

Java

Licenza

Questo lavoro è rilasciato sotto licenza **Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License**. Per vedere una copia di questa licenza visita <http://creativecommons.org/licenses/by-nc-sa/4.0/> o invia una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



Prefazione

Questo manuale vuole essere di aiuto a me per ripassare in vista del tirocinio e per gli studenti di *Programmazione ad Oggetti* del PoliTo come eventuale riferimento in fase di preparazione dell'esame.

Buon lavoro a tutti,

Francesco Clemente

Java Environment

Java mette a disposizione costrutti OO: definizione di classi in modo gerarchico, creazione/distruzione dinamica, scambio di messaggi.

Non offre costrutti procedurali: non ci sono funzioni o variabili globali.

È indipendente dalla piattaforma, perchè trasformato in linguaggio intermedio (bytecode) e poi interpretato.

É estremamente dinamico e robusto: controlli in compilazione, a run-time, garbage collector e gestione errori a runtime con il meccanismo delle eccezioni.

Supporta generazione della documentazione e raggruppamento in librerie (packages). Include un sacco di utility nelle librerie standard (concorrenza, GUI, networking).

Classi: una classe è una rappresentazione astratta di una entità, ne definisce la struttura.

```
/* MyFirstClass.java */
public class MyFirstClass{
    // Stuff
}
```

Metodi: gli strumenti mediante cui è possibile operare sugli oggetti (istanze di classi, ovvero rappresentazioni concrete con specifiche caratteristiche di ciò che è definito da una classe).

Esiste un metodo `main` da cui parte l'esecuzione del programma.

```
public static void main(String[] args){
    // Stuff
}
```

Una volta compilati i file `.java` si ottengono i file `.class` in bytecode che poi sono eseguibili dalla JVM (Java Virtual Machine) attraverso il comando `java MyFirstClass` ad esempio.

I programmi Java possono essere "*impacchettati*" in file **jar** che sono archivi compressi che contengono delle meta-informazioni per specificarne l'esecuzione.

Convenzioni:

- Si usa la **camelBackCapitalization** per nomi composti
- Le classi hanno nomi che iniziano per maiuscola
- Metodi, oggetti, attributi, variabili locali iniziano per minuscola
- Le costanti sono tutte in maiuscolo, se su più parole separate da underscore

Le basi di Java

I commenti si fanno come in C:

- Su più linee `/* commento */`
- Su singola linea con `// commento`.
- Esiste inoltre una tipologia di commenti detti **Javadoc** che permettono poi di costruire in automatico la documentazione e si fanno su più linee con la sintassi

```
/**
 * Questo è un commento Javadoc
 * @author Francesco Clemente
 * Questa classe/metodo fanno cose
 */
```

Ogni blocco di codice è racchiuso tra graffe e queste definiscono lo *scope* di una variabile.

Le variabili possono dichiarate anche "in mezzo" al codice e non necessariamente in cima.

I costrutti sono gli stessi del C nella maggior parte dei casi: `if-else`, `switch`, `while`, `do-while`, `for`, `break`, `continue`.

Nel caso dello `switch` ore è possibile anche su una stringa.

Esiste inoltre il tipo `boolean` che ammette i valori `true` e `false` ed è l'unico tipo ammesso come condizione.

In java sono ammessi i tipi primitivi del C come `int`, `float`, `double`, `char` e così via; si dichiarano allo stesso modo e nel momento in cui vengono dichiarati gli si alloca lo spazio necessario in memoria. Nel momento in cui si dichiara un oggetto, di fatto si sta dichiarando solo un puntatore. Lo spazio vero e proprio in memoria viene allocato quando si fa uso della parola chiave `new` e viene chiamato il costruttore.

Gli `int` ed i `float` sono su 32 bit, `long` e `double` su 64 bit. I numeri vengono rappresentati in complemento a due, i `char` sono su 16 bit codificati in UTF16.

Allo stesso modo del C funzionano gli operatori aritmetici, di confronto, bitwise, assegnazione, incremento e logici (questi funzionano solo tra operandi booleani).

