1
Linguaggio Java

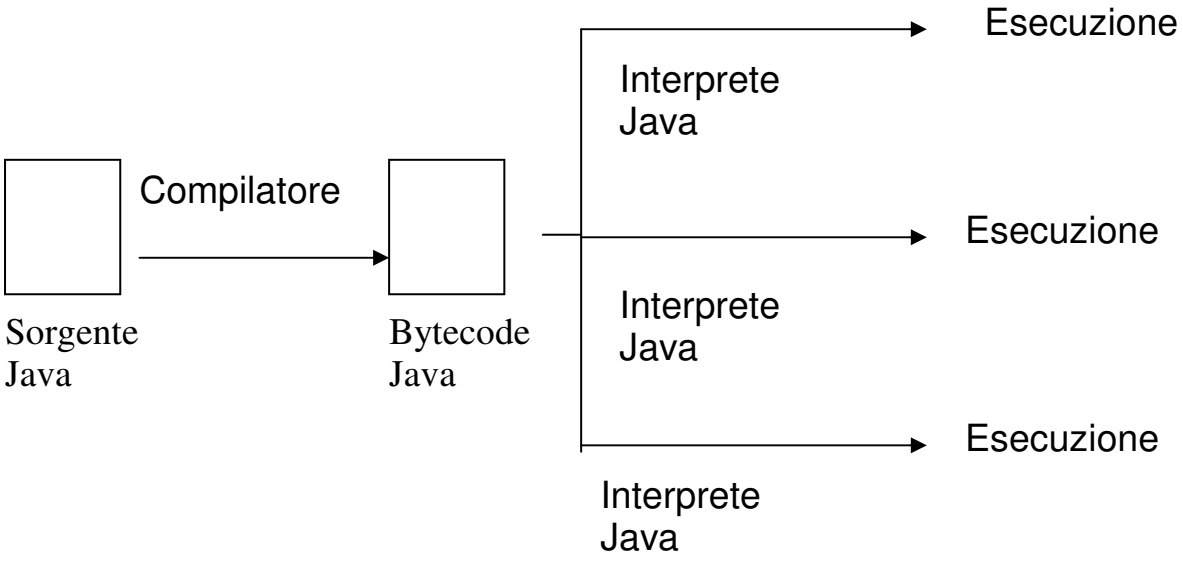
1.1 Introduzione

Java è un linguaggio orientato agli oggetti sviluppato da Sun Microsystem

Java consente di realizzare applicazioni e applet

Un applet è un programma scaricabile dalla rete ed eseguibile da un browser all'interno di una pagina web.

L'ambiente Java comprende sia un compilatore che un interprete.



Bytecode: pseudo istruzioni macchina indipendenti dalla piattaforma

Compensazione velocità - **portabilità**

L'interprete è incorporato nei browser abilitati Java

```
SET PATH=…;C:\JDK1.1\BIN
SET CLASSPATH=C:JDK1.1\lib\classes.zip;.;
```

Esempio

```
class Ciao1 {
        public static void main(String args[]) {
                        System.out.println("Ciao!!");
                                                        }
}
```

I file sorgenti Java devono avere lo stesso nome della classe che definiscono.

javac Ciao1.java

java Ciao1

2
Programmazione ad oggetti in Java

2.1 Classi ed oggetti

classe: modello generico di un insieme di oggetti con caratteristiche simili

oggetto: particolare istanza concreta di un elemento di una classe

2.2 Attributi e comportamento

gli attributi descrivono lo stato, l'aspetto e le altre qualità dell'oggetto;

gli attributi sono rappresentati dai campi dell'oggetto

esistono anche campi di classe

il comportamento di una classe descrive come opera un oggetto di quella classe;

il comportamento è definito dai metodi della classe, funzioni definite all'interno della classe che operano sugli oggetti della classe;

In Java non esistono funzioni definite al di fuori di una classe

Esempio

```
class Motocicletta { }
```

==========

```
class Motocicletta {
String marca;
String colore;
boolean accesa = false;

void accendi()
        { if (accesa == true)
                System.out.println("È già accesa");
        else
                {
                accesa = true;
                }
        }

void mostra()
        {System.out.println("Questa moto è una "+ marca+"  "+colore);
        if (accesa==true)
                System.out.println("È accesa ");
        else
                System.out.println("È spenta ");
        }
}

class ProgrammaMotocicletta {

public static void main (String args[]) {
        Motocicletta m;
        Motocicletta m2;
        m=new Motocicletta();
        m.marca = "Guzzi 1000";
        m.colore = "rossa";
        System.out.println("Descrizione: ");
        m.mostra();
        System.out.println("Accensione: ");
        m.accendi();
        System.out.println("Accensione: ");
        m.accendi();
        m2=new Motocicletta();
        m2.marca = "Honda 1000";
        m2.colore = "nera";
        System.out.println("Descrizione: ");
        m2.mostra();
```

```
        System.out.println(m == m2);  //false
        Motocicletta m3;
        m3 = m2;
        System.out.println(m3 == m2);  //true
 }}
```

