traceframes`).

`-tracestackmaps`
Śledzi operacje wykonywane na mapach stosu. Mapy stosu umożliwiają zapisywanie na stosie informacji o bieżących referencjach, które są potem wykorzystywane przez garbage collector.

`-tracethreading`
Śledzi zdarzenia odnoszące się do wątków, takie jak:

Utworzenie wątku
Uruchomienie wątku
Przełączanie wątków
Zatrzymanie wątku
Wstrzymanie wątku
Wznowienie wątku

`-traceverifier`
Śledzi pracę weryfikatora kodu pośredniego.

Jeżeli w maszynę wirtualną wkompilowano menedżera aplikacji KVM JAM, można korzystać z następujących opcji:

`-jam`
Umożliwia korzystanie z menedżera aplikacji. Jeżeli wprowadzisz tę opcję, musisz też pamiętać o argumencie `classfile`, który powinien zawierać adres URL pliku deskryptora, opisującego aplikację, która ma zostać wczytana i uruchomiona. Format tego pliku opisany jest w dokumentacji „KVM Porting Guide", wchodzącej w skład archiwum zawierającego wzorcową implementację CLDC, które można pobrać z Internetu.

`-appsdir` *`katalog`*
Określa katalog lokalny, w którym mają być instalowane aplikacje wczytywane przez JAM.

`-repeat`
Wczytuje i wykonuje na okrągło aplikację, której klasę wskazano w linii poleceń, do czasu, aż użytkownik nie przerwie tej „karuzeli".

Przykłady

`kvm -classpath mojaApl.jar com.mojafirma.MojaApl`
Wczytuje i uruchamia klasę `com.mojafirma.MojaApl`, szukając klas w pliku JAR o nazwie *mojaApl.jar*.

`kvm_g -traceall -classpath mojaApl.jar com.mojafirma.MojaApl`
Wczytuje i uruchamia klasę `com.mojafirma.MojaApl`, szukając klas w pliku JAR o nazwie *mojaApl.jar*. Zostaje uaktywnione zapisywanie wszystkich informacji debuggera. Uważaj, bo będzie ich naprawdę dużo.

```
kvm_g -debugger -port 2850 -classpath mojaApl.jar com.mojafirma.MojaApl
```

Wczytuje klasę `com.mojafirma.MojaApl`, szukając klas w pliku JAR o nazwie *mojaApl.jar* i wstrzymuje jej wykonywanie do czasu, aż na porcie 2850 pojawi się połączenie zainicjowane przez debugger.

Patrz również

- Dokument „KVM Porting Guide", który znajdziesz w archiwum zawierającym wzorcową implementację CLDC

midp: środowisko wykonywania aplikacji zgodnych z profilem MID (MID Profile Execution Environment)

Dostępność

Wzorcowa implementacja MIDP

Składnia

```
midp [opcje]

midp [opcje] [-Xdescriptor nazwapliku] klasa

midp [opcje] -Xdescriptor nazwapliku

midp [opcje] -autotest URL_deskryptora [nazwa_MIDletu]

midp [opcje] -transient URL_deskryptora [nazwa_MIDletu]

midp [opcje] -install [-force] URL_deskryptora

midp [opcje] -run (numer zestawu | nazwa pod jaką MIDlet zostanie
zachowany) [nazwa_MIDletu]

midp [opcje] -remove (numer zestawu | nazwa pod jaką MIDlet
zostanie zachowany | all)

midp [opcje] -list

midp [opcje] -storageNames
```

Opis

midp jest programem wykonywalnym, zawierającym implementację maszyny wirtualnej KVM, klasy wymagane przez profil MIDP w wersji 1.0, a także implementację podsystemu zarządzania aplikacjami (*Application Management Software*). Jest to więc kompletne środowisko, w którym można testować i instalować MIDlety.

Zarządzanie MIDletami i ich przechowywanie

MIDlet składa się z jednego lub większej liczby plików klas oraz ze skojarzonych z nimi zasobów, przechowywanych w pliku JAR. Można też połączyć kilka MIDletów w tak zwany zestaw. Wszystkie MIDlety wchodzące w skład jednego zestawu zapisane są w tym samym pliku JAR i zarządza się nimi tak, jak gdyby były one pojedynczym MIDletem. Gdy korzystasz z polecenia `midp`, są one instalowane w symulowanej pamięci nieulotnej jako niepodzielna całość i na tej samej zasadzie odbywa się ich dezinstalacja. Co więcej, są one wykonywane na tej samej kopii maszyny wirtualnej.

Gdy chcesz przeprowadzić jakieś testy, możesz wczytywać MIDlety z lokalnego systemu plików, ale w praktyce prawie zawsze będą one pobierane z sieci lub wgrywane przez połączenie lokalne z jakimś hostem, na przykład komputerem typu desktop. Ponieważ plik JAR zawierający zestaw MIDletów może mieć znaczne rozmiary, z każdym zestawem skojarzony jest plik opisu archiwum (JAD), który jest na tyle mały, że można go szybko pobrać z sieci i który zawiera informacje o zestawie, na podstawie których użytkownik może zdecydować, czy chce go zainstalować. Oprogramowanie zarządzające aplikacjami (AMS) pracujące na urządzeniu MIDP (takim jak telefon komórkowy) zazwyczaj najpierw pobiera plik JAD, którego lokalizacja określana jest przez adres URL. Jeżeli użytkownik zdecyduje się zainstalować dany zestaw MIDletów, AMS ściąga plik JAR, którego położenie można odczytać wśród informacji zapisanych w pliku JAD. Następnie zestaw MIDletów zapisywany jest na urządzeniu, co umożliwia ich wczytywanie.

Składnia polecenia `midp` ma wiele wariantów. Umożliwiają one uruchamianie MIDletów na wiele różnych sposobów i korzystanie z zestawu funkcji zarządzających, obsługiwanych przez AMS.

