Java per programmatori C++

autore: Matteo Lucarelli

ultima versione su: matteolucarelli.altervista.org

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.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 (Eclipse in testa), ma citiamo NetBeans in quanto è quello ufficialmente proposto.
- 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 (almeno inizialmente) è il primo (SDK), oltre ad un editor di testi ed una console funzionante.

Nomi e organizzazione dei vari pacchetti possono ovviamente cambiare con l'uscita di nuove versioni. Si rimanda al sito ufficiale per un elenco aggiornato.

L'idea di semicompilato multipiattaforma

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 (il *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 che si comportano come delle normali applicazioni C/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 hwApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString("Hello world!", 50, 25);
        }
}
```

  La JDK standard include anche il comando *appletviewer* per eseguire localmente le applet (va eseguito con argomento la pagina HTML che include l'applet)

- *servlet*: anche queste sono applicazioni pensate per un uso web. La differenza principale con le applet è che il codice viene eseguito lato server. L'utilizzo diretto delle servlet è sempre più raro essendo preferito un utilizzo indiretto basato su JSP (Java Server Pages) oppure su framework (Tomcat, JBoss).

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):

```
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* dovrete digitare il percorso completo (in Windows probabilmente *C:\j2sdk<version>\bin*). L'eventuale, tipico, errore *NoClassDefFoundError* è spesso 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\a\classi\aggiunte\oppure\a\file.jar prova
```

Il compilatore crea un file .class per ogni classe contenuta nel file .java. Laddove il file contenga anche delle classi anonime verrano creati anche dei file con i caratteri "$n" nel nome. 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.

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* (si veda comunque più avanti il paragrafo *interfacce*.

Jar

I file .jar sono archivi java, analoghi ai tipici tgz o zip. Più classi bytecode possono essere raggruppate 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.

Il bytecode contenuto in un archivio jar può essere eseguito senza prima scompattarlo:

```
java -jar archivio.jar
```

L'archivio può essere utilizzato anche come repository di codice comune e conseguentemente indicato nel classpath.

La creazione di un archivio è analoga alla creazioni di un tgz:

```
jar cf archivio.jar contenuto
```

dove il contenuto è rappresentato da una lista di file e/o directory.

Un archivio eseguibile deve includere un file *Manifest* che contiene alcune indicazioni utili (versione della JDK) ma soprattutto indica in quale classe si trova la main. Per creare un archivio eseguibile è necessario quindi creare un file di testo (Manifest.txt) con almeno il seguente contenuto:

```
Main-Class: class
```

ad esempio con il comando

```
echo Main-Class: class > Manifest.txt
```

dove *class* indica il nome della classe entry-point (ovvero quella che contiene la main). Successivamente l'archivio va creato indicando il file:

```
jar cfm MyJar.jar Manifest.txt MyPackage/*.class
```

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 del package *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 *using namespace* del linguaggio C, ovvero evitano di dover qualificare il package per ogni oggetto utilizzato, ad esempio:

```
javax.swing.JFrame f = new javax.swing.JFrame("SwingExample");
```

se preceduto da

