

Introduzione a Java

La piattaforma

La piattaforma Java comprende:

- un linguaggio di programmazione
 - orientato agli oggetti (incapsulamento, ereditarietà, polimorfismo, *late-binding*)
 - distribuito (RMI, CORBA, URLs, ...)
 - semplice
 - non c'è l'aritmetica dei puntatori, per cui non è possibile accedere a regioni di memoria non inizializzate o non previste
 - è prevista la garbage collection, che consente il recupero automatico di regioni di memoria occupate da oggetti per i quali non ci sono più riferimenti (quindi nessun problema se si dimenticano `free` e `delete`)
 - multithreaded
 - sicuro
 - indipendente dalla piattaforma HW e SW che ospita la VM
- un ambiente di sviluppo (compilazione, interpretazione/esecuzione, generazione della documentazione, impacchettamento del codice)
- un insieme di ambienti di dispiegamento, per ciascuno dei quali è specificata una raccolta classi che definisce l'API (interfaccia di programmazione):
 - J2SE: una VM ed un insieme di classi Java per l'esecuzione di applicazioni in maniera isolata (o *standalone*) o all'interno in browser web (i cosiddetti *Applet*)
 - J2ME: per applicazioni dedicate ai dispositivi mobili (palmari, cellulari, ...) che, rispetto ai calcolatori desktop, presentano caratteristiche peculiari per quanto riguarda interazione, connettività, consumi, prestazioni.
 - J2EE: per applicazioni lato server.

La specifica

La specifica della JVM definisce l'architettura di una macchina astratta, che garantisce la portabilità del codice. Copre numerosi aspetti, fra cui

- **insieme di istruzioni.** Il codice compilato (*bytecode*) è rappresentato in un formato binario indipendente dall'architettura HW e dal sistema operativo. Le istruzioni sono composte da un codice operativo di lunghezza pari ad 1 byte e da zero o più operandi. L'insieme delle istruzioni è intenzionalmente non ortogonale;
- **tipi di dati** ed operazioni specifiche;
- **aree di memoria**
 - un insieme di **registri** per ogni thread (un thread è un flusso di esecuzione all'interno di un programma, come vedremo meglio in seguito);
 - uno *stack* per ogni thread di esecuzione;
 - *heap*, memoria dinamica condivisa tra i threads. In Java è presente un meccanismo di *garbage collection*, che non richiede la deallocazione esplicita della memoria;
 - un'area riservata al codice dei metodi (*method area*). Quest'area è condivisa tra i threads, e comprende le costanti, i campi e i metodi delle classi;
- formato dei files contenenti le classi

L'interprete

- carica il codice

- verifica il codice, ed in particolare controlla che

- i files contenenti le classi soddisfino il formato specificato
- non abbiano violazioni delle restrizioni di accesso
- non si generino errori di stack over-/under-flow
- i tipi dei parametri siano compatibili con quelli richiesti per il codice operativo
- non vengano effettuate conversioni di tipo non valide

- esegue il codice

Tutorial

Una **classe** rappresenta un tipo di dato complesso, che prevede **attributi** e **metodi**. In pratica può essere vista come l'evoluzione di una struttura del C che, oltre a prevedere un insieme di campi atti a memorizzare i dati (gli “attributi”), raccoglie anche le funzioni che operano sui dati (i “metodi”).

Istanziando una classe, cioè creando un **oggetto** di una determinata classe, si ottiene alloca una regione di memoria che, in generale, contiene sia i dati, sia le funzionalità necessarie per operare su questi. All’insieme dei dati contenuti in un oggetto, ovvero nei suoi attributi, si fa in genere riferimento con il termine di stato dell'oggetto.

In Java, la sintassi per la definizione di una classe è la seguente:

```
[modificatori] class nome_classe {  
    corpo_della_classe  
}
```

Possibili modificatori sono quelli di visibilità (`public`, `protected`, `private`, `package`), nonché `abstract` e `final`. Per quanto riguarda la visibilità, per i nostri scopi possiamo limitarci a considerare il solo modificatore `public`. Per `abstract` e `final` si rimanda la descrizione ad un secondo momento.

