

Java per programmatori C++

autore: Matteo Lucarelli

ultima versione su: matteolucarelli.net

Architettura e concetti generali

Cosa scaricare ed installare

Tutti gli strumenti per la programmazione java sono disponibili per il download gratuito sul sito ufficiale (java.sun.com). I pacchetti disponibili sono vari, e ne esistono più versioni. Diamo una breve spiegazione:

- Java Standard Edition SDK (*J2SE_SDK*): pacchetto indispensabile per la programmazione, contiene l'ambiente di run-time, le API, gli strumenti a riga di comando ed alcuni esempi.
- Java Enterprise Edition SDK (*J2EE_SDK*): come il precedente ma include anche web server, application server, ed in generale gli strumenti per la programmazione lato server (quindi non è indispensabile, almeno all'inizio).
- Java Documentation (*J2SDK_Doc*): pacchetto contenente tutta la documentazione necessaria (API e tools) in formato HTML
- NetBeans: ambiente di sviluppo integrato (IDE) per la programmazione Java. Include anche un GUI designer. Open source. Esistono molte alternative, ma citiamo NetBeans in quanto ufficialmente proposto da Sun.
- Java Runtime Environment (*J2SE_JRE*): piattaforma necessaria all'esecuzione di programmi Java. E' incluso nella SDK, e quindi non è necessario sulla macchina da sviluppo (sulla quale invece deve essere installata la SDK).

Dei pacchetti elencati l'unico strettamente necessario è il primo (SDK), oltre ad un editor di testi.

L'idea di semicompilato multiplatforma

Java comprende sia l'idea di linguaggio compilato che quella di linguaggio interpretato, cercando di prendere il meglio da entrambe. Dal codice sorgente infatti il compilatore ottiene un semicompilato (*java bytecode*) che necessita di un interprete (*Java Virtual Machine*) per essere eseguito.

Sorgente--(compilatore)--> Bytecode --(interprete)-->Eseguibile

I pregi più evidenti di questa architettura sono la portabilità, sia del codice sorgente che del semicompilato (entrambi sono identici per ogni piattaforma e sistema operativo), e la innata propensione cross-platform (compilazione ed esecuzione possono avvenire su piattaforme completamente diverse); Il che significa, ad esempio, che è possibile sviluppare su un PC Linux quello che poi dovrà girare su un palmare WindowsCE o su un mainframe Unix.

Il limite maggiore, almeno rispetto al C/C++, sta nell'efficienza (velocità), comunque superiore alla maggior parte dei linguaggi puramente interpretati.

Applicazioni, applets, ecc

In java si possono creare applicazioni stand alone, che devono implementare il metodo main(), e, se eseguite su una macchina dotata di JVM, si comportano come delle normali applicazioni C++ (hanno comunque bisogno dell'installazione del *Java Runtime Environment*).

Si possono creare però anche applicativi di natura diversa, e cioè:

- *applet*: funzionalità da aggiungere a pagine HTML all'interno del tag <APPLET CODE="file.class">. Generalmente sono scaricabili attraverso internet e eseguite attraverso il browser (che deve implementare il plugin per la Java Virtual Machine). Devono implementare un classe derivata dalla classe base *Applet*, contenente almeno un funzione *paint*:

```
import java.applet.*;

import java.awt.*;

public class HelloWorldApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

- servlet: anche queste sono applicazioni pensate per un uso web. La differenza principale con le applet è che il codice viene eseguito lato server. Per una spiegazione esauriente si rimanda alla documentazione ufficiale Sun.

Compilazione ed esecuzione

Per compilare il file (o meglio la *compilation-unit*) “prova.java”, ottenendo (nell’ipotesi che il sorgente contenga una sola classe pubblica di nome *prova*) il bytecode “prova.class”:

```
javac prova.java
```

Per eseguire (nell’ipotesi che la classe *prova* contenga una funzione *main*) “prova”:

```
java prova
```

Va notato come all’interprete vada passato il nome della classe (*prova*) e non il nome del file (*prova.class*).

Naturalmente se i comandi non fanno parte del vostra variabile d’ambiente *PATH* dovreste digitare il percorso completo (in Windows probabilmente *C:\j2sdk<version>\bin*). L’eventuale, tipico, errore *NoClassDefFoundError* è probabilmente causato dal mancato settaggio della variabile d’ambiente *CLASSPATH* (si veda il prossimo paragrafo). In luogo della variabile d’ambiente è anche possibile specificare i path aggiuntivi direttamente nel comando:

```
java -classpath \path\classes\aggiunte\;oppure\file.jar prova
```

Note:

- Il compilatore crea un file *.class* per ogni classe contenuta nel file *.java*.
- ogni *compilation-unit* può contenere una sola classe pubblica. Il nome di tale classe pubblica, se presente, deve coincidere con quello del file sorgente. Si noti quindi che, spesso, ogni compilation unit contiene l’implementazione di una sola classe.
- più classi bytecode possono essere raggruppate, a livello logico, come *package* (assimilabile ad una libreria) o, a livello fisico, come archivio *jar* (il cui utilizzo primario è quello di permettere di raggruppare in un solo file un programma composto da molte classi, e quindi spesso rappresenta “il programma” in Java).
- In Java non esiste la separazione tra definizione ed implementazione (i file *.h* e *.cpp*). La classe viene definita ed implementata nella sua *compilation-unit*.

Package e namespace

Il meccanismo di inclusione di codice o di libreria utilizzato da Java differisce notevolmente da quello del C. Nasce infatti per evitare ambiguità in caso di pubblicazione del codice, e non prevede file di definizione separati (*headers*).

La direttiva

```
package nomepackage
```

posta all'inizio di un sorgente (deve essere la prima riga non commentata) specifica che le classi definite nel sorgente fanno parte della libreria *nomepackage*, e quindi del namespace omonimo. Una classe definita nello stesso file sarà quindi completamente identificata come *nomepackage.unaclasse*, ed identificata in questo modo può essere utilizzata all'interno di altro codice.

Le direttive

```
import nomepackage.unaclasse;

import nomepackage.*;
```

importano all'interno del codice le funzionalità della classe *unaclasse* (nel primo caso) o dell'intero namespace *nomepackage* (nel secondo caso), funzionano cioè un po' come gli *#include* del linguaggio C (in realtà però non importano realmente il codice ma definiscono i *namespace* necessari al successivo codice).

Per permettere alla JVM di localizzare nel filesystem le classi ed i namespace utilizzati è necessario che la variabile d'ambiente *CLASSPATH* sia impostata alla radice del repository delle classi (ad esempio *CLASSPATH=C:\JAVA\LIB*). A partire da tale directory le classi saranno cercate utilizzando il loro namespace, quindi la classe *net.matteolucarelli.utils.parser.class* verrà cercata in *C:\JAVA\LIB\net\matteolucarelli\utils* (e quindi le classi di uno stesso *package* vanno poste in una singola directory). Naturalmente tale specifica non è necessaria per le classi che fanno parte della JVM.

Note:

- è buona abitudine iniziare la propria gerarchia dei *namespace* con il proprio dominio invertito (o con il proprio nomecognome) in modo da evitare ambiguità nel caso di successiva pubblicazione in rete.
- L'appartenenza ad un package ha anche un'effetto sulla accessibilità: le classi prive di specificatore (cioè non definite né *public*, né *protected*, né *private*) sono considerate *friendly*, cioè accessibili solo dalle altre classi appartenenti allo stesso *package*.

Java API

L'ambiente di programmazione java viene fornito con un vasto set di classi standard (ovviamente indipendente dalla piattaforma). Riportiamo alcuni dei package principali:

- *java.lang* : le classi fondamentali (importate per default)
- *java.util* : funzioni di utilità comune (random, list, ecc.)
- *java.io* : funzioni di input/output, manipolazione file, ecc.
- *java.math* : funzioni matematiche a precisione arbitraria
- *java.applet* : funzioni necessarie alla creazione di applet
- *java.awt* e *javax.swing* : funzioni per la creazione di GUI
- *java.net* : funzioni di networking
- *java.sql* : funzioni orientate all'interazione con basi di dati

La documentazione completa dell'API è disponibile sul sito *java.sun.com*.

Differenze Sintattiche

Oggetti

Java è un linguaggio strettamente object oriented. Il che significa che tutto il codice deve essere contenuto in classi, e quindi che non esistono funzioni che non siano metodi di qualche classe (funzioni globali o *top level*).

La funzione main (*top-level* in C++) deve essere definita come metodo statico pubblico di una classe pubblica (che può anche essere l'unica classe del codice):

