Corso sul linguaggio Java

Modulo JAVA9

A2 – JVM e processi

Prerequisiti

- Programmazione base in Java
- Concetti di base sui sistemi operativi
- Programmazione degli *stream*
- Valori di ritorno a livello di linea di comando

Introduzione

In molte situazioni può essere comodo che all'interno di un programma sia possibile chiedere l'esecuzione di un'altra applicazione.

Per affrontare questa esigenza, Java mette a disposizione la classe **Runtime** che presentiamo in questa Unità.

La classe **Runtime**

Vediamo ora come sia possibile attraverso la classe Java **Runtime** che da un'applicazione si possa richiamare l'esecuzione di comandi del sistema operativo o di applicazioni varie.

Realizzare una cosa simile, significa fare in modo che il processo **P** associato all'applicazione Java, generi un processo figlio **P1** che inizia ad evolvere in modo indipendente da **P**.

La classe **Runtime**

La JVM associa ad ogni applicazione un oggetto di una classe particolare: la classe **Runtime**; quindi non è necessario creare un oggetto di questa classe, ma è sufficiente eseguire una chiamata al metodo statico **getRuntime**() della classe **Runtime** appunto.

Runtime comando = **Runtime.getRuntime**();

L'oggetto *comando* è il riferimento utile per invocare molti metodi interessanti che, ad esempio, consentono di:

- individuare il numero di processori assegnati
- verificare la memoria assegnata
- avviare il garbage collector

La classe **Runtime**
- Il metodo exec()

Uno dei metodi attivabili sull'oggetto *comando* è **exec**(), di cui esistono i seguenti *overload,* ciascuno dei quali genera un processo:

- **public Process exec** (**String** *command*);
- **public Process exec** (**String** [ ] *cmdArray*);
- **public Process exec** (**String** *command*, **String** [ ] *envp*);
- **public Process exec** (**String** [ ] *cmdArray*, **String** [ ] *envp*);

Ognuno di questi metodi, munito degli opportuni parametri, provoca, da parte del sistema operativo, l'esecuzione di un dato programma e del conseguente processo.

La classe **Runtime**
- Il metodo exec()

Esaminiamo i due overload più comuni del metodo **exec()**:

1. **public Process exec** (**String** *command*)
   Una chiamata a questo metodo comporterà l'esecuzione del comando descritto in *command*. Nasce un processo che sarà del tutto indipendente dall'applicazione e continuerà anche al terminare della stessa.

2. **public Process exec** (**String**[ ] *command*)
   Nasce un processo, come nel precedente, ma offre la possibilità di eseguire più comandi. Alla terminazione di un processo, fa partire, se c'è, il successivo, anche dopo il termine dell'applicazione.

La classe **Runtime**
- I comandi interni del sistema operativo

```
public class Command
{   public static void main (String args[]) throws Exception
    {       Runtime comando = Runtime.getRuntime();
            comando.exec ("cmd.exe /c start dir");
    }
}
```

In questo primo esempio, l'applicazione crea l'oggetto *comando* che esegue il metodo **exec**().

Il metodo **exec**() consente di eseguire un comando interno del sistema operativo, come se fossimo a livello della linea di comando.

Il processo che lancia il metodo **exec**() termina, non avendo altre istruzioni, mentre il processo **cmd** terminerà quando nella finestra si scrive **exit** sulla linea di comando.

La classe **Runtime**

- Esecuzione di applicazioni

```
import java.io.*;
public class EseguiApps
{  public static void main (String[ ] a)
    {  try
       {   Runtime rt = Runtime.getRuntime();
           String cmd = "cmd /c start Winword.exe";
           //String cmd = "cmd /c start Excel.exe";
           //String cmd = "Notepad.exe D:/Promemoria.txt";
           Process proc = rt.exec(cmd);
       }
       catch (Exception e)    {  System.out.println("Errore");    }
  }
}
```

Questo utilizzo di **exec**() consente di lanciare applicazioni presenti sul sistema.