```
import javax.swing.*
```

diventa

```
JFrame f = new JFrame("SwingExample");
```

Per permettere alla JVM di localizzare nel filesystem le classi ed i namespace specifici 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 *matteolucarelli.utils.parser.class* verra cercata in *C:\JAVA\LIBmatteolucarelli\utils* (e quindi le classi di uno stesso *package* vanno poste in una singola directory). Tale specifica non è necessaria per le classi che fanno parte nativamente della JVM (ovvero le classi di base).

Note:

- E' buona abitudine iniziare la propria gerarchia dei *namespace* con il proprio dominio invertito (o con il proprio nomecognome) in modo da evitare successive ambiguità.
- 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*.

La sirettiva *import* può anche essere qulificata come *static*. In tal caso gli oggetti della classe importata potrenno, nel seguito, essere utilizzati senza qualificarne la classe (in modo simile alla *using* del C).

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.applet* e *java.servlet:* funzioni necessarie alla creazione di applet e servlet
- *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

Garbage Collector

Gli oggetti non vanno esplicitamente cancellati e non esiste neppure un modo esplicito per farlo (non esiste quindi una *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*.

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

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

Principali comandi

Il pacchetto JDK include diversi comandi utili alla gestione ed allo sviluppo in java, questi alcuni dei principali:

- **javac** : il compilatore
- **java** : l'interprete (o launcher)
- **appletviewer:** esecutore di applet locali
- **ControlPanel:** pannello di controllo per settaggi di sicurezza, networking, debug, ecc
- **jar:** archiviatore. Estrae, crea e modifica i file jar
- **jconsole:** permette di agganciare un processo in esecuzione e mostrarne informazioni e risorse
- **javadoc:** crea documentazione HTML a partire dai commenti nei sorgenti
- **javap:** disassemblatore: permette di estrarre informazioni varie da un file .class
- **extcheck:** verifica eventuali conflitti prima dell'installazione di un nuovo jar
- **jdb:** il debugger

Debug

Il debug di un'applicazione java si svolge sostanzialmente allo stesso modo di quanto avviene per gdb: compilare con l'opzione -g (*javac -g sorgente.java*) e lanciare il debugger (*jdb programma*). Al prompt di jdb sono disponibili vari comandi, questi i principali:

- **run:** esegue il programma
- **print:** mostra il valore di oggetti e primitive, può accettare anche un'espressione complessa, Es: *print new java.lang.String("Hello").lenght*
- **dump:** simile a print ma più dettagliato
- **where:** stampa il contenuto dello stack
- **stop:** imposta un breakpoint, accetta numeri di linea (*stop at class:22*), istruzioni (*stop in java.lang.String.lenght*), ecc.
- **cont:** riprende l'esecuzione dopo un breakpoint
- **step:** avanza l'esecuzione

Gestione dei progetti

Per la gestione dei progetti, analogamente a C/C++, può essere utilizzato make. Un semplice makefile generico per java:

```
# nome del progetto
PROJ = HelloWorld

# lista dei sorgenti java separati da spazi
SOURCES = HelloWorld.java

# comando del compilatore
JC = javac

# flag di compilazione (-g include le informazioni di debug)
JFLAGS = -g

CLASSES = $(SOURCES:.java=.class)

default: $(CLASSES)
	echo Main-Class: $(PROJ) > Manifest
	jar cfm $(PROJ).jar Manifest *.class
	$(RM) Manifest

.SUFFIXES: .java .class
.java.class:
	$(JC) $(JFLAGS) $*.java

.PHONY: clean
clean:
	$(RM) *.class
	$(RM) $(PROJ).jar
```

Programmazione ad oggetti

Classi ed oggetti

Java è un linguaggio strettamente object oriented. Il che significa che tutto il codice deve essere contenuto in classi, e quindi non esistono funzioni

che non siano metodi di qualche classe (funzioni globali o *top level*). Eventuali globali possono essere sostituite da metodi e variabili statiche.

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
}
```

Le comuni funzioni 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
```

Nella definizione di una classe il costruttore non è esplicitamente richiesto. Il metodo *finalize()* può essere utilizzato come sostituto del distruttore. Se necessario il costruttore è semplicemente una funzione pubblica, priva di valore di ritorno, avente lo stesso nome delal classe:

```
public class Test {
        public Test();
}
```

Tutte le classi hanno una superclasse, ovvero ereditano da una classe precedente. Se l'ereditarietà non è esplicita la classe eredita dall'oggetto base *Object*.
Come in C++ la classe derivata accede ai metodi *protected* e *public* ma non ai metodi *private* della sua superclasse. L'ereditarietà utilizza la parola chiave *extends*.

Il linguaggio permette anche l'esistenza di classi anonime ovvero non riferite a nessuna variabile. Le classi anonime vengono utilizzate tipicamente per definizioni temporanee o locali:

```
JButton b = new JButton("Close");

// l'oggetto di classe ActionListener risulta anonimo
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        f.dispatchEvent(new WindowEvent(f, WindowEvent.WINDOW_CLOSING));
    }
});
```

Il linguaggio Java non permette l'overloading degli operatori, ovvero non è possibile, come in C++, ridefinire il comportamento di un operatore su uno specifico oggetto.

Annidamento

Il java permette l'annidamendo delle classi. Le classi annidate possono o meno essere definite statiche. Per le classi statiche si usa generalmente le definizione di *nested* mentre per quelle non statiche si usa la definizione di *inner*. Le classi nested (statiche) non possono riferirsi direttamente ai metodi e variabili della classe contenitore. Le classi inner invece possono accedere anche a metodi e variabili private della classe contenitore.

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

L'istanza della classe annidata può evidentemente esistere solo all'interno della classe contenitore.

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

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
        }
}
```

Le interfacce possono inoltre contenere la definizione di costanti. Gli utilizzi più comuni per le interfacce consistono nella definizione di oggetti intercambiabili (ovvero che implementano la medesima interfaccia) oppure per nascondere l'implementazione, pubblicando la sola interfaccia.