Wykonanie MIDletu bez instalowania go na stałe

Jeżeli tylko testujesz aplikację, możesz ją wykonać lub wczytać zestaw MIDletów i wybrać ten z nich, który chcesz uruchomić, bez zapisywania go na stałe do symulowanej pamięci nieulotnej urządzenia. Najprostszym sposobem na uruchomienie wybranego MIDletu jest skorzystanie z tej postaci polecenia `midp`, w której wymagane jest określenie nazwy klasy MIDletu. Na przykład:

```
midp -classpath . ora.ch4.FormExampleMIDlet
```

Ta postać polecenia `midp` przydaje się do testowania MIDletów, które nie zostały jeszcze spakowane do pliku JAR. Jeżeli MIDlet musi odwoływać się do właściwości aplikacji,

zapisanych w pliku JAD, możesz wprowadzić argument `-Xdescriptor` i określić lokalizację pliku deskryptora, który ma zostać w tym celu użyty. Musi to być nazwa pliku występującego w lokalnym systemie plików:

```
midp -classpath . -Xdescriptor ora\ch4\Chapter4.jad ora.ch4.FormExampleMIDlet
```

Aby wywołać zestaw MIDletów i pozwolić użytkownikowi wybrać MIDlet, który ma zostać wykonany, wprowadź nazwę pliku JAD zestawu, ale pomiń nazwę klasy:

```
midp -classpath . -Xdescriptor ora\ch4\Chapter4.jad
```

Ta odmiana polecenia `midp` wymaga, aby zestaw MIDletów był spakowany w pliku JAR, którego nazwa figuruje w pliku deskryptora aplikacji pod atrybutem `MIDlet-Jar-URL`. Istnieje również możliwość tymczasowego zainstalowania zestawu MIDletów, wybrania jednego z nich, uruchomienia go, a następnie automatycznego odinstalowania całego zestawu. Służą do tego opcje `-transient` oraz `-autotest`. Na przykład:

```
midp -transient http://www.midlethost.acme.com/suite.jad CalendarMIDlet
midp -transient http://www.midlethost.acme.com/suite.jad
midp -autotest http://www.midlethost.acme.com/suite.jad CalendarMIDlet
```

Opcja `-transient` przeprowadza pojedynczy cykl *zainstaluj/wykonaj/usuń*, natomiast `-autotest` powtarza go do czasu, aż nie zostanie on przerwany przez użytkownika. Przydaje się to przy automatycznym testowaniu MIDletów, zwłaszcza tych, które nie wymagają interakcji ze strony użytkownika. Jeżeli w linii poleceń nie wprowadzono nazwy MIDletu, zostanie wyświetlona lista wszystkich aplikacji dostępnych w danym zestawie, z której użytkownik będzie mógł wybrać tę, którą będzie chciał uruchomić.

Zarządzanie zestawami MIDletów

Mechanizm AMS (Oprogramowanie zarządzające aplikacjami), wbudowany w polecenie `midp`, może być obsługiwany bądź z linii poleceń, bądź za pomocą graficznego interfejsu użytkownika. Poniższe polecenie uruchamia emulator telefonu komórkowego i wyświetla menu, dzięki któremu użytkownik może poinformować AMS, że chce zainstalować zestaw MIDletów za pośrednictwem sieci.

```
midp
```

Informacje wymagane do pobrania i zainstalowania na stałe zestawu MIDletów mogą być też wprowadzone w linii poleceń, co pozwala na uniknięcie interakcji z graficzną postacią AMS:

```
midp -install http://www.midlethost.acme.com/suite.jad
midp -install -force http://www.midlethost.acme.com/suite.jad
```

Za pomocą opcji `-force` można wymusić reinstalację już zainstalowanego zestawu MIDletów, bez usuwania wcześniejszej kopii. Polecenie `midp` obsługuje możliwość wczytywania zestawów MIDletów z serwerów sieciowych za pośrednictwem mechanizmu OTA (patrz podrozdział „Dostarczanie over-the-air" w rozdziale 3.). Przesyłanie danych odbywa się w tym przypadku z wykorzystaniem protokołu HTTP.

Wprowadzając opcję `-list`, można uzyskać listę aktualnie zainstalowanych zestawów MIDletów:

```
midp -list
```

To polecenie wyświetla krótkie podsumowanie na temat każdego zainstalowanego zestawu MIDletów. Zwracane przez nie dane mogą wyglądać na przykład tak:

```
[1]
 Name: Chapter4
 Vendor: J2ME in a Nutshell
 Version: 1.0
 Storage name: #J2#M#E%0020in%0020a%0020#Nutshell_#Chapter4_
 Size: 23K
 Installed From: http://hostname/path/Chapter4.jad
 MIDlets:
 [lista MIDletów w zestawie]
```

Każdy zestaw MIDletów ma nadany własny numer (1 w powyższym przykładzie) oraz nazwę, pod którą zostanie on zachowany, a której format zależy od konkretnej implementacji. Polecenie `midp` tworzy ją według szablonu `nazwaDostawcy_nazwaZestawu_`, przy czym poprzedza wszystkie wielkie litery symbolem `#` i zamienia znaki niealfanumeryczne na odpowiadające im znaki zapisane w systemie Unicode, poprzedzając je znakiem procenta (`%`). Pozwala to przechowywać nazwy bez żadnych przekłamań nawet na tych urządzeniach, które obsługują jedynie 8-bitowe znaki, bądź nie rozróżniają wielkich i małych liter. Listę nazw, pod którymi zapisane są wszystkie zestawy MIDletów występujące na urządzeniu, można poznać wydając polecenie:

```
midp -storageNames
```

Po zainstalowaniu zestawu MIDletów uruchamiasz emulator, w którym zostanie wyświetlona ich lista. Możesz z niej wybrać ten MIDlet, który ma zostać uruchomiony. W tym celu wprowadź opcję `-run` wraz z numerem lub nazwą, pod którą przechowywany jest dany zestaw:

```
midp -run 1
midp -run #J2#M#E%0020in%0020a%0020#Nutshell_#Chapter4_
```

Na podobnej zasadzie, posługując się nazwą lub numerem zestawu, możesz go usunąć:

```
midp -remove 1
midp -remove #J2#M#E%0020in%0020a%0020#Nutshell_#Chapter4_
```

Opcje

Polecenie `midp` ma trzy opcjonalne argumenty:

`-classpath` *`ścieżka`*
: Określa lokalizację plików klas MIDletów. Ta opcja przydaje się wówczas, gdy chcesz przetestować lokalnie zainstalowane MIDlety, które nie zostały jeszcze spakowane do plików JAR. Poszczególne lokalizacje, które mogą być nazwami katalogów lub nazwami plików JAR, oddziela się od siebie separatorem specyficznym dla danej platformy. Na przykład w systemie Windows jest to średnik, a w Uniksie dwukropek.