La sintassi per la definizione degli attributi, all'interno del corpo di una classe, è:

```
[modificatori] tipo nome [=valore_predefinito];
```

Anche per un attributo si può specificare uno dei 4 modificatori di visibilità visti sopra: trascurando per il momento i modificatori `protected` e `package`, ci limitiamo a dire che un attributo `private` è visibile solo ai metodi della classe, mentre un attributo `public` è visibile anche all'esterno della classe; in linea di

principio è sconsigliato l'uso di attributi pubblici, preferendo la definizione di opportuni metodi che ne consentano lettura/scrittura.

Ai modificatori di visibilità si aggiungono i modificatori `static` e `final`: il primo indica che all'attributo si può accedere anche senza istanziare la classe (una sorta di variabile globale), mentre il secondo indica che l'attributo non può essere modificato (rappresenta, cioè, una costante).

Il tipo può essere un tipo primitivo del linguaggio Java (`int`, `byte`, `float`, `double`, `boolean`, ecc.) oppure una classe o un'interfaccia (per quest'ultima, vedi nel seguito).

Infine, la definizione dei metodi deve aderire alla seguente sintassi:

```
[modificatori] tipo_di_ritorno nome_metodo ([argomenti]) {  
    corpo_del_metodo  
}
```

con gli stessi modificatori di visibilità applicabili agli attributi, oltre ai modificatori `final` e `static` (anche qui, come nel caso degli attributi, indica che non è necessaria l'istanziamento della classe, e quindi definisce una sorta di funzione).

Attraverso i modificatori di visibilità si riescono a realizzare gli obiettivi di **incapsulamento** dei dati e delle implementazioni, facendo sì che all'esterno siano disponibili solo le funzionalità necessarie per interagire con un determinato oggetto, mentre l'effettiva rappresentazione dei dati, le implementazioni ed altre funzionalità (accessorie) sono note solo a chi ha sviluppato la classe.

ESEMPIO: Si supponga allora di voler definire una classe Java per rappresentare un giocatore, ad esempio di una squadra di calcio. Per semplicità, si consideri che il giocatore sia caratterizzato da un nome e dal numero di goal segnati in campionato.

Si prevedono inoltre dei metodi atti a leggere le suddette informazioni, nonché ad aggiornarle (si prevede la possibilità di aggiornare il numero di goal segnati, con il vincolo che non possano diminuire). Il codice della classe potrebbe allora essere il seguente:

```
public class Player
{
    // lo stato del giocatore: nome e numero di reti segnate
    private String name = null;
    private int goals = 0;

    public Player( String n )
    {
        name = n;
    }

    public String getName()
    {
        return name;
    }

    public int getGoals()
    {
        return goals;
    }

    public void incGoals()
    {
        goals++;
    }

    public String toString()
    {
        return new String( name
            + " has scored " + goals + " goals." );
    }
}
```

Altra importante caratteristica della programmazione ad oggetti è il **sovraccaricamento** (o *overloading*) dei metodi, per cui possono esistere più metodi con lo stesso nome che differiscono solo per il numero ed il tipo dei parametri. Questa proprietà potrebbe essere sfruttata per definire un nuovo metodo `incGoals` che consente di incrementare il numero di reti segnati per più di un'unità alla volta:

```
public void incGoals( int g )
{
    goals += g;
}
```

Per verificare il corretto funzionamento della classe `Player`, si può scrivere un semplice programma di test:

```
public class TestPlayer
{
    public static void main( String[] args )
    {
        if ( 2 != args.length )
        {
            System.err.println(
                "java TestPlayer <name> <goals>"
            );
            System.exit( 1 );
        }
        Player p = new Player( args[ 0 ] );
        p.incGoals( Integer.parseInt( args[ 1 ] ) );
        System.out.println( p.toString() );
    }
}
```

Si osservi che questa seconda classe contiene solo il metodo `main`, che è il metodo principale che viene invocato dall'interprete Java quando si richiede l'esecuzione di un programma. Gli argomenti passati a linea di comando sono contenuti nel vettore di stringhe `args`. Dal codice del programma si evince anche la sintassi per istanziare un oggetto attraverso l'operatore `new` e l'assegnazione del valore restituito ad una variabile.

