

# Java

## Programmazione orientata agli oggetti

Nella programmazione orientata agli oggetti (OOP), un'applicazione consiste di una collezione di oggetti interagenti, ognuno dei quali comprende una struttura dati e un'insieme di operazioni che possono esserle applicate. Gli oggetti aiutano il programmatore a gestire la complessità dei sistemi software, migliorandone la comprensibilità, la riutilizzabilità e la robustezza.

La *comprensibilità* è migliorata poiché ogni oggetto si riferisce ad un'entità che ha un preciso significato nel dominio dei problemi affrontati o delle soluzioni che si cercano; inoltre, gli oggetti interagiscono fra loro mediante meccanismi ben definiti.

La *riutilizzabilità* deriva direttamente dalla comprensibilità: se un oggetto è ben comprensibile, potrà essere utilizzato in applicazioni differenti, grazie al meccanismo dell'ereditarietà.

La *robustezza* riguarda l'architettura del programma e la possibilità di evitare errori in fase di esecuzione. Poiché gli oggetti utilizzano tecniche di incapsulamento per nascondere all'utente alcune loro parti, un programma orientato agli oggetti è di norma meno soggetto ad errori di un programma modulare.

Le principali caratteristiche della OOP sono Identità, Classificazione, Ereditarietà, Polimorfismo e Incapsulamento.

### ***Identità:***

Ogni oggetto è unico, perciò due oggetti sono distinti anche se i loro attributi sono identici.

### ***Classificazione:***

Ogni oggetto è istanza di una "classe", che rappresenta un insieme di oggetti aventi le stesse proprietà.

### ***Ereditarietà:***

Mediante questo potente meccanismo, il programmatore può costruire nuove classi estendendo quelle esistenti. Ogni nuova classe può derivare da una sola classe esistente (ereditarietà singola) o da più classi (ereditarietà multipla). Le classi derivate sono anche chiamate "sottoclassi".

***Polimorfismo e "dynamic binding":***

Se una variabile O contiene un oggetto che appartiene a una classe C, si dice che O è staticamente legato a C (static binding). Se C ha sottoclassi e se durante l'esecuzione del programma viene assegnato ad O un puntatore ad un oggetto che appartiene a una sottoclasse D di C, si dice che O è dinamicamente legato a D (dynamic binding). Quando un oggetto variabile viene dinamicamente legato a classi differenti, in sede di esecuzione del programma, si dice che è polimorfo.

***Incapsulamento:***

Ogni oggetto possiede un "interfaccia", costituita da variabili e funzioni a cui l'utente può accedere liberamente, e da una "struttura interna", che contiene dati e funzioni inaccessibili all'utente. L'incapsulamento è anche detto "Information Hiding".

Per definizione, un linguaggio è:

1. object-based, se ha specifici costrutti per la manipolazione di oggetti;
2. class-based, se è object-based e ogni oggetto appartiene ad una classe (in tal caso si dice anche che il linguaggio supporta l'"astrazione dei dati");
3. object-oriented se è class-based e le classi possono condividere caratteristiche attraverso l'ereditarietà;
4. strongly-typed object-oriented, se è sia object-oriented che strongly-typed, cioè se ogni nuova classe costituisce un nuovo tipo di dato.

Esempi di linguaggi strongly-typed object-oriented sono Eiffel (Meyer, 1988), C++ (Stroustrup, 1986), e Java.

## **Il linguaggio Java**

Java deriva da un progetto di ricerca dalla Sun Microsystems teso allo sviluppo di un ambiente operativo snello, affidabile, portabile, distribuito e in tempo reale, dedicato ai servizi di rete. Inizialmente, il linguaggio adottato fu il C++, ma i numerosi problemi incontrati condussero allo sviluppo di un linguaggio completamente nuovo, che tuttavia mantiene molte somiglianze col C++, e con altri linguaggi come SmallTalk ed Eiffel.

Di seguito sono elencate le caratteristiche fondamentali di Java.

***Familiarità e semplicità:***

Java riprende molte caratteristiche del C e del C++, il che rende questo linguaggio estremamente familiare alla maggioranza dei programmatori, che possono cominciare a programmare in Java dopo un periodo di apprendimento relativamente breve. Inoltre,

nella creazione di Java, la Sun ha eliminato tutte quelle caratteristiche del C++ considerate di dubbia utilità, rendendo Java il più semplice possibile.

***Programmazione orientata agli oggetti:***

Java offre un supporto completo alla OOP, tramite incapsulamento, polimorfismo, ereditarietà e dynamic binding.