Classi ed oggetti

Classe: definisce una struttura comune per un insieme di oggetti; consiste in un set di *membri* che sono *attributi*, *metodi*, *costruttori*.

```
// file Persona.java

public class Persona{
    // attributi
    private String nome, cognome;
    private int annoDiNascita;
    private boolean eViva;

    /* costruttore: chiamato quando l'oggetto
       di tipo Persona viene creato con la
       parola chiave new */
    public Persona(String nome, String cognome, int anno){
        this.nome = nome;
        this.cognome = cognome;
        annoDiNascita = anno;
        eViva = true;
    }

    // metodi: i messaggi che può ricevere
    public void cambioNome(String newName){
        nome = newName;
    }

    public void deceduta(){
        eViva = false;
    }
}
```

```
public boolean eAncoraViva(){
    return eViva;
}
}
```

I metodi rappresentano i messaggi che gli oggetti possono ricevere, ovvero è il modo in cui è possibile interagirvi.

Una classe può definire più volte un metodo con lo stesso nome, a patto che abbia diversa *signature* (insieme di nome metodo e lista ordinata dei tipi degli argomenti). In tal caso si parla di **overloading**.

Oggetto: una istanza specifica di un qualcosa di generico definito da una classe. È identificato da una classe a cui appartiene, uno *stato*, ovvero i valori dei suoi attributi, ed un *identificatore interno univoco* che è una sorta di indirizzo di memoria. Da zero a molti riferimenti possono puntare allo stesso oggetto (*aliasing*).

```
...
Persona maria = new Persona("Maria", "Rossi", 1923);
Persona laMiaCaraVicina = maria;
maria.deceduta();
if (laMiaCaraVicina.eAncoraViva())
    System.out.println("È viva.");
else
    System.out.println("F");
// verrà stampato a schermo che è deceduta
// in quanto maria e laMiaCaraVicina coincidono
// ovvero sono solo modi diversi di interrogare
// la stessa area di memoria che ha uno specifico
// indirizzo
...
```

Quando un oggetto viene definito, prima che venga effettuata la `new` e che venga quindi inizializzato, esso punta a `null`.

Quando un oggetto diventa *unreachable*, ovvero non esiste più nessun *reference* ad esso, questo occupa la memoria fintanto che il **garbage collector** non libererà la memoria ad esso associato.

La parola chiave `new` permette di allocare la memoria nell'heap associata all'oggetto che si sta creando. Chiama il costruttore dell'oggetto, un particolare metodo senza tipo di ritorno ed avente lo stesso nome della classe, il quale restituisce il riferimento all'oggetto appena creato.

Ovvermai non fosse definito esplicitamente un costruttore, viene chiamato quello di default che è "sottointeso" e non prevede argomenti.

Quando viene creato un nuovo oggetto, i suoi attributi, salvo esplicitamente dichiarato, vengono messi a 0 nel caso di tipi numerici, a falso nel caso di booleani ed i riferimenti a `null`.

N.B. è possibile fare overload anche del costruttore.

Il rilascio della memoria non è fatto esplicitamente; volendo è possibile definire un metodo `public void finalize()` che verrà invocato al "passaggio" del garbage collector.

La keyword `this` permette di far riferimento agli attributi dell'oggetto ovvermai esistesse una ambiguità quando i parametri di un metodo hanno lo stesso nome degli attributi dell'oggetto stesso.

Ci si riferisce ai metodi o gli attributi di un oggetto attraverso la **dotted notation** ovvero la sintassi con il "punto". Nel caso in cui un metodo o attributo venisse chiamato da un metodo dello stesso oggetto, non è necessario farvi riferimento con la dotted notation salvo per risolvere eventuali ambiguità. La dotted notation funziona anche concatenata come si vede nella chiamata a `System.out.println("Hello world!");`

Gli operatori di confronto sui riferimenti fanno il confronto tra i riferimenti stessi e non sul loro contenuto.