`-help`
Wyświetla informacje na temat dostępnych opcji i sposobu ich użycia, po czym kończy działanie polecenia.

`-version`
Wyświetla obsługiwane wersje CLDC i MIDP oraz numer wersji pliku wykonywalnego, po czym kończy działanie polecenia.

Poza tymi argumentami można stosować wszelkie opcje dostępne dla KVM (patrz wcześniejszy podrozdział „kvm: Kilobyte Virtual Machine"), w tym opcje związane z debugowaniem, o ile tylko polecenie `midp` zostało skompilowane w odpowiedni sposób.

Patrz również

- *kvm*
- *emulator*

emulator: emulator z pakietu J2ME Wireless Toolkit

Dostępność

J2ME Wireless Toolkit

Składnia

```
emulator [opcje] [nazwaklasy]
```

Opis

Polecenie `emulator` zapewnia środowisko, w którym można wykonywać aplikacje i zarządzać nimi. Wchodzi ono w skład pakietu J2ME Wireless Toolkit. Pod względem funkcjonalności i opcji, które można wprowadzać z linii poleceń, bardzo przypomina polecenie `midp`, przy czym umożliwia ono stosowanie „skór" imitujących wygląd różnych urządzeń oraz plików konfiguracyjnych, co pozwala emulować różne urządzenia bez konieczności modyfikowania kodu. Choć można wywoływać emulator z linii poleceń, najczęściej dostęp do niego uzyskuje się pośrednio — poprzez interfejs `KToolBar` obecny w pakiecie Wireless Toolkit.

Opcje

Działanie polecenia `emulator` zależy od przekazanych do niego opcji. Wyróżniamy trzy tryby pracy:

- Wyświetlanie informacji po wprowadzeniu opcji `-help`, `-version` oraz `-Xquery`. W tym przypadku argument *`nazwaklasy`* nie jest wymagany, a polecenie kończy swą pracę zaraz po zwróceniu żądanych informacji.
- Uruchamianie MIDletu pochodzącego z lokalnego systemu plików bądź wczytanego z serwera sieciowego, ale bez instalowania go. W tym trybie pracy stosuje się opcję `-classpath` wraz z nazwą klasy lub opcję `-Xdescriptor`, której może, ale nie musi, towarzyszyć nazwa klasy.
- Korzystanie z dostępnego na emulatorze oprogramowania zarządzającego aplikacjami do instalowania, uruchamiania oraz kasowania zestawów MIDletów bądź wyświetlania ich listy. W tym trybie pracy stosuje się opcję `-Xjam`.

Poniżej znajduje się omówienie poszczególnych opcji `emulatora`.

`-classpath` *`ścieżka`*
: Określa lokalizację plików klas MIDletów. Ta opcja przydaje się wówczas, gdy chcesz przetestować lokalnie zainstalowane MIDlety, które nie zostały jeszcze spakowane do plików JAR. Poszczególne lokalizacje, które mogą być nazwami katalogów lub nazwami plików JAR, oddziela się od siebie separatorem specyficznym dla danej platformy. Na przykład w systemie Windows jest to średnik, a w Uniksie dwukropek.

`-cp` *`ścieżka`*
: Synonim `-classpath`.

`-help`
: Wyświetla dozwolone argumenty polecenia, po czym kończy jego działanie.

`-version`
: Wyświetla numer wersji pakietu J2ME Wireless Toolkit oraz wykorzystywanych przez niego implementacji CLDC i MIDP, po czym kończy wykonywanie polecenia.

`-Xdebug`
: Przygotowuje emulator na debugowanie w czasie rzeczywistym. Tę opcję trzeba stosować razem z `-Xrunjdwp`.

`-Xdevice:`*`nazwa`*
: Uruchamia emulator urządzenia o podanej nazwie. Poszczególne urządzenia różnią się między sobą ilością dostępnej pamięci, urządzeniami wejściowymi oraz ekranami, a dla każdego z nich stosowana jest inna „skóra" emulatora. Domyślnie rozpoznawane są następujące *`nazwy`*:

  `DefaultColorPhone`
  : Telefon komórkowy z kolorowym wyświetlaczem

  `DefaultGrayPhone`
  : Telefon komórkowy z wyświetlaczem pracującym w skali szarości

  `MinimumPhone`
  : Podstawowy, dwukolorowy telefon

  `Motorola_i85s`
  : Telefon komórkowy Motorola i85s

`PalmOS_Device`
: Pseudourządzenie pracujące pod kontrolą systemu PalmOS

`RIMJavaHandheld`
: Bezprzewodowy palmtop Research In Motion

`-Xdescriptor:`*`nazwaPliku`*
: Wczytuje zestaw MIDletów po zbadaniu zawartości wskazanego pliku JAD i pozwala użytkownikowi wybrać MIDlet, który ma zostać uruchomiony. Jeżeli zostanie wprowadzony opcjonalny argument `classname`, zakłada się, że określa on jeden z MIDletów w zestawie, który chcemy wywołać. Argument *`nazwaPliku`* może być adresem URL bądź ścieżką w lokalnym systemie plików.

`-Xheapsize:`*`rozmiar`*
: Ustawia rozmiar sterty (ang. *heap*), zastępując wartość domyślną dla danej implementacji. Parametr *`rozmiar`* może mieć wartość bezwzględną, wyrażoną w bajtach (na przykład `131072`), bądź zapisaną skrótowo, np. `512k` lub `2M`. Nie może ona być mniejsza niż `32K`, ani większa niż `64M`.