Si badi che, nelle specifiche della SUN, si intende per "polimorfismo" il fatto che uno stesso "segnale" generico inviato a differenti oggetti dà risultati differenti a seconda del tipo di ogni oggetto, e si intende per "dynamic binding" che un programma Java può richiamare dinamicamente definizioni di classi durante l'esecuzione.

***Neutralità:***

Il compilatore Java non genera codice macchina, nel senso di istruzioni dedicate ad una specifica piattaforma hardware, ma un codice di più alto livello, indipendente dall'hardware, chiamato bytecode, in grado di essere interpretato ed eseguito dal sistema runtime di Java su un qualsiasi calcolatore. I vantaggi di questo approccio sono evidenti in un'architettura di rete distribuita, come Internet, a cui sono collegati calcolatori strutturalmente diversi come PC, Macintosh e workstation Unix.

***Portabilità:***

La portabilità del codice Java è garantita dalla chiara definizione di tutti i tipi di dati primitivi, in termini di precisione e dell'aritmetica applicata. Ad esempio un tipo int è un intero in complemento a due su 32 bit, su qualunque sistema in cui sia stato implementato l'interprete e il sistema runtime di Java. La sintassi del linguaggio non ammette regole che siano "dipendenti dall'implementazione". Inoltre il sistema Java stesso è completamente portabile: il compilatore Java è scritto in Java, e il sistema di runtime è scritto in ANSI C.

***Robustezza:***

Java garantisce che i programmi siano robusti, affidabili e sicuri, in una quantità di modi diversi. La sintassi del linguaggio richiede che tutte le dichiarazioni siano esplicite, e non ammette dichiarazioni implicite in stile C/C++. Essendo Java un linguaggio "strongly typed", il compilatore può eseguire meticolose e stringenti verifiche di correttezza sul codice, segnalando molti tipi diversi di errori prima ancora che il programma venga eseguito. Numerosi controlli sono anche eseguiti in fase di esecuzione, al fine di garantire la consistenza e la flessibilità del codice. Inoltre, il gestore della memoria di Java elimina la possibilità di sovrascritture dei dati, grazie anche al fatto che l'aritmetica dei puntatori non fa parte delle specifiche del linguaggio.

## Java: interpretato e dinamico

Il compilatore Java produce codice in forma di bytecode, che viene interpretato ed eseguito sulla Java Virtual Machine (JVM), inclusa nel sistema runtime di Java: in questo modo, i programmi Java possono essere eseguiti su un qualsiasi calcolatore per cui la JVM sia stata resa disponibile.

La compilazione non prevede una fase di "linking" di librerie: questo lavoro, che di fatto consiste nel caricamento di nuove classi, viene svolto dal Class Loader, con un processo dinamico ed incrementale: le classi vengono caricate quando sono necessarie, e possono provenire dall'ambiente locale o dalla rete; prima di essere passato all'interprete per essere eseguito, il nuovo codice viene controllato per verificarne la correttezza.

Il sistema di compilazione di Java risolve uno dei più annosi problemi del C++, consistente nel fatto che ogni volta che il codice sorgente di una classe viene alterato, tutte le sottoclassi devono essere ricomilate, in un procedimento a catena. Il compilatore Java, invece, non trasforma i riferimenti alle classi in valori numerici, e passa all'interprete i riferimenti in forma simbolica, che vengono risolti solo in fase di esecuzione del codice, dinamicamente. In questo modo, nuove versioni di classi possono essere caricate senza che il programma che le utilizza debba essere ricompilato.

Anche l'allocazione degli oggetti è dinamica: il compilatore non decide quale spazio di memoria vada assegnato ad ogni oggetto, demandando questo compito al sistema runtime.

## Sicurezza

La sicurezza è un fattore importante in un ambiente di rete, e Java erige diversi strati di protezione contro codice potenzialmente dannoso, attraverso il compilatore, l'interprete e il sistema runtime.

**Allocazione della memoria:** la prima linea di difesa sta nel sistema di allocazione della memoria, che è appannaggio del sistema runtime, e non del compilatore, cosicché viene a dipendere dalle caratteristiche hardware e software della piattaforma su cui viene eseguito Java. Inoltre, Java non possiede i puntatori, nel senso di celle di memoria che contengono gli indirizzi di altre celle: i riferimenti sono gestiti in modo simbolico, e trasformati in veri indirizzi di memoria solo durante l'esecuzione.