==========

```
class Motocicletta {
String marca;
String colore;
boolean accesa = false;

void accendi()
{ if (accesa == true)
        System.out.println("È già accesa");
else
        {
        accesa = true;
        System.out.println("Ora è accesa");
        }
}
void mostra()
{System.out.printnl("Questa moto è una "+ marca+"  "+colore);
if (accesa==true)
        System.out.println("È accesa ");
else
        System.out.println("È spenta ");
}

public static void main (String args[]) {
        Motocicletta m;
        m=new Motocicletta();
        m.marca = "Guzzi 1000";
        m.colore = "rossa";
        System.out.println("Descrizione: ");
        m.mostra();
        System.out.println("Accensione: ");
        m.accendi();
        System.out.println("Accensione: ");
        m.accendi();
 }
}
```

2.3 Ereditarietà, interfacce e packages

E' possibile definire una sottoclasse discendente da una sovraclasse;

La sottoclasse eredita campi e metodi della sovraclasse.

Ogni classe può avere una sola sovraclasse;

Le interfacce sono collezioni di dichiarazioni di metodi che possono essere aggiunti alle classi; consentono di avere comportamenti analoghi in rami diversi della gerarchia delle classi.

Un package è una collezione di classi e interfacce. Consente di rendere disponibili gruppi di classi quando è necessario e consente di eliminare conflitti potenziali fra i nomi delle classi.

La libreria delle classi del Java Developer's Kit è contenuta nel package java, l'unico che è garantito presente in tutte le implementazioni.

Per default sono disponibili solo le classi contenute nel package java.lang

Altre classi vanno riferite esplicitamente o importate: java.awt.Color

Esempio

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

// per importare tutto il package:
/*
import java.awt.*;
*/

public class Ciao2Applet extends java.applet.Applet {



Font f = new Font("TimesRoman",Font.BOLD,36);

public void paint (Graphics g) {
                g.setFont(f);
                g.setColor(Color.RED);
                g.drawString("Hello again!",5,25);
                                                }
}
```

```
<HTML>

    <HEAD>
        <TITLE>Secondo Applet</TITLE>
    </HEAD>

    <BODY>
        <P>Il mio secondo applet dice:
        <APPLET CODE="Ciao2Applet.class"
                WIDTH=150 HEIGHT=50>
        </APPLET>
    </BODY>

</HTML>
```

Hyper Text Markup Language

3
Elementi del linguaggio

3.1 Commenti

I compilatori Java fanno differenza fra lettere maiuscole e minuscole.

```
/*
questo e` un commento
*/

/**
questo e` un commento
*/


// questo è un commento
```

il carattere "**;**" rappresenta un terminatore di istruzioni;
ogni istruzione deve essere terminata da un "**;**"

3.2 Variabili

Esistono 3 tipi di variabili:
campi di oggetti, campi di classe, variabili locali a metodi

Esempio

```
int i,j;
int k = 8;
String parola;
int m = 0, n=1;
```

3.3 Tipi di dato

Il tipo dei dati definisce l'**insieme di valori**, o dominio, che un oggetto può assumere, inoltre definisce le **operazioni** che possono essere effettuate sugli oggetti del tipo specificato.
Il tipo delle variabili deve essere definito esplicitamente in Java.

I tipi di dato in Java possono essere classificati come segue:

tipi di dato semplici che distinguiamo in
- **tipo void**
- **tipi reali:**
  - **tipo float** (32 bit, singola precisione)
  - **tipo double** (64 bit, doppia precisione)
- **tipi interi (integral);**
  - **tipo byte;** (8 bit, [-128,127])
  - **tipo short;** (16 bit, [-32.768,32.767])
  - **tipo int;** (32 bit, [-2.147.483.648, 2.147.483.647])
  - **tipo long;** (64 bit, [-9.223.372.036.854.775.808, 9.223.372.036.854.775.807])
- **tipo char;** (16 bit, standard Unicode)
- **tipo boolean** (true, false);

tipo array;
classi e interfacce

Un oggetto di una certa classe può assumere valori di oggetti di una qualsiasi sottoclasse. (**polimorfismo** sui dati)

```
class Persona {…}

class Studente extends Persona {…}

double x = 1;