Scope ed incapsulamento

L'incapsulamento permette di mettere in pratica: modularità, *information hiding* (identificare e delegare responsabilità ai componenti; solo i metodi pubblici possono essere chiamati al di fuori della classe).

I modificatori di visibilità applicabili ai membri di una classe sono: `private` che rende il membro invisibile all'esterno ed utilizzabile solo dall'interno della classe stessa; `public` che rende il membro visibile ed utilizzabile anche dall'esterno.

Esistono dei metodi che prendono il nome di *getter e setter* che vengono utilizzati per leggere e scrivere i valori di attributi privati, cosa che consente di effettuare anche dei controlli specifici sul valore prima di effettuare l'assegnazione.

Un **package** permette di raggruppare classi che hanno ragione di stare insieme. Questo consiste in una cartella che contiene tutte le classi che vi appartengono. Ogni package definisce un nuovo **scope** ovvero un nuovo "spazio di visibilità". È inoltre possibile usare lo stesso nome per una classe in package differenti, in tal caso però le classi omonime non saranno importabili contemporaneamente.

Un package è identificato da un nome con struttura gerarchica (*fully qualified name*). Per assegnare una classe ad un determinato package, oltre che posizionarlo nella corretta cartella, è necessario usare la keyword `package` e specificare il package di appartenenza come primissima cosa nel file sorgente.

Per essere utilizzato invece, è necessario fare una `import` nella classe che lo vuole utilizzare.

N.B. Gli import non sono ricorsivi; eventuali sottopackage non saranno importati.

```
package it.clementefnc.myFistPackage;

import it.clementefnc.myThirdPackage.*;
import it.clementefnc.mySecondPackage.myFirstClass;
import it.clementefnc.myClass;

public class mySecondClass{
    private myFirstClass mfc = new myFirstClass();
    private it.clementefnc.myClass mc1;
}
```

Nel caso in cui non venga specificato alcun package, la classe apparterrà al package di default. Ciò implica che questa non sia accessibile da altre classi che risiedono in altri packages.

L'interfaccia (ciò che è visibile all'esterno) di un package consiste nelle classi pubbliche che lo compongono.

Nel caso in cui si dovesse omettere il modificatore di visibilità (`public` o `private`) si parla di visibilità di package e si intende che tutto è visibile alle classi appartenenti allo stesso package ma non al di fuori.

Definire una classe privata non ha senso perchè sarebbe visibile solo a se stessa, ergo risulterebbe inutilizzabile.

Stringhe

In Java non esistono gli array di char sono diversi dalle stringhe.

`String` è una classe appartenente alla libreria `java.lang` ed è un tipo immutabile ovvero non modificabile (non si possono cambiare le lettere ad esempio, ma si può riassegnare).

Esiste la classe `StringBuffer` che è modificabile.

Sono concatenabili con l'operatore `+`.

La classe `String` ha il metodo `int length()` che restituisce la lunghezza o `boolean equals(String s)` che permette di fare un confronto con un'altra stringa, `String toUpperCase()` e tanti altri.

La classe `StringBuffer` ha il metodo `append(String str)` per effettuare la concatenazione o `insert (int offset, String str)` per effettuare un inserimento "in mezzo" e così via.

In Java i caratteri sono rappresentati in Unicode.

Classi wrapper

Le classi wrapper sono la "versione ad oggetti" dei tipi primitivi; mettono a disposizione operazioni di conversione tra differenti tipi.

Nel package `java.lang` troviamo:

- Tipi primitivi: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`, `void`
- Classi wrapper: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Void`

Il sistema di **autoboxing/autounboxing** consente di effettuare una conversione automatica tra classi wrapper e tipi primitivi.

Array

Un array è una sequenza ordinata di variabili omogenee a cui si accede a mezzo indice. Possono contenere tipi primitivi o riferimenti ad oggetti. La dimensione di un array può essere definita a run-time durante la creazione dell'oggetto ma non cambiata dopo.