Queste caratteristiche non devono essere viste come una limitazione per il programmatore, ma come un metodo per garantire una maggiore robustezza e affidabilità delle applicazioni.

*Verifica del bytecode*: il codice Java proveniente dalla rete potrebbe essere stato generato da un compilatore che non soddisfa le regole di sicurezza di cui al punto precedente, e quindi potrebbe non essere affidabile. Di conseguenza, prima di procedere all'esecuzione del codice, il sistema runtime ne effettua un'analisi al fine di verificare che:

1. non crei puntatori;
2. non violi le restrizioni di accesso;
3. acceda agli oggetti per quello che sono, senza fare cast illegali .

Il caricatore e il verificatore del bytecode non fanno differenza fra codice locale e codice scaricato dalla rete: se il bytecode arriva all'interprete per essere eseguito, significa che è stato controllato e che è sicuro.

Naturalmente, il processo di verifica interpone un ostacolo alla rapidità tra la fase di caricamento e la fase di esecuzione del codice, ma allo stesso tempo permette di ottenere un certo numero di informazioni sul bytecode, consentendo all'interprete di lavorare più velocemente, senza dover fare ulteriori verifiche sull'affidabilità di ciò che esegue.

*Caricamento del bytecode*: durante la sua esecuzione, un programma Java può richiedere il caricamento di una o più classi o interfacce: dopo che il nuovo codice è stato controllato dal *Verificatore*, il *Caricatore del bytecode* costituisce una seconda linea di difesa.

All'insieme di classi locali viene assegnato un certo spazio dei nomi, ed un altro è assegnato ad ogni sorgente di classi che provengono dalla rete. Ogni volta che una classe fa riferimento ad un'altra classe, questa viene cercata prima nello spazio dei nomi locali, e poi nello spazio dei nomi esterni. Non c'è possibilità, per una classe esterna, di aggirare questo sistema e sovrapporsi alle classi locali, né a classi esterne che provengano da una sorgente diversa.

*Networking Package*: il Networking Package di Java consente di gestire vari protocolli di rete, come ad esempio Telnet ed FTP, e può essere configurato per:

- impedire l'accesso a qualsiasi host;
- permettere solo l'accesso agli host da cui sia stato importato codice Java;

- permettere solo l'accesso agli host oltre il Firewall da cui sia stato importato codice Java;
- permettere l'accesso a qualsiasi host.

## Multithreading

Un *thread* è un singolo flusso sequenziale all'interno di un programma; in Java, più thread possono essere eseguiti "contemporaneamente"; di fatto l'esecuzione di un programma è sempre sequenziale: è il runtime di Java che si occupa di saltare da un thread all'altro, simulando la contemporaneità dei vari processi. Il metodo usato è il "time-slicing", che assegna ad ogni thread una certa percentuale del tempo-macchina a disposizione. Tuttavia, non tutti i sistemi supportano il time-slicing, e perciò ad ogni thread viene assegnata una certa priorità: un thread a priorità superiore può soppiantare i thread a priorità inferiore, conquistando il controllo della CPU.

Java supporta il multithreading a livello di sintassi del linguaggio, e tramite il sistema runtime. A livello di linguaggio, un thread è un oggetto della classe standard *Thread*, o di una sua sottoclasse, e possiede i metodi necessari a controllarne l'avvio, l'esecuzione, l'interruzione e lo stato. I metodi che sono dichiarati con la parola chiave *synchronized* non sono soggetti ad esecuzione concorrente, e cadono sotto il controllo dei *monitor*, che assicurano la consistenza dei valori delle variabili all'interno del metodo. Ogni classe ed ogni oggetto istanziato ha il suo proprio monitor, che viene attivato quando è necessario. I monitor di Java sono "re-entrant", il che significa che un metodo può acquisire lo stesso monitor più di una volta, in istanti successivi.

I metodi sincronizzati consentono di scrivere applicazioni "thread-safe", le cui funzioni, cioè, possono essere richiamate da più thread concorrenti, ma, fra questi, solo uno alla volta può accedere alle variabili di istanza.

## Specifiche del linguaggio

**Tipi di dato:** ogni variabile ed ogni espressione possiede un "tipo", che determina l'intervallo dei valori che una variabile può contenere, le operazioni possibili su questi valori ed il significato delle operazioni. Java possiede due diversi tipi: semplici e composti. Sono semplici quei tipi che non possono essere ulteriormente decomposti in

altri tipi: gli interi (*int*), i numeri in virgola mobile (*float*) e i caratteri (*char*) sono tutti esempi di tipi semplici. I tipi composti sono vettori, classi ed interfacce.