La classe **Runtime**

- Valore di ritorno

```
import java.io.*;
public class ExitValue
{  public static void main (String[] a)
  {    try
    {  Runtime rt = Runtime.getRuntime();
       String cmd = "cmd /c start dir";
       Process proc = rt.exec(cmd);
       System.out.println("Exit " + proc.waitFor());
    }
    catch (Exception e)
    {        System.out.println("Errore");    }
  }
}
```

Il valore di ritorno dell'applicazione può essere testato dalla linea di comando mediante il comando

if %**errorlevel**% **neq** 0 **echo** Errore

Il valore di ritorno dell'applicazione può essere ottenuto anche mediante **exitValue**() ma solo se il processo chiamante termina, altrimenti solleva eccezione.

I metodi della classe **Runtime**

La classe **Runtime** esibisce i seguenti metodi:

Funzioni classe Runtime	
METODO	**EFFETTO**
static Runtime getRuntime()	Dà l'oggetto Runtime associato all'applicazione
int availableProcessors()	Dà il numero di processori disponibili per la JVM
long freeMemory()	Dà la memoria disponibile per la JVM
long totalMemory()	Dà la memoria totale disponibile per la JVM
public Process exec (String *command***);**	Esegue il comando *command*
public Process exec (String [] *cmdArray*);	Esegue i comandi contenuti in *cmdArray*
public Process exec (String *command*, **String [] *envp*);**	Esegue *command* con gli argomenti nell'ambiente *envp*.
public Process exec (String [] *cmdArray*, String [] *envp*, File dir**);**	Esegue *command* con gli argomenti *cmdArray* nella *directory* indicata.

La classe **Process**

- Standard stream

In tutti i moderni sistemi operativi i **canali standard** (o *standard streams*), rappresentano i dispositivi logici di input e di output che collegano un programma con l'ambiente operativo in cui esso viene eseguito (tipicamente un terminale testuale) e che sono connessi automaticamente al suo avvio.

Questi canali predefiniti, disponibili nei sistemi operativi Unix e Unix-like, negli ambienti d'esecuzione del linguaggio C, C++ e Java, sono detti **standard input**, **standard output** e **standard error** (talvolta abbreviati rispettivamente in **stdin**, **stdout** e **stderr**).

La classe **Process**

- Cattura di standard error

Il comando "javac", immesso senza parametri da linea di comando, produce una schermata con la sua sintassi e le opzioni d'uso.

```
import java.io.*;
public class getErrorStream
{  public static void main (String[] a)
 {  try { Runtime rt = Runtime.getRuntime();
          Process proc = rt.exec("javac");
          int exitVal = proc.waitFor();
          System.out.println("Exit " + exitVal);
        }
    catch (Exception e)  { System.out.println("Errore");   }
 }
}
```

Questa applicazione, tuttavia, quando va in esecuzione, non produce la classica schermata dell'help del comando **javac**. Inoltre viene prodotto in output *Exit 2*. Perché? Cosa manca nel punto indicato dalla freccia?

La classe **Process**

- Cattura di standard error

I buffer di input e di output hanno dimensione limitata, per cui un input o un output molto veloci (come il comando "javac" precedente) possono rischiare di far perdere dati in transito e bloccare o far terminare un sottoprocesso.

Quindi, prima di stampare il valore di ritorno occorre implementare un metodo *getOut*() che si occupa di gestire lo *stream* di errore e di stampare linea per linea l'effetto del comando "javac".

Questo metodo utilizza la classe **InputStreamReader** bufferizzata in modo da consentire la lettura riga per riga.

Notiamo che se il programma che lanciamo produce output o attende input, ci dobbiamo assicurare, reciprocamente, che il nostro processo sia in grado di gestire stream di input o di output.