Viene dichiarato con una delle seguenti sintassi equivalenti:

```
int[] ary;  
int ary[];
```

Un array è un oggetto ed è memorizzato nell'heap per cui è necessario inizializzarlo con l'operatore new o effettuando una inizializzazione statica

```
int[] ary1 = new int[10];  
int[] ary2 = {1, 2, 3, 4, 5};  
Gadget[] mieiGadget = {new Gadget("Cuffie"),  
                        new Gadget("Cellulare")};  
// vi si accede a mezzo indice  
System.out.println(ary2[2]); //stamperà 3
```

N.B. In Java non esiste l'aritmetica dei puntatori

Esiste un costrutto che prende il nome di for-each che permette di ciclare su tutti gli elementi di un array o di una classe che *implementa* Iterable (parleremo delle *interfacce* più avanti).

La sintassi è la seguente for (Type var : set_expression) dove Type è il tipo della variabile i-esima di ciclo var e set_expression è il nostro oggetto iterabile. In pratica ad ogni loop var conterrà l'elemento i-esimo.

È possibile costruire anche array multidimensionali come array di array.

N.B. L'allocazione della memoria non è contigua per cui è possibile riassegnare facilmente una "riga".

Esiste la possibilità di creare metodi con un numero non ben definito di argomenti utilizzando la notazione varargs.

```
public void myMethod(int ... values){  
    for(int x : values){  
        System.out.println(x);  
    }  
}
```

static e final

Le variabili di classe definite con modificatore `static` rappresentano una proprietà comune a tutte le istanze della classe. Esistono anche se nessun oggetto non è mai stato istanziato (ancora).

I metodi `static` non sono specifici di alcuna istanza, sono utilizzati per implementare *funzioni*.

L'accesso ad un membro statico per cui avviene chiamando il nome della classe e non quello dell'oggetto.

È possibile importare tutti gli elementi statici attraverso una *import statica* con la keyword `import static fully.qualified.name.*;`

Quando un attributo è dichiarato con il modificatore `final` esso non può essere più modificato dopo la costruzione dell'oggetto; può essere inizializzato *inline* o dal costruttore.

Dichiarare una variabile `final static` consente in pratica di dichiarare una costante, per cui è opportuno utilizzare un nome variabile completamente in maiuscolo.

Esistono anche i blocchi di inizializzazione statica che vengono eseguiti nel momento in cui la classe viene caricata (ovvero all'avvio del programma).

enum

Un tipo enumerativo può assumere solo uno dei valori enumerati.

```
public enum ColoriPrimari {  
    ROSSO, GIALLO, BLU  
}  
...  
ColoriPrimari colore = ColoriPrimari.ROSSO;
```

Memoria

Java utilizza 3 aree di memoria:

- Statica: elementi che vivono per l'intera esecuzione del programma (definizione di classi, variabili statiche)
- Heap: elementi creati a run-time (con la keyword `new`)
- Stack: elementi creati in blocco di codice (variabili locali e parametri dei metodi)

Per quanto riguarda la classe `String`, essa mantiene un *pool* di stringhe distinte.

Il metodo `intern()` della classe `String`, cerca nel pool se c'è una stringa `equals()` a quella in questione, se non c'è la aggiunge e restituisce il reference a quest'ultima. Per ogni *literal* il compilatore effettua una chiamata ad `intern()`.

Il **garbage collector** è un componente della JVM che si occupa di pulire la memoria dagli oggetti non più referenziati da una variabili. Viene eseguito periodicamente senza una temporizzazione predefinita.

Classi annidate

Ci sono 4 tipi di classi annidate:

- Static nested class: una classe dichiarata all'interno di un'altra classe con l'attributo statico; si comporta come una classe normale. Sono usate per nascondere le annidate all'esterno e permette di incapsulare dettagli interni
- Inner class (ovvero una classe innestata non statica): ha il vantaggio di poter essere vista dall'esterno
- Local inner class: dichiarata dentro un metodo

```
public void method(){
    final int j=1;
    class X {
        int plus(){ return j + 1; }
    }
    X x = new X();
    System.out.println(x.plus());
}
```

Il reference alla variabile j viene sostituito con il valore corrente.

