ARCHITETTURA J2ME
E LIBRERIA MIDP

By Fabrizio Russo
27/11/2002

www.frusso.it/j2me

J2ME è una architettura che si rivolge ad una moltitudine di apparecchi anche profondamenti diversi tra di loro (Internet ScreenPhone, cerca persone, telefonini, PDA, elettrodomestici, ...) per questo motivo SUN ha definito una architettura ed una serire di librerie di classi per affrontare in modo univico la programmazione di questi dispositivi.

Per fare ciò SUN ha introdotto i concetti di configurazione e di profili

Una configurazione definisce una piattaforma Java per un'ampia gamma di prodotti. La configurazione è strettamente legata alla JVM e definisce le funzionalità del linguaggio Java e le librerie principali che la JVM mette a disposizione. Una configurazione definisce i requisiti di memoria, dello schermo, la connettività in rete che un dispositivo (per essere dichiarato conforme ad una data configurazione) deve possedere. Attualmente esistono due tipologie di configurazione: CDC (Connected Device Configuration) e CLDC (Connected, Limited Device Configuration)

Requisiti minimi	CDC	CLDC
Memoria per l'esecuzione di Java	>= 512 K	>= 128 K
Memoria per l'allocazione dinamica	>= 256 K	>= 32 K
Connettività di rete	persistente ed a banda larga	limitata ed ad accesso intermittente
Interfaccia utente		limitata
Alimentazione		ridotta (batterie)

Decidere se un dispositivo appartiene ad una configurazione piuttosto che ad un altra non è banale. La tecnologia continua ad evolvere e, man mano, che si procede le due configurazioni tenderanno a fondersi (pensate ad un PC di fascia alta comprato nel 2000, oggi sarebbe già obsoleto).

Esempi di CLDC sono sicuramente (ad oggi) un cellulare, un cerca persone o un palmare.

La classificazione dei dispositivi a seconda del dispositivo non è sufficiente. Per rispondere alle grandi differenze di funzionalità e per introdurre una maggiore flessibilità rispetto ai cambiamenti tecnologici, SUN ha introdotto il concetto di Profilo nella piattaforma J2ME.
Un profilo è una estensione della configurazione, fornisce le librerie per permettere ai programmatori di scrivere applicazioni per un particolare tipo di dispositivo.

Il più famoso profilo (quello che andremo a trattare) è il MIDP (Dispositivo Mobile di Informazione) che definisce le API per i componenti dell'interfaccia utente, per gli input, la gestione degli eventi, la memoria persistente (RMS), le funzioni di rete e i timer, il tutto prendendo in considerazione le limitazioni dello schermo e la memoria degli apparecchi mobili.

Semplificando possiamo immaginare la configurazione come una astrazione dell' hardware, mentre il profilo l'insieme delle librerie (API) necessarie per poter sviluppare applicazioni. Dal punto di vista architetturale la struttura è la seguente:

A livello più basso si trova il sistema operativo del dispositivo (nel caso dei Nokia il sistema operativo è Simbian OS)

Il secondo livello è rappresentato dalla KVM. La KVM è l'implementazione di riferimento della SUN per le VM dedicate a dispositivi wireless. (vedremo dopo le differenze con le J2SE)

Il terzo livello è rappresentato dalla configurazione (CLDC)

L'ultimo livello (quello più alto) si trova il profilo MIDP cioè le librerie necessarie per sviluppare applicazione.

Un'attenzione particolare riveste sia il linguaggio che la Java Virtual Machine dato che entrambi differiscono da quelli della J2SE (la versione standard di Java).