Reference

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
String s2 = s;             // assegnazione di una reference
```

Dopo l'assegnazione la reference avrà tutte le proprietà dell'oggetto di partenza anzi, è più corretto dire che, entrambe le reference (quella creata all'istanza e quella assegnata successivamente) sono perfettamente equivalenti.

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

Assegnazione

L'operatore di assegnazione ('=') applicato agli oggetti assegna la reference, non il valore. si consideri ad esempio il seguente codice, che si comporta in modo molto diverso rispetto al C++:

```
Class a=new Class
Class b=new Class
b.set(1);
a = b;         // assegnazione by refernce, quindi da qui a e b si riferiscono allo stesso oggetto
b.set(2);
// ... qui b è uguale ad a
```

Per lo stesso motivo non è possibile scrivere una espressione tipo:

```
Class a, b, c;
a = b + c;
```

Che va invece scritta

```
a=b.add(c);
```

A differenza del C/C++, dove l'ordine non è definito, in Java gli operatori aventi la medesima precedenza vengono sempre valutati da sinistra a destra. Per il resto l'ordine di precedenza è sostinzialmente lo stesso del C++.

Passaggio di parametri

Il java definisce il passaggio di parametri sempre by-value. In realtà per tutti i tipi non primitivi, ciò che viene passato by-value è la reference all'oggetto e ciò, per un programmatore C, assomiglia più al passaggio by reference perché come se si passasse, by-value, un puntatore.

```
public class esempio {
        static void change(anumber n){
                n.val=2;
        }
        public static void main(String[] args) {
                anumber a=new anumber(1);
                System.out.println(a.val); // stampa 1
                change(a);
                System.out.println(a.val); // stampa 2
        }
}
```

```
class anumber{
        public anumber(int v){
                val=v;
        }
        public int val;
}
```

Questo secondo esempio chiarisce meglio il concetto di *reference-by-value*:

```
public void foo(aclass a) {
        a = new aclass("B");
}

aclass a = new aclass("A");
foo(a);
```

dopo la chiamata a *foo* la reference *a* continua a puntare all'oggetto originale (quello creato con parametro "A") e non al nuovo oggetto istanziato nella funzione *foo* con il parametro "B".

Altre differenze

Ordine delle definizioni

In java l'ordine delle definizioni non ha importanza, ovvero la definizione di una classe, di una funzione o di una variabile può anche essere successiva al suo uso. Si veda l'esempio del paragrafo **Passaggio di parametri** dove la classe anumber è definita sotto la main. Allo stesso modo il costruttore non deve essere la prima funzione di una classe e la main può essere posta ovunque.

Per questo motivo è fortemente consigliabile adottare un proprio ordine e seguirlo. Una delle convenzioni possibili è: variabili statiche, altre variabili, costruttore, altri metodi. Per ognuno dei gruppi: prima le pubbliche, poi le protette ed infine le private.

Scope

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

Ad esempio il seguente codice, lecito in C, genera errore in Java:

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

Enum

L'implementazione dei tipi enumerativi è più completa di quella di C/C++. Gli elementi sono infatti degli oggetti. E' utilizzato il costrutto analogo al C:

```
public enum Day {
    SUNDAY,
    MONDAY,
    // ecc...
}
```

che può però essere esteso con degli argomenti che verranno passati al costruttore:

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    Planet(double mass, double radius) {
        // ...
    }
    // ...
}
```

Stringhe

A differenza degli altri oggetti Le stringhe possono essere inizializzate in questo modo:

```
String s=“abcd”;
```

Questo rivela una particolarità del java, ovvero che l'oggetto stringa è immutabile, non esiste cioè una assegnazione, ovvero ogni successiva assegnazione causa l'istanza i una nuova stringa. Si consideri ad esempio:

```
a = b;                  // assegna la reference a alla stringa b ?
b = "ghi";              // modifica b ?
System.out.println(a);  // a dovrebbe quindi essere uguale a b ?
```

In realtà alla terza istruzione a possiede ancora il precedente valore (ovvero non "ghi"), perchè la seconda istruzione non modifica il valore di b ma crea una nuova stringa.

L'attributo static

L'attributo static assume significati di versi in funzione della sua applicazione:

- variabile statica: comune a tutte le istanze della classe (analogo ad una variabile globale)
- metodo statico: esiste indipendentemente dall'istanza della classe (analogo ad una funzione globale)
- classe statica (solo per classi annidate): non può accedere agli attributi della classe contenitore
- blocco statico: viene eseguito al caricamento della classe
- import statico: permette di riferisi agli oggetti importati senza qualificatori

È possibile riferirsi a metodi e variabli 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.

```
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++;
```

Analogamente al C/C++ un metodo statico può accedere solo a variabili statiche e metodi statici.