### *Tipi numerici:*

- *Tipi interi*

Gli interi hanno una precisione indipendente dalla macchina su cui viene eseguito Java, e possiedono sempre segno. Essi sono:

- byte (8 bit);
- short (16 bit);
- int (32 bit);
- long (64 bit).

- *Tipi in virgola mobile*

La parola chiave *float* indica un numero in virgola mobile a singola precisione (32 bit), mentre *double* indica doppia precisione (64 bit). Il formato e l'aritmetica floating point seguono le specifiche IEEE 754.

**Tipi carattere:** Java utilizza il set di caratteri Unicode, e quindi un *char* è un intero senza segno su 16 bit.

**Tipi booleani:** il tipo *boolean* è utilizzato per variabili che possono essere "vere" o "false", e per metodi che possono restituire i valori vero e falso. È anche il tipo restituito da espressioni che facciano uso di operatori relazionali, come  $<$  o  $\geq$ .

I valori booleani non sono numeri e non possono essere convertiti in numeri mediante casting.

**Vettori:** i vettori (o arrays) devono essere creati utilizzando l'operatore *new*. Ad esempio, l'istruzione:

```
char x[]=new int[10];
```

crea un vettore di interi costituito da 10 elementi, numerati da 0 a 9. La dimensione di un vettore non può essere specificata nella dichiarazione (*int x[]*), ma solo tramite l'operatore *new*.

Il linguaggio non supporta l'uso di matrici a più dimensioni, ma è possibile creare vettori di vettori, del tipo:

```
int x[][]=new int[10][15];
```

**Classi:** le classi consentono l'utilizzo dell'OOP. In Java, la creazione di una nuova classe implica la creazione di un nuovo tipo di dato.

Ogni classe è definita nel modo seguente:

```
[modificatori] class Nome_Classe
[extends Nome_Classe_di_Base]
[implements interface {Nome_Interfaccia} ]
{
    // Corpo_della_classe
}
```

Ad esempio:

// Una classe punto

```
public class Point
```

```
{
    float x,y;
    // ...
}
```

// Un punto stampabile

```
class PrintablePoint extends Point implements Printable
```

```
{
    public void print() { /* ... */ }
    // ...
}
```

Tutte le classi derivano automaticamente dalla classe di base generica *Object*. Se una classe è dichiarata senza specificare esplicitamente una classe di base, si assume che questa sia *Object*. Ad esempio:

```
class Point
```

```
{
    float x,y;
}
```

equivale a

```
class Point extends Object
```

```
{
    float x,y;
}
```

Il linguaggio supporta solo l'ereditarietà singola, ma alcune delle caratteristiche dell'ereditarietà multipla si possono ottenere grazie all'utilizzo delle interfacce.

**Casting fra le classi:** il casting fra le classi è possibile, poiché ogni nuova classe è un tipo, e Java supporta il casting fra i tipi. Se B è una sottoclasse di A, ogni istanza di B può essere usata come se fosse un'istanza di A, senza bisogno di casting esplicito: questo è chiamato "ampliamento" (*widening*). Se invece un'istanza di A deve essere usata come se fosse un'istanza di B, il programmatore deve definire un tipo di conversione, o cast: questo è chiamato "riduzione" (*narrowing*). Il cast da una classe ad una sottoclasse viene sempre controllato in runtime, per verificarne la correttezza. È invece vietato il cast fra classi "sorelle", ovvero fra classi che siano derivate da una stessa classe di base. La sintassi del cast è:

```
(nome_classe)nome_oggetto
```

Si badi che questa operazione coinvolge solo il riferimento all'oggetto, e non l'oggetto stesso. Tuttavia, l'accesso alle variabili di istanza è determinato dal tipo del riferimento, come nell'esempio seguente:

```
class ClassA
{
    String name="ClassA";
}

class ClassB extends ClassA
{
    String name="ClassB";
}

//...

void test()
{
    ClassB b=new ClassB();
    println(b.name);    // Stampa "ClassB"
    ClassA a;
    a=(ClassA)b;       // Cast
    println(a.name);   // Stampa "ClassA"
}
```

**Metodi:** i metodi sono le funzioni membro di una classe o di un'interfaccia. Ogni metodo ha un tipo di ritorno, a meno che non sia un costruttore. Se un metodo non restituisce alcun risultato, esso va dichiarato come void.