`-Xjam:`*`polecenie`*
: Uruchamia emulator i wykonuje za pośrednictwem AMS operację określaną przez argument *`polecenie`*. Dozwolone operacje wymieniono w następnym podrozdziale.

`-Xquery`
: Wyświetla właściwości wszystkich urządzeń, które emulator jest w stanie emulować, w tym ich opisy, oraz informacje na temat ich ekranów i urządzeń wejściowych. Jeżeli chcesz zobaczyć przykładowe zwracane wyniki, zajrzyj do jednego z kolejnych punktów pt. „Przykłady”.

`-Xrunjdwp:`*`opcje`*
: Jeżeli ten argument używany jest wraz z `-Xdebug`, określa on sposób, w jaki zdalny debugger będzie się łączył z maszyną wirtualną, a także niezbędny do tego adres. *`Opcje`* mogą mieć następujące wartości:

```
transport=<transport>,address=<adres>,server=<y/n>
```

  gdzie zmiennej `transport` trzeba przypisać wartość `dt_socket`, natomiast `adres` ma postać `host:port`. Argument `server` powinien mieć zawsze wartość `y`.

`-Xverbose:`*`opcje`*
: Określa, na ile szczegółowe mają być informacje zwracane przez debugger. Opcjom można przypisać wartość `all` lub listę wybranych, oddzielonych od siebie przecinkami wartości z poniższego zestawu:

```
allocation      bytecodes       class
classverbose    events          exceptions
frames          gc              gcverbose
methods         methodsverbose  monitors
networking      stackchunks     stackmaps
threading       verifier
```

Polecenia służące do zarządzania aplikacjami

Argument `-Xjam` pozwala kontrolować oprogramowanie do zarządzania aplikacjami wbudowane w emulator. Zaraz za nim umieszcza się dwukropek oraz jedno z poleceń wymienionych w tabeli 8.1. Argument `-Xjam` można też wprowadzić bez żadnych dodatkowych opcji. Spowoduje to uruchomienie emulatora i wyświetlenie graficznego interfejsu AMS, tak jak zostało to opisane w punkcie „Oprogramowanie do zarządzania aplikacjami w Wireless Toolkit" w rozdziale 3.

Tabela 8.1. Polecenia AMS emulatora wchodzącego w skład pakietu Wireless Toolkit

Polecenie	Opis
`force`	Gdy to polecenie używane jest wraz z poleceniem `install`, wymusza ono instalację zestawu MIDletów, nawet jeśli jest on już zainstalowany
`install=`*`URL_deskryptora`*	Instaluje zestaw MIDletów, których plik JAD znajduje się we wskazanej lokalizacji
`list`	Wyświetla informacje dotyczące zainstalowanych pakietów MIDletów, w tym numery i nazwy, pod którymi są one przechowywane. Format zapisu tych danych został już omówiony w podrozdziale „Zarządzanie zestawami MIDletów"
`remove=`*`nazwa pod jaką przechowywany jest zestaw MIDletów`*	Usuwa zestaw MIDletów o podanej nazwie
`remove=`*`numer_zestawu`*	Usuwa zestaw MIDletów o wskazanym numerze
`remove=all`	Usuwa wszystkie zainstalowane zestawy MIDletów
`run=`*`nazwa pod jaką przechowywany jest zestaw MIDletów`*	Wyświetla listę wszystkich MIDletów wchodzących w skład zestawu o podanej nazwie i pozwala wybrać użytkownikowi ten z nich, który ma zostać uruchomiony
`storageNames`	Wyświetla nazwy, pod którymi zainstalowane są wszystkie dostępne na urządzeniu zestawy MIDletów. Nazwy te zostały dokładnie omówione we wcześniejszym podrozdziale, zatytułowanym „Zarządzanie zestawami MIDletów"
`transient=`*`URL_deskryptora`*	Tymczasowo instaluje zestaw MIDletów, pozwalając użytkownikowi wybrać i uruchomić jeden z nich, po czym usuwa cały zestaw z urządzenia. Jeżeli zestaw jest już zainstalowany, instalacja jest pomijana, ale trzeba pamiętać, że na końcu zostanie on usunięty

Przykłady

```
emulator -cp dir1;dir2;dir3 ora.ch5.AttributesMIDlet
```

Wykonuje MIDlet `ora.ch5.AttributesMIDlet`, wczytując jego klasy z podanej ścieżki.

`emulator -Xdebug -Xrunjdwp:transport=dt-socket,address=2000,`
`server=y -cp dir1;dir2;dir3 ora.ch5.AttributesMIDlet`

Wykonuje MIDlet `ora.ch5.AttributesMIDlet`, wczytując jego klasy z podanej ścieżki i przygotowując maszynę wirtualną na sesję debuggera.

`emulator -Xdescriptor:http://servername/path/suite.jad`

Wczytuje zestaw MIDletów, którego plik JAD wskazano w poleceniu, i pozwala użytkownikowi wybrać MIDlet, który ma zostać uruchomiony.

`emulator -Xdescriptor:http://servername/`
`path/suite.jad ora.ch5.AttributesMIDlet`

Wczytuje zestaw MIDletów, którego plik JAD wskazano w poleceniu, i uruchamia ten MIDlet, którego plik klasy nazywa się `ora.ch5.AttributesMIDlet`.

`emulator -Xquery`

Wyświetla informacje o wszystkich urządzeniach obsługiwanych przez emulator. Dla każdego z nich zwracane są tego typu dane:

```
# Properties for device DefaultGrayPhone
DefaultGrayPhone.description: DefaultGrayPhone
DefaultGrayPhone.screen.width: 96
DefaultGrayPhone.screen.height: 128
DefaultGrayPhone.screen.isColor: false
DefaultGrayPhone.screen.isTouch: false
DefaultGrayPhone.screen.width: 96
DefaultGrayPhone.screen.bitDepth: 8
```

`emulator -Xjam:install=http://servername//path/suite.jad`

Instaluje poprzez sieć ten zestaw MIDletów, którego plik JAD wskazano w wywołaniu polecenia. Jeżeli jest on już zainstalowany, wywołanie polecenia kończy się błędem.