Si precisa inoltre che in Java i tipi primitivi vengono passati per valore, mentre gli oggetti vengono passati per riferimento.

IN PRATICA: Salvando la prima classe in un file dal nome `Player.java` e la seconda in un file dal nome `TestPlayer.java`, dalla cartella in cui si trovano i due files è possibile compilare le classi con il comando:

```
javac -classpath . nome_classe.java
```

e quindi eseguire il programma con il comando

```
java -classpath . TestPlayer "Il Campione" 4
```

Il linguaggio Java, come anche il C++, prevede un meccanismo avanzato per gestire le condizioni di errore: le **eccezioni**. Quando viene generata un'eccezione, l'esecuzione si interrompe ed il controllo ritorna a livelli via via superiori, finché non viene raccolta e gestita. Questo consente la gestione asincrona delle condizioni di errore, ovvero non si richiede che ogni singola condizione di errore sia gestita là dove si verifica, a tutto vantaggio della leggibilità del codice. Per comprendere come usare le eccezioni per segnalare una condizione di errore possiamo correggere il metodo `incGoals(int g)` della classe `Player` come segue:

```
public void incGoals( int g )
    throws IllegalArgumentException
{
    if ( g <= 0 )
        throw new IllegalArgumentException(
            "argument goals must be > 0" );
    else
        goals += g;
}
```

Questo ci costringe a rivedere anche il programma principale, poiché nel caso in cui un metodo prevede la possibilità di lanciare (*throw* è lanciare!) un'eccezione, questa deve essere raccolta:

```

public class TestPlayer
{
    public static void main( String[] args )
    {
        if ( 2 != args.length )
        {
            System.err.println(
                "java TestPlayer <name> <goals>"
            );
            System.exit( 1 );
        }
        Player p = new Player( args[ 0 ] );
        try
        {
            p.incGoals( Integer.parseInt( args[ 1 ] ) );
            /*
                ...
                Qui si potrebbe aggiungere altro codice, che
                verrebbe eseguito solo nel caso in cui non si
                verificasse alcuna eccezione. La cosa
                interessante e' che in questo codice non ci
                dovremmo in alcun modo preoccupare della
                gestione degli errori, che viene differita al
                blocco catch qui sotto.
                ...
            */
            System.out.println( "Nessun errore." );
        }
        catch ( IllegalArgumentException iae )
        {
            System.err.println( "Exception occurred: " +
                iae.toString()
            );
        }
        finally
        {
            System.out.println( p.toString() );
        }
    }
}

```

Si noti anche il blocco try .. catch .. finally ..: l'invocazione di un metodo che può generare un'eccezione deve avvenire all'interno del blocco `try`, e nel corrispondente blocco `catch` ha luogo l'eventuale gestione dell'eccezione; infine, nel blocco `finally` (facoltativo) è contenuto del codice che verrà comunque eseguito, anche nel caso in cui si sia verificata un'eccezione (in genere in questo blocco si rilasciano le risorse eventualmente acquisite).

L'**ereditarietà** è uno dei principali costrutti dei linguaggi orientati agli oggetti, e quindi anche di Java: essa permette di definire nuove classi sulla base di classi esistenti, ereditando da queste attributi e metodi. In particolare, in Java, la classe derivata avrà accesso diretto ad attributi e metodi con visibilità `public` e `protected`, ma non a quelli con visibilità `private`, accessibili alla sola classe entro la quale sono definiti (qui, come nel seguito, non considereremo la visibilità di tipo `package`). La parola chiave Java attraverso la quale si dichiara che una classe discende da un'altra classe è **extends**.