Ogni metodo ha una lista di parametri consistente in coppie di tipi e nomi di parametri, separate da virgole. Se il metodo non utilizza parametri, la lista dovrebbe essere vuota. Le variabili dichiarate nei metodi, dette variabili locali, devono avere nomi diversi una dall'altra e diversi dai nomi dei parametri.

Java permette l'uso di metodi polimorfici, cioè permette di dichiarare all'interno di una stessa classe metodi con lo stesso nome ma con una diversa lista di parametri (overloading), e permette di dichiarare in una classe derivata metodi che abbiano lo stesso nome di altri dichiarati nella classe di base (overriding).

**Le variabili *this* e *super*:** nello spazio di visibilità di un metodo non statico, la variabile *this* contiene un riferimento all'oggetto corrente, e il suo tipo è la classe che contiene il metodo in esecuzione. La variabile *super* contiene un riferimento alla classe di base della classe corrente.

**Variabili di istanza:** tutte le variabili di una classe che sono definite fuori dal corpo di un metodo, e che non sono marcate come *static*, sono dette variabili di istanza. Se una variabile di istanza non possiede un inizializzatore, essa viene inizializzata automaticamente secondo le seguenti regole:

- le variabili numeriche sono inizializzate a zero;
- le variabili booleane sono inizializzate a falso;
- gli oggetti sono inizializzati a *null*.

**Overriding:** quando una sottoclasse dichiara un metodo che ha lo stesso nome, lo stesso tipo di ritorno e la stessa lista di parametri di un metodo della classe di base, si dice che quest'ultimo è sottoposto ad *overriding*: quando il metodo è invocato su un'istanza della sottoclasse, il metodo della sottoclasse è richiamato al posto di quello della superclasse. Al metodo *overridden* si può accedere utilizzando la variabile *super*, ad esempio:

```
setValues(...) // si riferisce al metodo della classe corrente  
super.setValues(...) // si riferisce al metodo della classe di base
```

**Overloading:** quando una sottoclasse dichiara un metodo che ha lo stesso nome, ma un diverso tipo di ritorno e/o una diversa lista di parametri di un metodo della classe di base, si dice che quest'ultimo è sottoposto ad *overloading*. La risoluzione dell'*overload* è la scelta fra quale metodo deve essere eseguito in un certo istante, fra tutti quelli che hanno lo stesso nome: essa viene eseguita dal compilatore sulla base della lista degli argomenti dei metodi, ignorando il tipo di ritorno, e considerando quale, fra i metodi considerati, ha il minor costo di conversione dei parametri.

Il costo di ogni metodo è il massimo costo di conversione di ognuno dei suoi argomenti, che possono essere oggetti o tipi primitivi.

Nel caso di tipi primitivi, si consulta una tabella apposita. Nel caso di conversione fra oggetti, il costo è pari al numero di collegamenti nell'albero delle classi tra la classe del

parametro fornito come argomento al metodo, e la classe dichiarata nella lista degli argomenti del metodo, considerando solo conversioni di ampliamento.

Nel caso due metodi abbiano lo stesso costo, non c'è possibilità di scelta per il compilatore, che genera un errore. Ad esempio:

```
class A
{
    int method(Object o,Thread t);
    int method(Thread T, Object o);
    void f(Object o,Thread t)
    {
        method(o,t);    // Invoca il primo metodo
        method(t,o);    // Invoca il secondo metodo
        method(t,t);    // Ambiguo: errore di compilazione
        method(o,o);    // Errore: non c'è un metodo corrispondente
    }
}
```

**Costruttori:** i costruttori sono speciali metodi utilizzati per l'inizializzazione degli oggetti, e si distinguono dagli altri metodi per il fatto di avere lo stesso nome della classe e per non avere alcun tipo di ritorno. I costruttori vengono chiamati automaticamente quando un oggetto viene creato.

**Creazione degli oggetti:** la dichiarazione di un oggetto non implica che all'oggetto sia assegnata un'area di memoria. Questo viene fatto solo dopo una chiamata esplicita dell'operatore `new`, che in aggiunta inizializza le variabili di istanza della classe, e richiama il costruttore.

**Garbage collection:** il sistema di gestione della memoria di tipo *garbage collection* affranca i programmatori dall'obbligo di distruggere esplicitamente oggetti non più utilizzati durante l'esecuzione del programma: di questo si occupa automaticamente il sistema, che libera i segmenti di memoria a cui non si fa più riferimento.