- Anonymous inner class: classi locali senza nome; ottenibili solo attraverso l'ereditarietà (implementa una interfaccia o estende una classe).

Ereditarietà

Con ereditarietà si intende la possibilità di una classe di essere "sotto-tipo" di un'altra; la classe derivata conterrà tutti i membri della classe superiore più quello che essa definirà. La classe "figlia" avrà inoltre la possibilità di fare **override** della definizione dei metodi esistenti proponendone una sua versione.

```
// Persona.java
class Persona{
    private String nome, cognome;
    private int annoNascita;
    public Persona(String nome, String cognome, int anno);
    public int getAnnoNascita(){
        return annoNascita;
    }
}

// Lavoratore.java
class Lavoratore extends Persona{
    private String impiego;
    private int annoInizioLavoro;
```

```

public Lavoratore(String nome, String cognome,
                  int annoNascita, int annoInizLavoro){
    super(nome,cognome,annoNascita);
    this.annoInizioLavoro=annoInizLavoro;
}
public int getAnnoInizioLavoro(){..}
}
// Main
Lavoratore l = new Lavoratore("Mario","Rossi",1970,1989);
System.out.println(l.getAnnoNascita());

```

Si parla di polimorfismo quando si fa una cosa del tipo:

```

Persona p = new Lavoratore(..);
// ciò è corretto perchè un lavoratore è una persona

```

L'ereditarietà è una tecnica estremamente funzionale nel caso del riuso: ereditando del codice minimizza le ripetizioni di codice e permette di fare le modifiche in un solo punto e propagarle nelle sottoclassi.

È possibile sviluppare l'ereditarietà su più livelli e rappresentare la struttura tramite albero di ereditarietà; è importante non crearne troppi per evitare confusione.

Il casting è possibile anche nel caso delle classi.

```

Persona p = new Persona(..);
Lavoratore l = new Lavoratore(..);
Persona pl = l;

```

In tal caso si parla di **upcast** ed è sempre ammesso; ciò permette di trattare oggetti di classi differenti in funzione del fatto che sono figli di una classe comune. Il casting non modifica la struttura interna di un oggetto, per cui se un oggetto è di una data classe, resterà di quella classe anche se ci riferiamo a lui facendogli riferimento come se fosse del tipo di un suo antenato.

Il **downcast** ovvero l'assegnazione di un tipo più generico (e.g. *Persona*) ad uno più specifico (e.g. *Lavoratore*) non è sempre possibile e non è automatico, deve essere fatto *esplicitamente*. Qualora si facesse una operazione illegale, questa viene rilevata a *runtime* e viene lanciata una *eccezione* (ne parleremo più avanti).

È possibile usare l'operatore `instanceof` per avere informazioni su un reference, ovvero sapere se è di una data classe:

```

if (persone[i] instanceof Lavoratore){
    lav = (Lavoratore) persone[i];
}

```

Polimorfismo e Dynamic Binding

Un riferimento di un tipo A, può puntare ad un oggetto di tipo B se e solo se B coincide con T o B è una sottoclasse di A.

Il compilatore effettua un controllo sull'invocazione di un metodo in funzione del tipo di riferimento.

Si parla di **Dynamic Binding** il processo con cui si effettua una scelta tra metodi per cui è stato fatto l'override. In particolare se si sta invocando un metodo per cui è stato fatto l'override nel caso di un oggetto che in quel momento è stato referenziato a mezzo di una classe antenata, viene comunque chiamato il metodo della classe figlia.

Ciò permette, nel caso di più figli che hanno fatto override distinti di un metodo di un genitore, di essere trattati in modo uniforme se presi "tutti insieme", ma di gestire le cose in modo "personalizzato".

La parola chiave `super` permette di far riferimento alla classe padre e quindi accedere ai suoi attributi, metodi e costruttori a patto che non siano privati.

Per identificare un attributo, metodo o costruttore privato ma visibile alle classi derivate, si usa il modificatore `protected`.