La classe **Process**

- Cattura di standard error

Lo *standard error* o **stderr** è lo *stream* a cui vengono indirizzati i messaggi di errore dei comandi.

```
public static void getOut (Process p)
{try {   InputStream stderr = p.getErrorStream();
         InputStreamReader isr = new InputStreamReader (stderr);
         BufferedReader br = new BufferedReader (isr);
         String line = null;
         System.out.println("<ERROR>");
         while ((line = br.readLine()) != null)  System.out.println (line);
         System.out.println("</ERROR>");
      }
      catch (Exception e)  {  System.out.println("Errore");
   }
 }
```

Il valore di ritorno dell'applicazione è **2** poiché il comando non è andato a buon fine (mancano i parametri)

La classe **Process**

- Cattura di standard output

- Lo *standard output* o **stdout** è lo *stream* a cui vengono indirizzati gli output dei comandi.
- In questo caso il metodo *getOut(proc)* va opportunamente modificato e la sua istanza va fatta in un'applicazione come *ExitValue.java* vista in precedenza, in cui la stampa del contenuto della directory viene fatta da programma.
- La chiamata del metodo *getOut(proc)* va posta prima della stampa del valore di ritorno.
- Se il parametro è un file dati, viene aperto utilizzando l'applicazione relativa.

La classe **Process**

- Cattura di standard output

Lo *standard input* o **stdin** è lo *stream* che produce dati da utilizzare.

```
public static void getOut (Process p)
{try {      InputStream stdin = p.getInputStream();
            InputStreamReader isr = new InputStreamReader (stdin);
            BufferedReader br = new BufferedReader (isr);
            String line = null;
            System.out.println("<OUTPUT>");
            while ((line = br.readLine()) != null)  System.out.println (line);
            System.out.println("</OUTPUT>");
      }
      catch (Exception e)  {  System.out.println("Errore");
  }
}
```

Inserendo questa procedura in un'applicazione simile a *getErrorStream* precedente, si cattura l'output riga per riga.

I metodi della classe **Process**

La classe **Process** esibisce i seguenti metodi:

Funzioni classe Process	
METODO	**EFFETTO**
abstract int exitValue()	Dà il valore di uscita del sottoprocesso
abstract InputStream getErrorStream()	Dà l'error stream del processo
abstract InputStream getInputStream()	Dà l'input stream del processo
abstract OutputStream getOutputStream	Dà l'output stream del processo
abstract int waitFor()	Causa il blocco del thread corrente fino a che termini il processo rappresentato da questo oggetto
abstract void destroy()	Uccide il sottoprocesso

I metodi della classe **System**

- proprietà del sistema

Con la classe **System** si possono ispezionare le caratteristiche del sistema.

```
import java.io.*;
public class systemMethods
{  public static void main (String args[])
   {   System.out.println("System current properties");
       System.out.println("----------------------------------------");
       System.out.println("Name........: " + System.getProperty("os.name"));
       System.out.println("Versione...: " + System.getProperty("os.version"));
       System.out.println("Architettura: " + System.getProperty("os.arch"));
   }
}
```


I metodi della classe **System**

- cronometraggio di operazioni

Si possono cronometrare operazioni anche complesse, attraverso il valore dell'orologio interno. .

```
import java.util.*;
class ElapsedTime
{   public static void main (String args[]) {
     try { long start = System.currentTimeMillis( );
           System.out.println (new Date( ) + "\n");
           Thread.sleep (3000);
           System.out.println (new Date( ) + "\n");
           long end = System.currentTimeMillis( );
           long diff = end – start;
           System.out.println("La differenza e': " + diff + " msec");
         } catch (Exception e)    {  System.out.println("Errore!");
     }
   }
}
```

Argomenti

- La classe **Runtime**
  - Il metodo **exec**()
  - I comandi interni del sistema operativo
  - Esecuzione di applicazioni
  - Valore di ritorno
- I metodi della classe **Runtime**
- La classe **Process**
  - Standard stream
  - Cattura di standard error
  - Cattura di standard output
- I metodi della classe **Process**
- I metodi della classe **System**
  - - proprietà del sistema
  - - cronometraggio di operazioni

Altre fonti di informazione

- P.Gallo, F.Salerno – Informatica Generale 1, ed. Minerva Italica
- M.Romagnoli, P.Ventura – Linguaggio C/C++, ed. Petrini
- M. Bigatti – Il linguaggio Java, ed. Hoepli