**Metodi, variabili e inizializzatori statici:** variabili e metodi dichiarati come statici, utilizzando la parola chiave *static*, vengono applicati non a una singola istanza della classe, ma alla classe stessa, ovvero a tutte le sue istanze. Inoltre, una porzione di codice può essere dichiarata statica all'interno della definizione della classe, e viene chiamata "inizializzatore statico".

Una variabile statica può possedere inicializzatori, come una variabile di istanza, ed esiste una e una sola volta per ogni classe, indipendentemente da quante istanze della classe esistano.

Si può accedere ai metodi e alle variabili statiche sia utilizzando il nome della classe, sia utilizzando il nome di un'istanza della classe.

**Ordine di dichiarazione e di inicializzazione:** l'ordine di dichiarazione delle classi, e dei metodi e delle variabili di istanza all'interno delle classi, è irrilevante. Ogni metodo è libero di fare riferimento "in avanti" ad altri metodi e variabili non ancora dichiarati.

L'inicializzazione, invece, avviene in ordine lessicale quando una classe viene caricata. Se un'istruzione di inicializzazione statica fa riferimento ad una classe non ancora caricata, questa viene caricata e il suo codice di inicializzazione statica viene eseguito. Se durante la sequenza di inicializzazione si fa riferimento ad una classe non ancora inicializzata e precedente nella sequenza, si ha la creazione di un ciclo, che causa il lancio di un'eccezione.

La dipendenza in avanti per inicializzazioni di variabili statiche è proibita. Il seguente codice:

```
static int y=x+1;  
static int x=10;
```

causa un errore durante la compilazione del programma.

Una inicializzazione di una variabile di istanza può avere un'apparente dipendenza in avanti da una variabile statica, ad esempio:

```
int y=x+1;  
static int x=10;
```

In realtà la dipendenza non sussiste, poiché la variabile *x*, essendo statica, viene inicializzata prima della variabile *y*.

**Specificatori di accesso:** gli specificatori di accesso sono modificatori che consentono ai programmatori di controllare l'accesso a metodi e variabili.

I metodi e le variabili dichiarati pubblici (*public*) sono accessibili da qualunque classe o metodo da qualsiasi punto del programma. I metodi e le variabili dichiarati privati (*private*) sono *accessibili* solo dall'interno della classe in cui sono dichiarati. Poiché i metodi privati non sono visibili al di fuori della classe, essi sono finali e non possono essere sottoposti ad overriding. Inoltre, non si può fare overriding su un metodo non privato per renderlo privato. I metodi e le variabili dichiarati protetti (*protected*) sono accessibili dalle classi derivate, ma non da altre classi.

Classi, metodi e variabili che non hanno uno specificatore di accesso sono accessibili solo all'interno del package in cui sono dichiarate.

**Regole di visibilità delle variabili:** all'interno di un package, quando una classe è definita come derivata, le dichiarazioni fatte nella classe di base sono visibili nella sottoclasse. Quando si fa riferimento ad una variabile entro il corpo di un metodo, si usa la seguente regola: si ricerca prima all'interno del blocco corrente, poi in tutti i blocchi che lo includono, fino al blocco principale del metodo. Questo è considerato lo "spazio di visibilità" (*scope*) locale della variabile.

Se necessario, la ricerca continua nello scope della classe. Anzitutto si ricerca fra le variabili della classe corrente. Se la variabile cercata non è stata trovata, si cerca fra le variabili di tutte le classi di base, procedendo nella gerarchia delle classi dal basso verso l'alto, finché non si raggiunge la classe *Object*. Se la variabile non è stata trovata, si ricerca fra le classi importate e fra i package. Se anche questa ricerca fallisce, viene generato un errore di compilazione.

Non è consentito avere più variabili con lo stesso nome all'interno della stessa classe.

#### **Modificatori:**

- Classi, metodi e variabili *final*

Una classe *final* non può avere sottoclassi, un metodo *final* non può essere sottoposto ad *overriding*, una variabile *final* si comporta come una costante.

- Metodi nativi

I metodi dichiarati nativi (*native*) vengono implementati in un linguaggio dipendente dal sistema che si usa, come ad esempio il C, e non in Java. Essi non hanno un corpo, e la dichiarazione termina con un carattere ";". I costruttori non possono essere nativi.

- Metodi astratti

Un metodo dichiarato astratto (*abstract*) non possiede un corpo, e deve essere definito in una sottoclasse della classe in cui è stato dichiarato. Classi che contengano metodi astratti, o che ereditino metodi astratti senza definirli, sono considerate classi astratte, e come tali non possono essere istanziate.