`emulator -Xjam:install=http://servername//path/`
`suite.jad -Xjam:force`

Instaluje dany zestaw MIDletów, wymuszając przy tym zastąpienie jego wszelkich istniejących kopii.

`emulator -Xjam:run=#J2#M#E%0020in%0020a%0020#Nutshell_#Chapter5_`

Wyświetla listę wszystkich MIDletów występujących w zestawie o wskazanej nazwie i umożliwia użytkownikowi wybranie tego z nich, który ma zostać wykonany.

`emulator -Xjam:storageNames`

Wyświetla nazwy, pod którymi przechowywane są wszystkie zainstalowane zestawy MIDletów.

`emulator -Xjam:remove=1`

Usuwa zainstalowany zestaw MIDletów o numerze 1.

Patrz również

- *midp*

preverify: preweryfikator klas KVM

Dostępność

Wzorcowa implementacja CLDC, wzorcowa implementacja MIDP, pakiet Wireless Toolkit

Składnia

```
preverify [opcje] nazwyklas | nazwykatalogów | nazwyplikówJAR
```

Opis

Jest to preweryfikator klas, badający klasy, które mają zostać wczytane przez maszynę wirtualną zgodną z CLDC, taką jak na przykład KVM. Wszystkie klasy programu trzeba przed uruchomieniem zweryfikować, aby upewnić się, że mają one prawidłową postać i nie będą próbowały omijać reguł języka Java, co mogłoby doprowadzić do złamania systemu bezpieczeństwa.

Polecenie `preverify` przetwarza określony zbiór plików wejściowych, w których zapisane są klasy, i zapisuje wyniki we wskazanej lokalizacji, która musi być inna niż lokalizacja źródłowa. Chcąc wskazać pliki klas, które mają zostać przetworzone, można podać:

- Zbiór nazw klas, w którym położenie każdej klasy określa się względem ścieżki przekazanej w argumencie `-classpath` lub przechowywanej w zmiennej środowiskowej `CLASSPATH`
- Nazwę pliku JAR bądź ZIP, zawierającego pliki klas
- Nazwę katalogu, który ma być rekurencyjnie przeszukiwany w poszukiwaniu plików klas, a także plików JAR oraz ZIP

Wyniki pracy preweryfikatora zapisywane są w katalogu wskazanym po argumencie *`-d`*. Jeżeli taki argument nie zostanie wprowadzony, pliki trafią do katalogu o nazwie *output*. Zawartość plików JAR i ZIP jest natomiast zapisywana do plików JAR/ZIP o takich samych nazwach co pliki źródłowe, przy czym są one umieszczane w katalogu output.

Opcje

`@nazwapliku`
: Określa nazwę pliku, z którego wczytywane są argumenty, które zostaną wprowadzone w linii poleceń. Plik ten może zawierać tylko jeden wiersz tekstu, w którym będą zapisane dopuszczalne przez program argumenty. Zostaną one przetworzone po odczytaniu pliku. Jeżeli w pliku mają znaleźć się nazwy innych plików i katalogów, należy umieścić je w cudzysłowach. Mogą one przy tym zawierać spacje.

`-classpath` *`ścieżka`*
Określa położenie plików klas. Mogą to być nazwy katalogów lub plików JAR, oddzielone od siebie separatorem specyficznym dla danej platformy (na przykład w systemie Windows jest to średnik, a w Uniksie dwukropek). Opcja `-classpath` powinna określać położenie bibliotek podstawowych, jak również klas, które mają być poddane preweryfikacji, chyba że informacje te są już dostępne w zmiennej środowiskowej `CLASSPATH`.

`-cldc`
Jeżeli zostanie wprowadzony ten argument, preweryfikator klas będzie sprawdzał, czy nie wykorzystują one tych mechanizmów maszyny wirtualnej, które nie zostały ujęte w specyfikacji CLDC. Oznacza to, że klasy nie mogą korzystać z metod natywnych i operacji na liczbach zmiennoprzecinkowych, ani też finalizować obiektów. Jest to odpowiednik jednoczesnego wprowadzenia argumentów `-nofinalize`, `-nofp` oraz `-nonative`.

`-d` *`nazwakataloguwyjsciowego`*
Określa nazwę katalogu, do którego mają zostać zapisane klasy już po weryfikacji. Domyślnie, przy braku tego argumentu, przyjmuje się, że katalog nazywa się *output*. Jeżeli *preweryfikator* będzie przetwarzał jakieś pliki JAR lub ZIP, efekty jego pracy również powędrują do tego katalogu.

`-nofinalize`
Wprowadzenie tego argumentu sprawia, że preweryfikator będzie sprawdzał, czy klasy nie próbują finalizować obiektów. Jeżeli natomiast zostanie on pominięty, a użytkownik nie wprowadzi opcji `-cldc`, próba finalizacji obiektu spowoduje w trakcie wykonywania programu wygenerowanie błędu.

`-nofp`
Wprowadzenie tego argumentu sprawia, że preweryfikator będzie sprawdzał, czy w klasach nie występują odwołania do liczb zmiennoprzecinkowych. Jeżeli natomiast zostanie on pominięty, a użytkownik nie wprowadzi opcji `-cldc`, próba finalizacji obiektu spowoduje w trakcie wykonywania programu wygenerowanie błędu.

`-nonative`
Wprowadzenie tego argumentu sprawia, że preweryfikator będzie sprawdzał, czy w klasach nie zadeklarowano metod natywnych. Jeżeli natomiast zostanie on pominięty, a użytkownik nie wprowadzi opcji `-cldc`, próba finalizacji obiektuspowoduje w trakcie wykonywania programu wygenerowanie błędu. Pamiętaj jednak, że aplikacje napisane specjalnie z myślą o pewnych zmodyfikowanych wersjach KVM mogą korzystać z metod natywnych. Zostało to opisane w rozdziale 2., w części zatytułowanej „Współpraca z kodem natywnym". W tym przypadku opisywany argument nie znajduje zastosowania.