Classe Object

`java.lang.Object` è la classe di cui tutte le altre sono figlie. Ogni classe è direttamente o indirettamente una sottoclasse di `Object`, per cui qualsiasi oggetto può essere visto come una istanza di `Object`.

La classe `Object` mette a disposizione dei servizi utili per le altre classi (e.g. i metodi `toString()` ed `equals(Object)`, `hashCode()`, `clone()`); di questi spesso è fatto l'override.

Si può dire che i reference di tipo `Object` siano analoghi ai `*void` del C.

Classi astratte

Spesso una superclasse è utilizzata solo per definire un comportamento comune per delle classi figlie, ma alcuni metodi potrebbero non avere implementazioni ovvie nella superclasse. Le classi astratte permettono di lasciare il comportamento parzialmente definito, cosa che però le rende non istanziabili.

Una classe astratta deve essere definita con il modificatore `abstract` ed a sua volta ogni metodo incompleto deve avere lo stesso modificatore.

Interfacce

Le interfacce sono uno speciale tipo di classe dove i metodi sono implicitamente `abstract`, gli attributi implicitamente `static` e `final` e tutti i membri implicitamente `public`.

Viene definita con la parola chiave `interface` invece che con `class`.

Una classe `implements` una interfaccia. Una classe può implementare più interfacce ma può estendere solo una classe.

Quando si implementa una interfaccia, tutti i suoi metodi devono essere implementati a meno che la classe non sia astratta.

Una interfaccia può estendere un'altra o più interfacce.

Spesso le interfacce si utilizzano per definire classi anonime, ovvero classi definite inline implementando i metodi:

```
Iface obj = new Iface(){  
    public void method(){..}  
}
```

Un esempio di interfaccia è `java.lang.Comparable` che mette a disposizione il metodo `compareTo(Object)` che permette di effettuare il confronto tra due oggetti e definire un ordine.

Similmente si comporta `java.util.Comparator` con il metodo `compare (Object, Object)`.

Un'altra interfaccia comune, citata precedentemente è `java.lang.Iterable` che permette di essere utilizzata all'interno di un `foreach`. Al suo interno definisce un metodo `iterator()` che restituisce un oggetto di tipo `java.util.Iterator` il quale possiede due metodi `hasNext()` e `next()` i quali rispettivamente restituiscono un `boolean` ed un `Object`.

Metodi default

Una implementazione per i metodi delle interfacce può essere fornita nel caso in cui siano definiti con modificatore `default` i quali: non possono far riferimento ad attributi non-statici, possono referenziare argomenti o altri metodi, possono essere *overridden*.

Sono utilizzati per aggiungere funzionalità ad interfacce già esistenti senza "rompere" le classi che implementano.

Intefacce funzionali

Interfaccia che contiene un solo metodo: gli `static` e `default` non contano.

La semantica è funzionale: il risultato del metodo è basato sui soli argomenti.

Sono definite in `java.util.function`

Anonymous Inner class e Lambda

Sono una implementazione *inline* di interfacce contenenti un solo metodo.

```
// Stampante.java
public interface Stampante{
    public void handle(Object o);
}
// Main
Stampante s = new Stampante(){
    public void handle(Object o){
        System.out.println(o);
    }
}
```

Una funzione Lambda è una definizione di una anonymous inner class "al volo". Equivalentemente al codice di prima, si può scrivere:

```
Stampante s = o -> System.out.println(o);
```

Le lambda hanno la seguente sintassi:

- Parametri
 - Nessuno: `()`
 - Uno: `x`
 - Due o più: `(x, y)`
 - I tipi possono essere omessi
- Corpo:
 - Espressione: `x + y`
 - Blocco di codice: `{return x+y;}`

I tipi sono generalmente omessi in quanto il compilatore assegna automaticamente il tipo corretto in funzione del contesto.

Un **method reference** è una rappresentazione compatta di una istanza di una interfaccia funzionale che invoca un singolo metodo.

```
Consumer stampante;  
stampante = System.out::println;  
stampante.consume("Hello!");
```