- Metodi e blocchi sincronizzati

Un metodo o un blocco di codice marcato con la parola chiave *synchronized* non può essere eseguito contemporaneamente ad altri segmenti di codice che

accedano alle medesime risorse, e si dice che possiede una *lock*: ogni oggetto ed ogni classe sono associati ad una e una sola *lock*. Ad esempio:

```
class Point
{
    float x,y;
    synchronized void scale(float f)
    {
        x*=f;
        y*=f;
    }
}
```

Quando un metodo sincronizzato viene invocato, esso aspetta fino a quando non può acquisire la lock per una certa istanza (o classe, se è un metodo statico), quindi esegue il suo codice e alla fine rilascia la lock.

I blocchi di codice sincronizzati si comportano in modo simile, ma la classe o l'oggetto su cui si richiede una lock deve essere dichiarato esplicitamente, ad esempio:

```
class Rectangle
{
    Point topLeft;
    // ...
    void print()
    {
        synchronized (topLeft)
        {
            println("topLeft.x = " + topLeft.x);
            println("topLeft.y = " + topLeft.y);
        }
        // ...
    }
}
```

**Interfacce:** un'interfaccia specifica una collezione di metodi senza che ne venga fornita un'implementazione, al fine di permettere l'inclusione di un certo protocollo di metodi in una classe, senza restringere l'implementazione all'ereditarietà singola. Quando una classe implementa un'interfaccia, deve definirne i corpi di tutti i metodi, a meno che non sia una classe astratta.

Le interfacce consentono di superare alcune delle limitazioni dell'ereditarietà singola, e consentono a più classi di condividere lo stesso tipo di programmazione, senza che ogni classe debba preoccuparsi delle implementazioni delle altre. Tuttavia, poiché le interfacce coinvolgono il dynamic binding, c'è una certa diminuzione delle prestazioni nei programmi che ne fanno uso.

L'esempio seguente mostra la dichiarazione di un'interfaccia e di una classe che la implementa:

```
public interface Storing
{
    // ...
    void reconstitute(Stream s);
}

public class Image implements Storing
{
    // ...
    void reconstitute(Stream s)
    { /* ... */ }
}
```

Come le classi, le interfacce possono essere pubbliche o private, e seguono le stesse regole di scope. I metodi di un'interfaccia sono sempre pubblici; le variabili possono essere pubbliche, statiche e *final*.

**Interfacce come tipi:** la dichiarazione "nomeInterfaccia nomeVariabile" indica che la variabile è istanza di qualche classe che implementa l'interfaccia nomeInterfaccia. Quando utilizzate come tipi, le interfacce si comportano esattamente come le classi: ciò consente al programmatore di specificare che un oggetto deve implementare una data interfaccia, senza conoscere l'esatto tipo o l'ereditarietà dell'oggetto. Ad esempio:

```
class StorageManager
{
```

```
Stream s;
// ...
void pickle(Storing obj) { obj.freeze(s); }
}
```

**Combinazione di interfacce:** un'interfaccia può incorporare una o più altre interfacce, mediante l'uso della parola chiave *extends*, come in questo esempio:

```
interface DoesItAll extends Storing, Painting
{
    void doesSomethingElse();
}
```

**Package:** i *package* sono gruppi di classi e interfacce, e ogni classe o interfaccia appartiene ad un package. Per convenzione, i nomi dei package consistono di parole separate da punti, dove la prima parola rappresenta l'organizzazione che ha sviluppato il package.

Il package a cui appartiene una certa unità di compilazione è specificato utilizzando la parola chiave *package*. Quando questa istruzione è presente, essa deve essere la prima linea effettiva di codice, escludendo le righe vuote ed i commenti. Il formato è `package nome_package;`

Quando un'unità di compilazione non ha un'istruzione del genere, essa viene posta nel package predefinito, che non ha nome.

**Uso di classi ed interfacce da altri package:** la parola chiave *import* è utilizzata per importare classi ed interfacce da un dato package a quello corrente, cosicché i loro nomi siano visibili nello spazio dei nomi corrente. Importare un package significa rendere disponibili tutte le sue classi ed interfacce pubbliche. Esempio:

```
// Importa tutte le classi da acme.project
import acme.project.*;
// Importa solo la classe Employee_list da acme.project
import acme.project.Employee_list;
```

In alternativa, si può accedere a classi o interfacce di un package indicando esplicitamente il nome del package a cui la classe o interfaccia appartiene, ad esempio:

```
acme.project.Employee_List list=new acme.project.Employee_List();
```

**Eccezioni:** quando si ha un errore durante l'esecuzione di un programma, il codice che lo rileva può lanciare un'eccezione. Per definizione, le eccezioni causano la stampa di

un messaggio di errore e la terminazione dei thread, ma i programmi possono catturare le eccezioni e recuperare gli errori.