Per quanto riguarda il linguaggio Java le differenze salienti sono:

Calcoli in virgola mobile. I calcoli in virgola mobile fanno un uso intenso del processore e data l'assenza di un co-processore matematico e le limitate risorse di un processore presente su un dispositivo wireless si è deciso (nell'implmentazione del CLDC) di non supportare la virgola mobile. Questo si traduce nell'assenza (nel linguaggio Java per J2ME) dei tipi primitivi `float` **e** `double`**.**

Finalizzazione. In una classe J2SE si può dichiarare un metodo con il nome `finalize()` **che venga chiamato dal garbage collector prima di rimuovere l'oggetto. Il CLDC non supporta questo metodo dato che l' overload necessario (in termini di tempo) per la deallocazione delle risorse sarebbe troppo onerosa per il processore.**

Gestione delle eccezioni. La JVM supporta un numero limitato di eccezioni. Il motivo principale è da ricercare nel fatto che la gestione delle eccezioni in J2SE è molto complessa e tutti questi compiti sono pesanti da far svolgere ad un processore limitato, inoltre i sistemi embedded forniscono loro stessi una gestione degli errori interna, quindi, al presentarsi di un errore non si avrebbe il tempo di "catturare" l'eccezione in quanto il sistema ha già reagito per noi.

Oltre al linguaggio, anche la VM sono differenti. Una JVM per J2ME ha le seguenti limitazioni:

Gruppi di thread. Per questa implementazione JVM i thread sono trattati ordinatamente, oggetto per oggetto. La JVM non supporta la classe ThreadGroup quindi non si possono lanciare (o fermare) più thread in un colpo solo ma devono essere startati uno alla volta.

Finalizzazione. Non supportando il linguaggio Java di J2ME l'uso del metodo `finalize()` **neanche l'implementazione della JVM supportera la finalizzazione.**

Calcoli a virgola mobile. Dato che il linguaggio Java non supporta i dati in virgola mobile le implementazioni Java per J2ME mancano di tale supporto.

Interfaccia Nativa Java. Per ridurre il livello di danneggiamento di informazioni a livello del sistema, sono state eliminate le API per l'invocazione di metodi nativi, quindi non è possibile chimare funzioni native del sistema operativo ospite per effettuare operazioni (come accedere alla rubrica)

Caricatore di classi personalizzate. CLDC richiede che la JVM implementi un class-loader. Il caricatore di classi subisce severi controlli e non può essere sostituito, ignorato o modificato.

Reflection. In J2SE si possono usare le classi Reflection per ottenere informazioni sulla JVM in esecuzione. Queste API (causa consumo di risorse) non sono disponibili in CLDC

Inoltre un altro aspetto molto importante è la sicurezza, infatti ogni dispositivo che esegue un'applicazione Java necessita di una protezione da un eventuale codice "maligno" che possa accedere alle informazioni o alle risorse del sistema.
A livello delle applicazioni i programmi scritti usando J2SE possono caricare un gestore della sicurezza. Il gestore garantisce o vieta l'accesso attraverso una serie di chiamate di metodo, verificando i permessi

appropriati.
Nonostante questo modello sia sufficiente per J2SE, i requisiti di leggerezza di CLDC impediscono il loro utilizzo.

La configurazione CLDC definisce un modello conosciuto come "*sandbox*" che vieta l'accesso a tutto quello che è fuori dalla scatola. I limiti della send-box sono definiti tramite la coppia Configurazione/Profilo.

Un ultimo aspetto (ma molto importante) dell'architettura è quello della verifica dell'integrità dei file di classe.

Prima di essere eseguito un codice viene processato dalla J2SE per verificare che tutte le operazioni che il codice deve eseguire siano lecite. Questa operazione di verifica richiede da sola un minimo di 50 kbyte, per non parlare della memoria e del tempo di calcolo. Inutile dire che "date le risorse limitate del dispositovo" queste operazioni non possono essere eseguite a run-time.

Per evitare di eseguirle a run-time la J2ME introduce due nuovi concetti. La pre-verifica e la verifica.

1 - PreVerifica: Durante il processo di programmazione, o prima di caricare una classe su di un apparecchio, si esegue un programma per l'inserimento di attributi addizionali nel file di classe. Queste informazioni riducono l'ammontare di tempo e di memoria necessaria alla JVM per seguire il punto 2

2 – Verifica: Una volta che il dispositivo carica una classe pre-verificata, il verificatore interno dell'apparecchio ne percorre tutte le istruzioni. In un qualsiasi momento di questa fase il caricatore può rimandare un errore e scartare il file di classe.