`-verbose`
Sprawia, że informacje kontrolne kierowane są do standardowego strumienia błędów.

`-verify-verbose`
Sprawia, że na standardowy strumień błędów kierowane są szczegółowe informacje kontrolne. Wprowadzenie tej opcji może zaowocować zwracaniem naprawdę dużych ilości informacji.

Przykłady

Chcąc poddać procesowi preweryfikacji pojedynczą klasę o nazwie `ora.ch2.KVMProperties`, znajdującą się względem bieżącego katalogu w lokalizacji *tmpclasses\ora\ch2\KVMProperties.class* i zapisać jej zweryfikowaną wersję do pliku o nazwie *output\ora\ch2\KVMProperties.class*, trzeba wpisać poniższe polecenie. Podstawowe biblioteki klas znajdują się w tym przykładzie w katalogu *c:\j2me\j2me_cldc\bin\common\api\lclasses*:

```
preverify -classpath c:\j2me\j2me_cldc\bin\common\api\
↳lclasses;tmpclasses ora.ch2.KVMProperties
```

Chcąc poddać weryfikacji wszystkie klasy zapisane w *tmpclasses\native.jar*, zapisać je w pliku *native.jar* w katalogu bieżącym i upewnić się, że nie występuje w nich finalizowanie obiektów ani operacje na liczbach zmiennoprzecinkowych, należy napisać:

```
preverify -classpath c:\j2me\j2me_cldc\bin\common\api\lclasses -nofp -
↳nofinalize -d . tmpclasses\native.jar
```

Aby zweryfikować wszystkie klasy zapisane w katalogu *tmpclasses* i jego podkatalogach oraz skierować klasy wynikowe do bieżącego katalogu, w którym ma zostać utworzona taka sama struktura katalogów, należy napisać:

```
preverify -classpath c:\j2me\j2me_cldc\bin\common\api\lclasses -d . tmpclasses
```

Patrz również

- *kvm*
- Dokument „KVM Porting Guide", który znajdziesz w archiwum ze wzorcową implementacją CLDC

MakeMIDPApp: narzędzie konwertujące pliki JAD na PRC

Dostępność

MIDP for PalmOS

Składnia

```
java -cp Converter.jar com.sun.midp.palm.database.MakeMIDPApp
[opcje] plikjar
```

Opis

Polecenie `MakeMIDPApp` zamienia zestaw MIDletów, składający się z pliku JAD i JAR, na postać nadającą się do zainstalowania na urządzeniach z systemem PalmOS. *MakeMIDPApp*

jest narzędziem napisanym w języku Java, które znajdziesz w pliku *%INSTALL_DIR%\Converter\Converter.jar*, gdzie *%INSTALL_DIR%* jest katalogiem, w którym zainstalowałeś oprogramowanie MIDP for PalmOS.

Opcje

`-help`
Wyświetla składnię polecenia i listę rozpoznawanych przez nie opcji.

`-v`
Wyświetla szczegółowe informacje. Jeżeli wprowadzisz tę opcję dwukrotnie (np. `-v -v`), polecenie będzie zwracało jeszcze więcej informacji.

`-jad` *`plik`*
Wskazuje plik JAD zestawu MIDletów. Ten argument jest opcjonalny, ale jeśli go nie wprowadzisz, to wszelkie właściwości aplikacji zapisane w pliku JAD, które nie zostały też zapisane w manifeście pliku JAR, nie będą dostępne dla aplikacji. Omówienie właściwości aplikacji znajdziesz w rozdziale 3., w części zatytułowanej „Prosty MIDlet".

`-name` *`nazwa`*
Nadaje zestawowi MIDletów nazwę, która będzie wyświetlana na urządzeniu docelowym przez oprogramowanie służące do wywoływania aplikacji. Nazwa może zawierać spacje, pod warunkiem, że zostanie ona ujęta w cudzysłów. Nie ma gwarancji, że nazwy o długości przekraczającej 9 znaków będą wyświetlane w całości. Jeżeli ten argument zostanie pominięty, wykorzystywana jest nazwa zestawu MIDletów pochodząca z pliku manifestu lub z pliku JAD (o ile zostanie on wskazany).

`-longname` *`nazwa`*
Długa nazwa MIDletu, składająca się z maksymalnie 31 znaków, która będzie wykorzystywana tam, gdzie będzie wystarczająco dużo miejsca na ekranie. Na przykład na liście MIDletów, wyświetlanej w oknie dialogowym *Developer preferences* (dostępnym w trakcie wykonywania MIDletu w menu *Options*). Jeżeli w nazwie występują spacje, powinno się ją ująć w cudzysłów.

`-icon` *`plik`*
Określa ikonę, która będzie reprezentować zestaw MIDletów na liście możliwych do odpalenia aplikacji (gdy lista będzie wyświetlana w trybie ikon). Jeżeli ten argument zostanie pominięty, będzie stosowana ikona domyślna. Podana ikona może być zapisana w jednym z trzech formatów:

BMP
Mapa bitowa systemu Windows

PBM
Przenośna mapa bitowa (Portable bitmap format)

BIN
Mapa bitowa w formacie PalmOS

Nie są obsługiwane skompresowane oraz kolorowe pliki BMP. Aby uzyskać najlepszy efekt, obrazek powinien być kwadratem o boku 32 pikseli, w którym przynajmniej

5 pikseli od zewnątrz po lewej i prawej stronie oraz 10 pikseli w pionie powinno mieć kolor biały. Jeżeli rozmiar obrazka jest nieprawidłowy, zostanie od przeskalowany, co może wiązać się z pogorszeniem jakości.

`-smallicon` *`plik`*
Określa ikonę, która będzie reprezentować zestaw MIDletów na liście możliwych do odpalenia aplikacji, gdy lista będzie wyświetlana w trybie opisowym. Jeżeli ten argument zostanie pominięty, będzie stosowana ikona domyślna. Wskazany obrazek powinien mieć 15 pikseli szerokości i 9 pikseli wysokości.