Esempio di blocco statico (che non esiste in C/C++):

```
public class HelloWorld {
    static {
        System.out.println("Loaded public class HelloWorld");
    }
    //....
}
```

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 le #define del C/C++).

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

```
// definizione di una constante 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 agli *inline* del C). Si noti che la definizione *private* contiene implicitamente quella di *final,* perchè non permette estensioni da parte di classi ereditate.

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

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 *lenght*):

```
// 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.lenght; 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 e 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
```

Annotazioni

Le annotazioni sono utilizzate per impartire istruzioni al compilatore o all'interprete. Iniziano con il carattere '@'. Le più comuni sono:

```
@Override         // Controlla che il metodo sovrascriva il corrispondente metodo nell'interfaccia.
@Deprecated       // Etichetta il metodo come obsoleto
@SuppressWarnings // Silenzia gli warning
```

Esiste anche la possibilità di creare delle annotazioni ad hoc.

For each

Della classica sintassi *for* è disponibile una variante (for each) che permette di iterare una struttura dati:

```
int [] array;

// ...
int sum=0;
for(int item:array){
    sum += item;
}
```

Sono iterabili in questo modo array, vector ed in generale tutte le strutture dotate di iteratore.

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

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

API

Stringhe

La gestione delle stringhe è analoga a quella dell'oggetto *string* nella *stl* del c++.
Per la formattazione è disponibile il metodo *format* con una sintassi analoga a quella della *printf*:

```
String = String.format("Result=%f",floatVar);
```

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;
}
```

le implementazioni dei tipi aggregati si sono evolute in versioni successive quindi per una descrizione esaustiva meglio consultare la documentazione ufficiale della specifica versione di JDK.

Thread

L'esecuzione asincrona di codice è possibile secondo diverse implementazioni. Le principali sono la classe Runnable e la classe Thread.

La prima prevede di creare una classe che implementi l'interfaccia Runnable e contanga almeno il metodo *run()* ,che rappresenta l'entry-pont del thread (o il suo loop principale):

```
public class Thr1 implements Runnable {
        public void run() {
                //... codice della funzione di ingresso
        }
}

// ...altrove, avvio del thread:
Thr1 t1 = new Thread(new Thr1());
t1.start();
```

La seconda prevede di creare una classe derivata dalla classe base Thread e che sovrascriva il metodo *run(),*

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

// ...altrove
// avvio del thread
Thr1 t1 = new Thr1();
t1.start();
// ...
// attesa del termine del thread
t1.join();
```

La principale differenza tra le due implementazioni è che la prima permette di accedere alle funzioni anche senza lanciare il thread (chiamandole direttamente invece che come argomento di 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'ogetto thread, sia presente la chiamata:

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

Esistono inoltre alcune implementazioni di thread ad alto livello dette timer. Il seguente codice implementa un timer chiamato dopo 3 secondi:

```
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 SezioneCritica1(){
        // ...
    }
    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).

Stream

Gli stream vengono utilizzati per le operazioni di input e output. Esistono diverse tipologie di stream a seconda dell'oggetto trattato (byte, caratteri, stringhe ,ecc)

```
// Esempio di bytestream per copiare file

FileInputStream in = new FileInputStream("infile");      // o FileReader (per charstream) o BufferedReader (per stringhe)
FileOutputStream out = new FileOutputStream("outfile");  // o FileWriter (per charstream) o PrintWriter (per stringhe)
int c;

while ((c = in.read()) != -1) {
   out.write(c);
}
```

Gli stream devono essere chiusi tramit il metodo *close* quando non più in uso.

GUI

Per la creazione di interfacce grafiche sono disponibili diversi toolkit, i principali sono:

- **JavaFx:** è l toolkit ufficile di Oracle, partendo da zero è probabilmente la scelta giusta in quando prevedibilmente sarà quello meglio sviluppato in futuro.
- **Swing:** è il precedente toolkit ufficiale di Oracle, ancora molto utilizzato. E' ricco di componenti anche di terze parti.
- **AWT:** è quello più basilare, su cui è basato swing. Offre buone prestazioni ma manca di componenti avanzati. Adatto ad applicazioni semplici e ben testato.
- **SWT:** creato da IBM per Eclipse, è indipendente sia da swing che da AWT.