Con questo sistema il dispositivo è in grado di rifiutare del codice potenzialmente "difettoso". Infatti se non ci fosse un tale processo, nessuno bi impedirebbe di utilizzare, per esempio, calcoli in virgola mobile nelle vostre applicazioni. Successivamente, fatto il deploy sul terminale, quando si procede all'esecuzione si incorre ad errori di sistema con eventuale perdita di dati (e di tempo).

Fatta questa breve panoramica sull'architettura torniamo pure a quello che a noi programmatori interessa di più (cioè la libreria MIDP).

Un dispositivo MIDP deve garantire un set di requisiti minimi sia hardware che software. In particolare un dispositivo che aderisca a questo standard deve avere:

- **Uno schermo almeno 96x54 pixel**
- **Dispositivi input per utente (tastiera, penna, ...)**
- **128 K per eseguire i componenti MIDP**
- **8 K per memorizzare dati persistenti (attraverso la libreria RMS)**
- **32 K per eseguire Java**
- **Connettività di rete wireless**

La libreria MIDlet è formata in totale da sette package

`java.io`	**Classi che fornisce sistemi di input ed output attraverso stream**
`java.lang`	**Classi di sistema derivate dalla J2SE**
`java.util`	**Classi di utilità derivate da J2SE**
`javax.micoredition.io`	**Supporto per le connessioni wireless**
`javax.microedition.lcdui`	**Supporto per l'interfaccia utente**
`javax.microedition.midlet`	**Classi di base per le MIDlet**
`javax.microedition.rms`	**Supporto per il sistema di persistenza**

I primi tre package rappresentano la compatibilità con il J2SE, mentre gli atri quattro package rappresentano la libreria MIDP e tutte le classi necessarie per sviluppare applicazioni wireless.

Il più importante tra tutti e sette i package è sicuramente `javax.microedition.midlet` **che contiene una classe** `javax.microedition.midlet.MIDlet` **ed una eccezione** `javax.microedition.midlet.MIDletStateChangeException`.

La classe principale di un'applicazione è la classe `javax.microedition.midlet.MIDlet`.

Una MIDlet è una applicazine definita nel profilo MIDP. Ogni applicazione deve estendere questa classe per permettere al software di controllo delle applicazioni (implementato a livello di CLDC) di recuperare le proprietà definite nell'applicazion descriptor e di notificare le richieste di stato.
I metodi di questa classe consentono al software di gestione delle applicazioni di creare, avviare, mettere in pausa e distruggere una MIDlet.

Una MIDlet (in generale) è un insieme di classi disegnate per essere eseguire e controllate dal sistema MIDP attraverso il passaggio tra diversi stati

Gli stati consentono al software di gestione di monitorare una MIDlet e di invocare i metodi opportuni ogni volta che questa cambia stato. Supponiamo infatti che mentre stato utilizzando la vostra applicazione, il telefono riceve una telefonata. Il sistema di gestione (CLDC) riceve l'evento e (cambianto stato) chiama automaticamente il metodo della MIDlet che la mette in pausa. Al termine della chiamata, vi chiederà se continuare o meno con l'utilizzo dell'applicazione e quindi richiamerà per voi il metodo per riavviare la MIDlet.

Una volta terminato lo sviluppo di un'applicazione, per poterla distribuire è necessario creare un file che contenga tutte le classi e le risorse. Il file che si andrà a creare è un file JAR. Questo file JAR oltre alle classi Java ed alle risorse ha anche un file molto importante. Questo file si chiama `manifest.mf` **e si trova nella cartella** `META-INF`**. In questo file si trovano tutta una serie di attibuti molto importanti per la definizione della MIDlet.**

Nome Attributo	Scopo	Richiesto
MIDlet-Name	Nome della MIDlet	Si
MIDlet-Version	Numero di versione del MIDlet	Si
MIDlet-Vendor	Autore della MIDlet	Si
MIDlet-	Riferimento ad un MIDlet specifico all'interno della Suite	Si
MicroEdition-Profile	Quale profilo J2ME viene richiesto	Si (MIDP-1.0)
MicroEdition-Configuration	Quale configurazione viene richiesta	Si (CLDC-1.0)
MIDlet-Icon	Icona usata dal gestore dell'applicazione	No
MIDlet-Info-URL	URL per le info supplementari	No

Alcune di queste informazioni non sono obbligatorie, mentre altre lo sono e doveno rispettare alcuni vincoli (che vedremo successivamente).