Nel nostro esempio, l'ereditarietà può risultare utile per definire una nuova classe per gestire le particolarità dei portieri: i portieri, come gli altri giocatori, possono segnare goal, ma possono anche parare i tiri in porta... oppure possono non pararli! A questo scopo si definisce una nuova classe `GoalKeeper` che discende da `Player`, ereditandone attributi e metodi, e ve ne aggiunge altri, per considerare le particolarità dei portieri:

```
public class GoalKeeper extends Player
{
    /*  estende lo stato di Player con due nuovi attributi, che
        memorizzano il numero di tiri parati e non parati
    */
    private int saves = 0;
    private int failedSaves = 0;

    public GoalKeeper( String n )
    {
        /*  super(...) richiama il costruttore della classe da
            cui deriva, cioè Player
        */
        super( n );
    }

    //  estende le funzionalita' di Player con 2 nuovi metodi
    public void incSaves()
    {
        saves++;
    }
}
```

```

public void incFailedSaves()
{
    failedSaves++;
}

/* ridefinisce il metodo, già' definito in Player;
   si appoggia comunque alla precedente definizione, che
   richiama con super.toString()
*/
public String toString()
{
    return new String (
        super.toString() +
        System.getProperty( "line.separator" ) +
        "He succeeded in " + saves +
        " saves, and failed in " + failedSaves + "."
    );
}
}

```

Si noti che, oltre all'aggiunta di nuovi attributi e metodi, è stata ridefinito il metodo `toString()`, precedentemente definito in `Player` (in realtà il metodo è definito per ogni classe che discende da `Object`, ed in Java tutte le classi discendono, seppur implicitamente, da `Object`; per cui, anche nel caso di `Player`, si trattava in realtà di una **ridefinizione**). La possibilità di ridefinire i metodi nelle classi derivate è un'altra significativa caratteristica dei linguaggi ad oggetti, e viene solitamente indicata con il termine inglese *overriding*.

Anche questa volta, per verificare il corretto funzionamento della classe, è possibile introdurre una classe di test, come la seguente:

```

public class TestGoalKeeper
{
    public static void main( String[] args )
    {
        if ( 1 != args.length )
        {
            System.err.println(
                "\tjava TestGoalKeeper <name>" );
            System.exit( 1 );
        }
        GoalKeeper gk = new GoalKeeper( args[ 0 ] );
        gk.incGoals();
        gk.incSaves();
    }
}

```

```
gk.incSaves();  
gk.incFailedSaves();  
System.out.println( gk.toString() );  
}  
}
```

Nell'esempio compare la parola chiave `super`, che viene utilizzata per far riferimento ad attributi, costruttori e metodi della superclasse. Di frequente utilizzo è anche la parola chiave `this`, che viene utilizzata da un oggetto per indicare un riferimento a se stesso.

Si precisa che in Java non è prevista l'ereditarietà multipla, e quindi una classe può discendere al più da un'altra classe. Per quanto questo vincolo possa sembrare una limitazione (rispetto, ad esempio, al C++), esso impedisce il manifestarsi dei tanti problemi associati all'ereditarietà multipla.

Nell'ambito di questa trattazione sull'ereditarietà è utile segnalare che per la definizione dei metodi è previsto anche il modificatore `abstract`, con il quale si indica che per il metodo in questione viene definita la firma (o *signature*, ovvero il nome e l'insieme dei parametri in ingresso e in uscita), ma non il corpo, ovvero l'implementazione. Se una classe contiene uno o più metodi astratti, anche la classe dovrà essere dichiarata astratta, e non può essere istanziata. L'utilità delle classi astratte deriva dal fatto che attraverso il metodo astratto esse prescrivono una funzionalità che tutte le sue classi derivate dovranno implementare per poter essere istanziate.

Il linguaggio Java prevede anche un'altro utile costrutto, l'**interfaccia**¹ (`interface`): un'interfaccia specifica un insieme di metodi senza peraltro fornirne un'implementazione. Una classe Java può implementare un'interfaccia fornendo un'implementazione per i metodi specificati da quest'ultima. Si osservi che in Java

¹ Un'interfaccia Java è assimilabile ad una classe C++ che ha tutti i metodi astratti e nessun attributo.

una singola classe può implementare più interfacce, riducendo così i disagi derivanti dall'assenza dell'ereditarietà multipla.

Ad esempio, l'API di Java (package `java.util`) prevede un'interfaccia per definire le funzionalità di una generica collezione di oggetti:

```
public interface Collection {
    boolean add(Object o)
    void clear()
    boolean contains(Object o)
    boolean isEmpty()
    Iterator iterator()
    boolean remove(Object o)
    int size()
    Object[] toArray()
    ...
}
```

Anche il tipo `Iterator` restituito da uno dei metodi è un'interfaccia:

```
public interface Iterator {
    boolean hasNext()
    Object next()
    void remove()
}
```

e definisce i metodi per iterare su una collezione di oggetti.