`-creator` *`id`*
Nadaje zestawowi MIDletów czteroznakowy numer ID identyfikujący jego twórcę. Numery te są wymagane przez system PalmOS. Jeżeli chcesz przypisać taki identyfikator jakiemuś komercyjnemu programowi, powinieneś go zarejestrować na stronie *http://www.palm.com/devzone/*. ID twórcy programu jest przypisywany do wykorzystywanego przez zestaw MIDletów obszaru pamięci nieulotnej. Pojawia się on także na liście wszystkich dostępnych zestawów MIDletów, widocznej w oknie dialogowym *developer preferences*. Jeżeli nie wprowadzisz numeru ID, zostanie on przydzielony automatycznie. W tym przypadku musisz wprowadzić argument `-type Data`.

`-type` *`typ`*
Określa rodzaj pliku wynikowego, jaki ma zostać utworzony. W argumencie *`typ`* są rozróżniane małe i wielkie litery. Może on przyjmować wartość `appl` lub `Data`. Pierwsza z nich jest wartością domyślną. Jeżeli nie wprowadziłeś identyfikatora twórcy programu za pośrednictwem argumentu `-creator`, musisz wprowadzić wartość `Data`. Zestawy MIDletów mające typ `Data` nie mogą być przesyłane przez port podczerwieni między urządzeniami z systemem PalmOS.

`-outfile` *`plik`*
Nazwa pliku, w którym ma zostać zapisany przekonwertowany zestaw MIDletów. Przyjęło się, że takie pliki mają rozszerzenie *.prc*.

`-o` *`plik`*
Synonim `-outfile`.

Przykłady

```
java -cp Converter.jar com.sun.midp.palm.database.MakeMIDPApp
-icon myIcon.bmp -smallicon myListIcon.bmp -jad Chapter3.jad
-o Chapter3.prc -type Data Chapter3.jar
```

Zamienia zestaw MIDletów zapisany w pliku *Chapter3.jar* i zbiór jego właściwości opisanych w pliku *Chapter3.jad* na postać nadającą się do wgrania na urządzenie z systemem PalmOS. Plik wynikowy będzie nosił nazwę *Chapter3.prc*. Ikony, które będą wyświetlane na urządzeniu w oprogramowaniu obsługującym odpalanie aplikacji, to *myIcon.bmp* (wykorzystywana, gdy lista aplikacji będzie wyświetlana w postaci ikon) oraz *myListIcon.bmp* (dla trybu opisowego). Ponieważ nie wprowadzamy ID twórcy, typ został określony na `Data`.

```
java -cp Converter.jar com.sun.midp.palm.database.MakeMIDPApp -jad
Chapter3.jad -o Chapter3.prc -creator ORA3 -name „Ch 3” -longname
„J2ME Chapter 3” Chapter3.jar
```

Zamienia zestaw MIDletów zapisany w pliku *Chapter3.jar* i zbiór jego właściwości opisanych w pliku *Chapter3.jad* na postać nadającą się do wgrania na urządzenie z systemem PalmOS. Plik wynikowy będzie nosił nazwę *Chapter3.prc*. Na ekranie, z którego można uruchamiać aplikacje, nasz zestaw MIDletów będzie reprezentowany przez ikony domyślne, podpisane `Ch 3`. Tam gdzie istnieje możliwość wyświetlenia dłuższej nazwy, pojawi się `J2ME Chapter 3`. Z zestawem MIDletów skojarzono ID twórcy o wartości ORA3, w związku z czym nie ma potrzeby wprowadzania argumentu `-type`.

Miej świadomość, że nie wszystkie kombinacje typu i numeru ID twórcy zaowocują utworzeniem programu, który będzie można uruchomić na dowolnym urządzeniu z systemem PalmOS. W zależności od zastosowanej kombinacji argumentów, można uzyskać różne rezultaty: (xxxx oznacza czterocyfrowy identyfikator)

`-creator XXXX -type appl`

Zawsze powoduje utworzenie dającego się uruchomić zestawu MIDletów. MIDlety będzie można przesyłać przez port podczerwieni na inne urządzenia.

`-creator XXXX -type Data`

Zestaw MIDletów będzie można zainstalować, ale nie da się go uruchomić ani przesłać przez port podczerwieni.

`-type Data`

Zestaw MIDletów będzie można uruchomić, ale nie da się go przesyłać przez port podczerwieni.

MEKeyTool: narzędzie do zarządzania certyfikatami z kluczami publicznymi

Dostępność

Wzorcowa implementacja MIDP, pakiet Wireless Toolkit

Składnia

```
java -jar MEKeyTool.jar -help
```

```
java -jar MEKeyTool.jar -list [-MEkeystore nazwapliku]
```

```
java -jar MEKeyTool.jar -import [-MEkeystore nazwapliku]
[-keystore nazwapliku] [-storepass hasło] -alias keyAlias
[-domain domena]
```

```
java -jar MEKeyTool.jar -delete [-MEKeystore nazwapliku]
-owner nazwaWłaściciela
```

Opis

MEKeyTool jest narzędziem służącym do zarządzania pamięcią, w której przechowywane są certyfikaty z kluczami publicznymi. Dzięki nim możliwa jest obsługa bezpiecznych połączeń sieciowych (HTTPS), zaimplementowana we wzorcowej implementacji MIDP oraz w pakiecie J2ME Wireless Toolkit. Narzędzie *MEKeyTool* dostarczane jest w postaci pliku JAR o nazwie *MEKeyTool.jar*, który można znaleźć w katalogu *%INSTALL_DIR%\bin*, gdzie `%INSTALL_DIR%` oznacza katalog, w którym zainstalowano pakiet J2ME Wireless Toolkit. Jest ono również udostępniane wraz ze wzorcową implementacją MIDP w postaci kodu źródłowego.