Oltre al file JAR la suite MIDlet deve essere corredato anche da un altro file particolare, chiamato application descriptor e che serve per fornire informazioni sulle MIDlet presenti all'interno del file JAR. Questo file normalmente ha lo stesso nome del file JAR ma ha l'estensione JAD.

Il file JAD fornisce informazioni al gestore dell'applicazione sul contenuto di un JAR e fornisce un mezzo di passaggio dei parametri a uno o più MIDlet senza dover cambiare il file JAR. Così come il file `manifest.mf` **anche il file JAD definisce alcuni attributi, alcuni obbligatori ed altri no, ed inoltre alcuni di questi devono obbligatoriamente avere lo stesso valore di quelli presenti nel file** `manifest.mf` **pena l'imposibilità (da parte del gestore) di installare il file JAR**

Nome Attributo	Scopo	Richiesto
MIDlet-Name	Nome della MIDlet	Si *
MIDlet-Version	Numero di versione del MIDlet	Si *
MIDlet-Vendor	Autore della MIDlet	Si *
MIDlet-	Riferimento ad un MIDlet specifico all'interno della Suite	Si
MIDlet-Jar-URL	URL del file JAR	Si
MIDlet-Jar-Size	Dimensione del file JAR in byte	Si
MIDlet-Data-Size	Numero minimi di byte richiesti per i recordStore	No
MIDlet-Edition-Description	Testo che descrive il MIDlet	No

*** Questi attributi devono essere identici a quelli definiti del file** `manifest.mf`

Finalmente, dopo una breve, panoramica sull' architettura si può passare a scrivere la prima MIDlet.

Come detto una MIDlet è una sotto classe della classe `javax.microedition.midlet.MIDlet`
e deve implementare alcuni metodi astratti della classe base per fornire alcuni servizi.

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;


public class PrimaMIDlet extends MIDlet {

   // Costruttore della classe
   public PrimaMIDlet() {
      super();
   }

   // Metodo invocato dal sistema quando parte la MIDlet
   protected void startApp() throws MIDletStateChangeException {
   }

   // Metodo invocato dal sistema per mettere in pausa la MIDlet
   protected void pauseApp() {
   }

   // Metodo invocato dal sistema per chiudere la connessione
   protected void destroyApp(boolean arg0) throws MIDletStateChangeException
   {
   }