```
public class Test {
    public static void main(String argv[]) {
        /*
         * ..codice della main..
         */
    }
    /*..altro codice della classe Test..*/
}
```

Ad esempio le comuni funzioni analoghe a quella della libreria standard C sono generalmente metodi dell'oggetto statico System:

```
System.out.println("Hello Word!"); // stampa una linea sul terminale
System.exit(0); // termina l'esecuzione del programma
System.in // lo standard input
System.err // lo standard error
```

Primitive

Ogni tipo primitivo ha una dimensione definita, indipendente dalla piattaforma e dal sistema operativo, per questo motivo in Java non esiste l'operatore *sizeof*.

Tipo	Dimensione	Reference
boolean	(true/false)	Boolean
char	16bit	Character
byte	8bit	Byte
short	16bit	Short
int	32bit	Integer
long	64bit	Long
float	32bit	Float
double	64bit	Double
void	-	Void

Tipi di primitive forniti da Java

Esistono inoltre i due tipi predefiniti per calcoli ad elevata precisione: *BigInteger* e *BigDecimal*. Entrambi possono rappresentare quantità con precisione arbitraria.

Istanza e concetto di reference

Ogni istanza, che non sia relativa ad una primitiva, deve essere esplicita, cioè espressa dalla direttiva *new*. La sola

definizione di tipo non rappresenta un'istanza, ma solo la creazione di una *reference* ad un oggetto. L'idea di *reference* è simile a quella di puntatore ad oggetto del linguaggio C++ (anche se non può essere manipolata direttamente).

```
int i; // istanza valida (perchè int è una primitiva)

Integer i = new Integer(); // è equivalente alla precedente, mentre
Integer i; // crea solo la reference

String s; // non è un'istanza: crea una reference a String, infatti
s="abcd"; // genera errore

String s = new String(); // istanza corretta
```

Le stringhe, invece, possono essere inizializzate in questo modo:

```
String s= "abcd";
```

questa, comunque, è una sintassi particolare valida solo in questo caso. E' quindi meglio abituarsi ad usare sempre la forma esplicita.

Si noti che le parentesi nell'istanza sono sempre necessarie, anche se il costruttore della classe non prevede argomenti.

Scope

Il Java supporta le stesse regole di scope (ambito di validità degli oggetti) del C/C++. L'unica differenza è che le variabili a scope limitato non possono nascondere quelle di scope più ampio aventi lo stesso nome.

Es (il seguente codice, lecito in C, genera errore in Java):

```
int x;
{
    int x;
}
```

Garbage Collector

Gli oggetti non vanno esplicitamente cancellati e non esiste neppure un modo esplicito per farlo (in modo analogo alla direttiva *delete*). Esiste invece il meccanismo di *garbage collector*, che serve appunto a liberare dalla memoria gli oggetti che non hanno più alcuna *reference* (e quindi non sono più accessibili). Questa è una comodità perchè non ci si deve preoccupare della distruzione, ma è necessario ricordare che il *garbage collector* lavora in modo asincrono, quindi non c'è alcuna certezza sul momento della liberazione della risorsa, e neanche sulla sua effettiva esecuzione. Nel caso in cui si rendessero necessarie operazioni di deallocazione non standard è possibile inserirle nel metodo *finalize()*, che viene chiamato dal *garbage collector*.

Valore di default

Ogni oggetto di tipo primitivo viene inizializzato al momento dell'istanza. Il valore di default è **false** per i boolean, **0** (zero) per tutti gli altri tipi primitivi e **null** per le reference.

L'attributo static

È possibile riferirsi a metodi e proprietà statiche sia utilizzando un oggetto che utilizzando il direttamente nome della classe (perché l'oggetto statico è condiviso tra tutte le istanze della classe, e quindi i due sistemi coincidono). Il secondo metodo è preferibile perché sottolinea la natura statica dell'oggetto, non essendo applicabile ad oggetti non statici.

Es:

```
class Prova{

    /* definizione dell'attributo statico */
    static int i=10;
}

/* utilizzo dell'attributo statico - metodo preferito */
Prova.i++;

/* utilizzo dell'attributo statico - metodo alternativo */
Prova p=new Prova();
p.i++;
```

Array

Per la creazione di una *reference* ad un array (attenzione, non viene creato l'oggetto array ma solo un "puntatore" ad array) esistono due sintassi equivalenti:

```
int[] Ar1;
int Ar1[];
```

Creazione ed istanza di un array (notare l'attributo read-only "length"):

```
// array a dimensione nota
int[] Ar1 = {0,1,2,3,4,5,6,7,8,9};

// array a dimensione definita da variabile
int dim=10;
Float[] ar = new Float[dim]; // crea un array di reference a Float
for (int i=0; i<ar.length; i++) ar[i]=new Float(); // istanzia i Float
```

Il Java permette inoltre di avere array di elementi non omogenei. Per fare ciò si usa l'oggetto *Object* (da cui ogni altro oggetto è derivato):

```
Object[] Ar = { 1, "abc" , 1.25 , 1D };
```

Naturalmente gli array possono avere dimensioni multiple:

```
int[][] a = {{0,1,2}{3,4,5}}; // crea e inizializza un array 3x2

int[][][] a = new int[2][5][8]; // crea un array 2x5x8

//crea ed inizializza un array 2x2 di oggetti non-primitive
String[][] sa = {{new String(""),new String("")}
                 {new String(""),new String("")}}
```

Esiste inoltre una definizione di array dinamico (quindi a dimensione variabile), chiamato *vector*. L'unico limite di tale oggetto è la restrizione ad array di *Object*.

```
import java.util.Vector;

Vector v=new Vector(); // creazione
v.add(new Integer(0); // aggiunta di un elemento Integer
v[0] = 10; // modifica di un elemento
Object o=v.get(0); // estrazione di un elemento
Integer i=(Integer)(v.get(0)); // estrazione e casting
```

Containers

Oltre agli array il java mette a disposizione, nel package *java.util*, altri tipi aggregati:

- *Collection*: gruppo dinamico di elementi, da cui derivano altri tipi tra i quali citiamo : *List* (gruppo ordinato), *Set* (non può contenere elementi duplicati), *Queue* (FIFO),
- *Map*: gruppo di coppie di elementi (key/value), da cui derivano altri tipo tra i quali citiamo: *HashMap* (per hash table, ottimizzata per l'accesso rapido) e *TreeMap* (ordinata).

Ogni tipo aggregati implementa i metodi comuni necessari alla manipolazione, come: *size*, *isEmpty*, *add*, *remove*, ecc. L'use dei container richiede attenzione dovuta al fatto che il tipo contenuto è generico (*Object*).

Nel seguente esempio la funzione *fill* ritorna una *Map* contenente tre elementi:

```
static Map fill(Map m){
    m.put("delfino", "acqua");
    m.put("rondine", "aria");
    m.put("giraffa", "terra");
    return m;
}
```

Il seguente esempio illustra l'uso dell'iterazione "for each" applicato agli elementi di una collection:

```
for (Object o : collection) System.out.println(o);
```

Per una descrizione esaustiva dei metodi relativi ai tipi aggregati si rimanda alla documentazione ufficiale (in particolare relativa alla versione posseduta, visto che le implementazioni dei tipi aggregati si sono evolute in versioni successive).

L'attributo *final*

L'attributo *final* ha due significati distinti. Se è riferito agli oggetti primitivi ne etichetta costante il valore, mentre con gli altri oggetti ne etichetta come costante la reference.

L'uso contemporaneo di *static* e *final* viene utilizzato per definire valori costanti *hard-coded*, e per convenzione i nomi sono espressi con caratteri maiuscoli (un po' come per le *#define* del C).

Gli oggetti etichettati come *final* ma non inizializzati devono forzatamente essere inizializzati nel costruttore.

Es:

```
// definizione di una costante hard-coded
static final int PI=3.1415;

// assegnamento run-time di una costante
// il valore assegnato non potrà essere cambiato run-time
final int seed = (int)(Math.random());

final String str = new String("abcd");
str = "defg";           // consentito
str = new String();    // errore: la reference deve restare costante
```

L'attributo *final* utilizzato su un metodo può avere due scopi separati: il primo è impedire ogni eventuale ridefinizione in classi ereditate, il secondo è permettere alcune ottimizzazioni (simili ad *inline* del C). Si noti che la definizione *private* contiene implicitamente quella di *final*, perchè non permette estensioni da parte di classi ereditate.

Infine l'attributo *final* può essere utilizzato sull'intera definizione di una classe. In questo caso impedisce ogni ulteriore derivazione su quella classe.

I thread

L'esecuzione asincrona di codice è possibile secondo due implementazioni sostanzialmente equivalenti. La più comune prevede di creare una classe derivata dalla classe base *Thread* e che sovrascriva il metodo *run()*, che rappresenta l'entry-point del thread (o il suo loop principale):

```
class Thr1 extends Thread {
    Thr1(){
        //codice del costruttore
    }
    public void run() {
        //codice della funzione di ingresso
    }
}

// in un'altra classe...
// avvio del thread
Thr1 t1 = new Thr1();
t1.start();
```

```
...
t1.join();      // attesa del termine del thread
```

Nel caso di codice multi-thread il programma termina quanto tutti i thread sono terminati. Java offre inoltre un altro tipo di thread, chiamato *daemon-thread* la cui esecuzione non inibisce la terminazione del programma. Perchè un thread sia definito come *daemon* è sufficiente che, relativamente all'oggetto thread, sia presente la chiamata:

```
Thread.setDaemon(true);
```

Esistono inoltre due differenti implementazioni di thread ad alto livello, dette timer. Il seguente codice implementa un timer chiamato dopo 3 secondi (per il secondo tipo di timer – *javax.swing.Timer* – si rimanda alla documentazione ufficiale):

```
public class Sveglia{
    Timer timer;
    public Sveglia() {
        timer = new Timer();
        timer.schedule(new Compito(),3000);
    }
    class Compito extends TimerTask {
        public void run() {
            System.out.println("Sveglia");
            timer.cancel(); //cancella il timer
        }
    }
}
```

Per proteggere i dati condivisi da thread concorrenti esiste l'attributo *synchronized*. Tale attributo definisce una sezione critica. Tale sezione è accessibile solo da un thread alla volta, mettendo in wait gli eventuali concorrenti fino ad accesso ultimato:

```
public class OggettoSincronizzato {
    public synchronized void SezioneCritical(){
        ...
    }
    public synchronized void SezioneCritica2(){
        ...
    }
}
```

Esistono inoltre le funzioni di *wait* (sospende il thread indefinitamente o fino ad un timeout stabilito), *await* (sospende fino ad un evento definito), *notify* e *notifyAll* (per svegliare i thread in stato di wait) e la possibilità di definire un *lock* esplicito (oggetto Lock).

Interfacce

La scrittura di un'interfaccia consiste nella definizione di una classe non implementata.

```
public interface contatore{  
    void start(int);  
    void stop();  
}
```

La classe che implementa un'interfaccia deve implementare almeno le funzioni definite dell'interfaccia:

```
public cont1 implements contatore{  
    void start(int){  
        // codice della funzione start  
    }  
    void stop(){  
        // codice della funzione stop  
    }  
}
```

La classe che implementa un'interfaccia deve implementare almeno le funzioni definite dall'interfaccia. Le interfacce possono inoltre contenere la definizione di costanti.