Dell'interfaccia `Collection` esistono numerose implementazioni, che differiscono per l'organizzazione interna della struttura dati che contiene gli oggetti: `LinkedList`, `Stack`, `Vector`, ...

Le diverse implementazioni possono fornire anche funzionalità aggiuntive (es. accesso diretto ad un elemento nel caso di un vettore), ma sicuramente prevedono quelle prescritte dall'interfaccia `Collection`. Considerato che la loro organizzazione interna può differire notevolmente, si comprende anche come mai per l'iterazione non si specifichi una classe ben precisa, ma piuttosto un'interfaccia: ciascuna collezione restituirà tramite il metodo `iterator()` un'istanza di una particolare classe, in grado di percorrere la specifica struttura dati.

Vediamo concretamente l'uso di collezioni e iteratori nel caso del nostro esempio. Per raccogliere tutti i giocatori di una squadra possiamo definire una nuova classe `Team`. Tuttavia, per gestire l'insieme dei giocatori, ci appoggeremo alle API di Java che, nel package `java.util`, forniscono un insieme di classi adatte alla creazione e manipolazione di collezioni di oggetti. La classe `Team` dovrà soddisfare almeno due requisiti: da un lato dovrà consentire il popolamento della squadra, e dall'altro dovrà restituire la somma dei goal segnati dai suoi giocatori. Una possibile implementazione per questa classe è la seguente:

```
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;

public class Team
{
    private Collection players = null;
    private String name = null;

    public Team( String n )
    {
        name = n;
        players = new LinkedList();
    }

    public void addPlayer( Player p )
    {
        players.add( p );
    }

    public int getGoals()
    {
        int goals = 0;
        for( Iterator i = players.iterator(); i.hasNext(); )
        {
            Player p = (Player) i.next();
            goals += p.getGoals();
        }
        return goals;
    }

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        String new_line = System.getProperty( "line.separator" );
    }
}
```

```
sb.append( "Composition of team \"" + name +
          "\":" + new_line
);
for( Iterator i = players.iterator(); i.hasNext(); )
{
    Player p = (Player) i.next();
    sb.append( p.toString() + new_line );
}
return sb.toString();
}
}
```

Nel programma riportato sopra, la classe `LinkedList` implementa l'interfaccia `Collection`, e pertanto una sua istanza può essere assegnata ad una variabile di tipo `Collection` (anche questa è una caratteristica dei linguaggi OO:

polimorfismo, o *dynamic binding*). Per una descrizione di `Collection` e `LinkedList`, così come della classe `StringBuffer` (anch'essa contenuta in `java.util`) si faccia riferimento alla documentazione delle API Java.

Qui di seguito è riportato un programma (`TestTeam`) che, facendo uso delle classi precedentemente definite, crea un'istanza di una squadra e la popola con un certo numero di istanze di giocatori (giocatori "qualunque" e portieri):

```
public class TestTeam
{
    public static void main( String[] args )
    {
        Team t = new Team( "Squadrone" );
        try
        {
            Player p = new GoalKeeper( "Johnny Portiere" );
            p.incGoals( 1 );
            t.addPlayer( p );
            p = new Player( "Gino Difensore" );
            p.incGoals( 3 );
            t.addPlayer( p );
            p = new Player( "Marc Attaccante" );
            p.incGoals( 8 );
            t.addPlayer( p );
        }
        catch ( IllegalArgumentException iae )
    }
}
```

```
{
    System.err.println(
        "An illegal argument was passed to a method"
    );
}
System.out.println( t.toString() );
System.out.println(
    "The team has scored " + t.getGoals() +
    " goals."
);
}
}
```

Si osservi come, nel corpo del programma, alla variabile `p`, di tipo `Player`, vengano indistintamente assegnati oggetti che sono istanze sia della classe `Player`, sia della classe `GoalKeeper`; questo è assolutamente lecito, poiché in virtù dell'ereditarietà una classe derivata può essere sempre usata in luogo di una classe da cui discende (direttamente o indirettamente).