}
```

La classe MIDlet mette a disposizione i seguenti metodi:

Classe MIDlet

`abstract protected void destroyApp(boolean unconditional)`
Segnala alla MIDlet di terminare e di entrare nello stato *Destroy*

`String getAppProperty(String key)`
Fornisce alla MIDlet il meccanismo per recuperare alcune proprietà definite all'interno del file JAD

`void notifyDestroyed()`
Usato dal una MIDlet per richiedere al sistema di essere terminata e di passare allo stato *Destroy*

`void notifyPaused()`
Notifica al sistema che la MIDlet non vuole essere attiva e di entrare nello stato di *Pause*

`protected abstract void pauseApp()`
Segnala alla MIDlet di fermarsi e di entrare nello stato *Pause*

`void resumeRequest()`
Segnala alla MIDlet che è intenzionata a tornare ad uno stato *Active*

`protected abstract void startApp()`
Segnala alla MIDlet che è entrata nello stato *Active*

Può sembrare, ad una prima lettura dell'elenco dei metodi, che alcuni metodi sembrano avere lo stesso significato. In effetti la MIDlet ha una comunicazione bidirezionale con il gestore delle applicazioni, infatti così come il gestore può mettere in pausa un MIDlet (per esempio per consentire all'operatore di poter rispondere ad una chiamata in arrivo) così il MIDlet può richiedere al gestore di essere messo in pausa o distrutto. Per capire meglio quanto accade è bene spiegare il ciclo di vita di una MIDlet.

Una MIDlet passa attraverso fasi differenti nel suo ciclo di vita e viene sempre considerato in uno di questi tre stati

PAUSA : Un MIDlet viene messo in stato di pausa dopo la chiamata del costruttore ma prima di essere avviato dal gestore di applicazioni. Una volta che il MIDlet è stato avviato, può passare alternativamente tra gli stati di Pausa e di Attivo.

ATTIVO : Il MIDlet è in esecuzione

DISTRUTTO : Il MIDlet ha rilasciato tutte le risorse che aveva acquisito ed è stato terminato dal gestore di applicazioni

Non appena la MIDlet viene creata (attraverso il suo costruttore) per definizione entra nello stato PAUSA anche si il suo metodo `pauseApp()` **non viene invocato.**

La differenza tra i metodi `destroyApp, pauseApp, startApp` **ed i metodi** `notifyDestroyed, notifyPaused, notifyRequest` **è che i primi vengono chiamati dalla MIDlet per indicare un tipo di comportamento (la MIDlet comunica che ha intenzione di mettersi in pausa o di chiudersi) mentre i secondi sono delle richieste che la MIDlet effettua al gestore dell'applicazione. Chiariamo con un esempio.**

Supponiamo di voler implementare un pulsante di EXIT. Applicativamente l'applicazione chiama il metodo della MIDlet `destroyApp()`**. L'esito di questo comando deve essere interpretato come una comunicazione che la MIDlet fa a tutti i suoi componenti dell'intenzione di chiudersi. La semplice invocazione di questo metodo NON chiude la MIDlet. Lo scopo di questo metodo è quello di chiudere tutte le risorse che la MIDlet ha allocato (database, thread, ecc). La vera chiusura dell'applicazione si ottiene con la chiamata al metodo** `notifyDestroyed()`**. Con questo metodo la MIDlet chiede al gestore di poter essere terminata incondizionatamente dato che avrà già provveduto a chiudere tutte le risorse allocate.**
Un esempio di metodo per poter chiudere correttamente un'applicazione potrebbe essere il seguente:

```
public void exitApplication() {
    try {
        destroyApp(false);
        notifyDestroyed();
    } catch (MIDletStateChangeException e) {
    }
}
```

Notate anche come il metodo `destoryApp()` **prenda in input un booleano e possa sollevare un'eccezione. Il boolean che il metodo prende in input non ha un significato particolare, il suoi significato dall'implementazione che fate di quel metodo.**

E' prassi comune utilizzare un valore *false* per indicare al metodo che si sta per uscire e che si desidera richiudere (se possibile) tutte le operazioni attualmente in corso (download in background, database aperti, ...) se non è possibile chiudere tutto, allora il metodo solleva un'eccezione ed in questo caso (nel metodo `exitApplication`**) non verrebbe richiamato il metodo** `notifyDestoryed()` **che, come detto, chiude inesorabilmente la MIDlet. Se invece si passa un valore booleano *true*, la richiesta è da intendere un pò più bruscamente e che quindi, se si stava scaricando un qualcosa, si deve interrompere il processo e consentire l'uscita immediata.**
Come detto non è una regola ma, come dire, è buona prassi.

Un altro metodo importante della classe MIDlet è il metodo `getAppProperty()` **che serve a leggere gli attributi definiti nel file** `manifest.mf`**.**

Un'osservazione va fatta anche sul metodo `startApp()`**.**

Quando un MIDlet sta per essere posto in stato di attività, il gestore di applicazioni chiama il metodo `startApp()`**. Questo metodo può essere chiamato più volte nel corso del ciclo di vita di un MIDlet dato che la MIDlet può passare più volte tra gli stati di PAUSA e di ATTIVO quindi è importante capire bene cosa inserire in questo metodo.**

Il codice che crea strutture dati che devono esistere durante tutto il ciclo di vita di un MIDlet deve essere allocato una volta sola, nel costruttore e non nel metodo `startApp()` **altrimenti si avrebbe la loro inizializzione ad ogni riavvio della MIDlet.**

Con questo, almeno per ora, è tutto.

Nel prossimo articolo ho intenzione di affrontare in dettaglio tutti i componenti delle interfaccia utente disponibili nella libreria MIDP