Gdy korzystasz z *MEKeyTool* w połączeniu z J2ME Wireless Toolkit, narzędzie to opiekuje się pamięcią przechowującą certyfikaty (nazywaną tu *pamięcią ME*), która domyślnie ma postać pliku dyskowego o nazwie *%INSTALL_DIR%\appdb_main.ks*. Wszystkie operacje odwołują się niejawnie właśnie do tego pliku, chyba że korzystając z opcji `MEkeystore`, wskazałeś jakiś inny. *MEKeyTool* potrafi wyświetlić zawartość pamięci certyfikatów, zaimportować certyfikaty z J2SE bądź usunąć je w razie potrzeby. Aby prawidłowo posługiwać się tym narzędziem, musisz rozumieć zasadę działania pamięci przechowującej certyfikaty w J2SE i znać polecenie `keytool`, za pomocą którego się nią zarządza. Zostały one opisane w książce „Java in a Nutshell" Davida Flanagana (O'Reilly).

Opcje

`-MEKeystore` *`nazwa pliku`*
: Określa lokalizację pamięci ME. Domyślnie ma ona postać pliku *appdb_main.ks* zapisanego w katalogu, do którego zainstalowano pakiet Wireless Toolkit.

`-keystore` *`nazwapliku`*
: Lokalizacja pamięci na certyfikaty, wykorzystywanej przez środowisko J2SE. J2SE jest bowiem dostarczane wraz z pewną ilością predefiniowanych certyfikatów, należących do znanych firm certyfikacyjnych, które można przenieść do pamięci ME. Pamięć wykorzystywana przez J2SE znajduje się w lokalizacji *%JAVA_HOME%\jre\lib\security\cacerts*.

`-storepass` *`hasło`*
: Hasło, za pomocą którego można się dostać do pamięci na certyfikaty J2SE. Domyślnie brzmi ono `changeit`, ale może zostać zmienione za pomocą polecenia J2SE o nazwie `keytool`.

`-alias` *`nazwaAliasu`*
: Wskazuje certyfikat, który ma zostać wyeksportowany z pamięci certyfikatów J2SE. Istnieje możliwość zobaczenia listy wszystkich certyfikatów zapisanych w pamięci, wraz z ich aliasami. Trzeba w tym celu wprowadzić polecenie J2SE o nazwie `keytool` z opcją `-list`:

```
keytool -list -keystore C:\jdk1.3.1\jre\lib\security\lib\cacerts
-storepass changeit
```

Typowe informacje zwracane przez to polecenie wyglądają tak:

```
Certificate fingerprint (MD5):
↳18:87:5C:CB:F8:20:5D:24:4A:BF:19:C7:13:0E:FD:B4
verisignserverca, Mon Jun 29 18:07:34 BST 1998, trustedCertEntry,
```

Aby zaimportować ten certyfikat do pamięci ME, musiałbyś posłużyć się aliasem `verisignserverca`.

`-owner` *`nazwaWłaściciela`*

Określa nazwę właściciela klucza, który ma zostać skasowany z pamięci ME. Jak widać na poniższym przykładzie, nazwy właścicieli są dość nieporęczne:

```
Key 1
 Owner: OU=Class 2 Public Primary Certification Authority;O=VeriSign,
 ↳Inc.;C=US
 Valid from Mon Jan 29 00:00:00 GMT 1996 to Wed Jan 07 23:59:59 GMT 2004
 Domain: untrusted
```

W tym przypadku nazwa właściciela to łańcuch *OU=Class 2 Public Primary Certification Authority;O=VeriSign, Inc.;C=US*.

`-domain` *`nazwaDomeny`*

Dzięki tej opcji możesz skojarzyć z kluczem importowanym do pamięci ME wybraną domenę bezpieczeństwa. Jeżeli chcesz po prostu zainstalować certyfikaty, dzięki którym będziesz mógł korzystać z obsługi HTTPS oferowanej przez wzorcową implementację MIDP, nie musisz korzystać z tej opcji.

Przykłady

Narzędzie *MEKeyTool* korzysta z domyślnej pamięci certyfikatów, mającej postać pliku wskazywanego przez względną ścieżkę dostępu *appdb_main.ks*. Chcąc uniknąć konieczności wprowadzania po opcji `-MEkeystore` bezwzględnych ścieżek dostępu, zazwyczaj najwygodniej jest przed wywołaniem narzędzia *MEKeyTool* spowodować, że katalog, w którym zainstalowano pakiet Wireless Toolkit, stanie się katalogiem bieżącym. W przykładach podanych w tym podrozdziale założono, że tak właśnie zrobiłeś.

Aby zaimportować do głównej pamięci ME certyfikat z pamięci certyfikatów J2SE o nazwie (aliasie) `versignserverca`, zapisany w pliku *c:\jdk1.3.1\jre\lib\security\cacerts*, należy wprowadzić następujące polecenie:

```
set JCE=c:\jdk1.3.1\jre\lib\security\cacerts
java -jar bin\MEKeyTool.jar -import -keystore %JCE% -storepass changeit
↳verisignserverca
```

Aby wyświetlić całą zawartość domyślnej pamięci ME, wpisz:

```
java -jar bin\MEKeyTool.jar -list
```

Wprowadzenie powyższego polecenia spowoduje zwrócenie tego typu informacji:

```
Key 1
 Owner: OU=Secure Server Certification Authority;O=RSA Data Security, Inc.;C=US
 Valid from Wed Nov 09 00:00:00 GMT 1994 to Thu Jan 07 23:59:59 GMT 2010
 Domain: untrusted
```

```
Key 2
 Owner: OU=Class 3 Public Primary Certification Authority;O=VeriSign, Inc.;C=US
 Valid from Mon Jan 29 00:00:00 GMT 1996 to Wed Jan 07 23:59:59 GMT 2004
 Domain: untrusted
```

Aby usunąć z domyślnej pamięci certyfikatów drugi klucz pokazany powyżej, wydaj następujące polecenie:

```
java -jar bin\MEKeyTool.jar -delete -owner „OU=Class 3 Public Primary
↳Certification Authority;O=VeriSign, Inc.;C=US”
```

Patrz również

- Opis polecenia `keytool` w książce „Java in a Nutshell” wydawnictwa O’Reilly.