Alcune eccezioni vengono lanciate dal sistema di runtime di Java, ma ogni classe può definire le sue eccezioni, e lanciarle mediante un'istruzione che consiste della parola chiave *throw* seguita dal nome di un oggetto, che deve essere istanza della classe *Exception* o di una sua sottoclasse. Ciò dirotta l'esecuzione del programma verso l'appropriato gestore di eccezioni. Ad esempio:

```
class MyException extends Exception
```

```
{ /* ... */ }
```

```
class MyClass
```

```
{
```

```
    // ...
```

```
    void oops()
```

```
    {
```

```
        if ( /* nessun errore */ )
```

```
        { /* ... */ }
```

```
        else
```

```
            throw new MyException();
```

```
    }
```

```
}
```

Per definire un gestore di eccezioni, il programmatore deve circondare il segmento di codice che può causare un'eccezione con un'istruzione *try*, seguita da una o più istruzioni *catch*, ognuna delle quali costituisce il vero e proprio gestore di una singola eccezione. Ad esempio:

```
try
```

```
{
```

```
    p.a=10;
```

```
} catch (NullPointerException e) {
```

```
    println("p e' un oggetto nullo");
```

```
} catch (Exception e) {
```

```
    println("C'e' stato qualche altro errore");
```

```
}
```

Una istruzione *catch* è equivalente alla definizione di un metodo con un solo parametro, che può essere una classe o un'interfaccia, e nessun tipo di ritorno. Si dice che il parametro coincide con l'eccezione se è istanza della stessa classe dell'eccezione, o di una sua superclasse; se il parametro è un'interfaccia, la classe dell'eccezione deve implementarla.

Una volta eseguita l'istruzione *catch*, l'esecuzione del programma riprende dal blocco successivo al costrutto *try/catch*: non è possibile, per il programma, continuare l'esecuzione dal punto in cui è avvenuta l'eccezione.

**L'istruzione *finally*:** un'istruzione *finally* garantisce che una certa parte del codice sia eseguita, indipendentemente dal fatto che l'eccezione si verifichi o no. Nell'esempio seguente, la parola "fine" viene stampata in ogni caso, ma le parole "dopo try" sono stampate solo se *a* è diverso da 10.

```
try
{
    if (a==10)
        return;
} finally {
    println("fine");
}
println("dopo try");
```

<b>Programmazione orientata agli oggetti</b> .....	<b>1</b>
Identità .....	1
Classificazione .....	1
Ereditarietà .....	1
Polimorfismo e "dynamic binding" .....	2
Incapsulamento .....	2
<b>Il linguaggio Java</b> .....	<b>2</b>
Familiarità e semplicità .....	2
Programmazione orientata agli oggetti .....	3
Neutralità .....	3
Portabilità .....	3
Robustezza .....	3
<b>Java: interpretato e dinamico</b> .....	<b>4</b>
<b>Sicurezza</b> .....	<b>4</b>
Allocazione della memoria .....	4
Verifica del bytecode .....	5
Caricamento del bytecode .....	5
Networking Package .....	5
<b>Multithreading</b> .....	<b>6</b>
<b>Specifiche del linguaggio</b> .....	<b>6</b>
Tipi di dato .....	6
Tipi numerici .....	7
Tipi carattere .....	7
Tipi booleani .....	7
Vettori .....	7
Classi .....	7
Casting fra le classi .....	9
Metodi .....	9
Le variabili this e super .....	10
Variabili di istanza .....	10
Overriding .....	10
Overloading .....	10
Costruttori .....	11
Creazione degli oggetti: .....	11
Garbage collection .....	11
Metodi, variabili e inicializzatori statici .....	11
Ordine di dichiarazione e di inicializzazione .....	12
Specificatori di accesso .....	12
Regole di visibilità delle variabili .....	13
Modificatori .....	13
Interfacce .....	15
Interfacce come tipi: .....	15
Combinazione di interfacce .....	16
Package .....	16
Uso di classi ed interfacce da altri package .....	16
Eccezioni .....	16
L'istruzione finally .....	18