Persona p = new Persona();          Persona p = new Studente();
                                    Studente s = new Studente();
```

Valori costanti

4 4L -15
0777 0004 (otttali)
0x12AF (esadecimale)
5.123 -23.45f
10E10 .12E-3

false true

'a' 'b'

stringhe:

"sono una stringa"
"contengo una \" stringa annidata\""

Caratteri speciali:

\'	'
\"	"
\n	a capo
\t	tab
\b	backspace
\\	\
\ddd	ottale
\xdd	esadecimale
\udddd	carattere unicode

3.4 Operatori

Operatori booleani

!	not logico
&&	and logico
\|\| | or logico |

valutazione short circuited true || (f(45) && g(7))

operatori aritmetici

\+ - *
/ *con operandi interi restituisce il quoziente della divisione fra interi*
% *resto della divisione fra interi*

5 / 2 -> 5%2 -> 5.0/2 ->
((double) 5) /2 -> (double) (5 /2) ->

operatori relazionali
== > < >= <= !=

operatori orientati ai bit
& and bit a bit
12 & 7 ->
| or bit a bit
12 |7 ->

^ or esclusivo bit a bit
~ complemento bit a bit
<< shift a sin.
>> shift a dest.

operatori di incremento (si applicano a variabili intere)

++ --

int m=0; int n = 0; m = ++n;
n->
m ->
int m=0; int n = 0; m = n++;
n->
m ->

concatenazione fra stringhe
+

String s = “Ciao “ + “mamma”;

s ->

String t = “Ciao “ + 2;
t ->

---

int n = 2*7-3-1;

n ->

int a,b;

a =b=3;

a->
b->

Precedenza operatori

Livello	Operatori
1	() [] .
2	! ~ ++ -- instanceof
3	new (tipo)
4	* / %
5	+ -
6	<< >>
7	< <= > >=
8	== !=
9	& *(and bit a bit)*
10	^
11	|
12	&&
13	||
14	? : C?E1:E2
15	= += -= *= /= %= &= ^= |= <<= >>=

m=1;

m+= 2; m ->

La valutazione degli operatori logici avviene da sinistra a destra. La valutazione si interrompe non appena è possibile stabilire il valore dell'espressione.

boolean b1;

int a =1, b = 2;

b1 = a < 3 && (a > 0 || b % 3 == 0);

b1 ->

Associatività

da sinistra a destra:

* / % + - << >> < <= > >= ==

!= & ^ | && ||

da destra a sinistra:

| ~ ++ -- (tipo) ?: = += -= /=

%= ^= |= <<= >>=

3.5 Operatori di assegnamento

L'operatore di assegnamento è **=**

L'esecuzione dell'espressione di assegnamento:

v1 = e1;

ha il seguente effetto:

1) viene valutato il valore dell'espressione e1 e tale valore viene scritto nella locazione che corrisponde a v1.
2) l'operatore di assegnamento restituisce inoltre il valore di e1.

Esempio

```
int a,b,c;

a=1;

b = c = a+2;
```

Oltre l'operatore di assegnamento semplice **=**, il Java offre anche i seguenti operatori di assegnamento composto:

***= /= %= += -= <<= >>= &= ^= |=**

In generale l'espressione:

V1 op= E2

ha lo stesso effetto di:

V1 = V1 op E2
con la differenza che E1 viene acceduto in memoria una sola volta.

Esempio
a = a+1; equivale a a += 1;

---

int a, b, c, d;

a=b=c=0;

a = ;

b = a++;

c = ++a;

a ◊ b ◊ c ◊

a++;

3.6 Tipi array

Gli array definiscono un tipo strutturato omogeneo.

Un array è un *insieme di elementi dello stesso tipo* senza limitazioni

Gli array Java sono oggetti.

Essendo oggetti gli array vanno prima dichiarati poi istanziati (con una chiamata di coastruttore).

La dichiarazione di una variabile di tipo array ha la seguente forma:

<DichiarazioneVariabileArray> ::=
<TipoElementi> <IdentificatoreVariabile> '[' ']**;**' |
<TipoElementi>'[' ']' <IdentificatoreVariabile>'**;**'

La creazione di un oggetto array può essere effettuata utilizzando new o elencando il contenuto dell'array al momento della dichiarazione; nel primo caso la dimensione dell'array viene indicata esplicitamente, nel secondo implicitamente.

Esempio

```
String [] colori = {"blu", "giallo","rosso"};

int numeri [];
numeri = new int[50];
```

Selezione di un elemento
attraverso l'utilizzo di un indice può essere selezionato un componente dell'array; il primo elemento dell'array ha indice 0; il campo length, comune a tutti gli oggetti array, contiene il numero di componenti dell'array.

Esempio

```
int [] vettore ;                    int n;
vettore = new int[100];             n = 0;

int v1[]  = new int[100];           int n = 0;

v1[0] = 1;
v1[99] = 2;
```

```
for (int i=0; i < v1.length; i++)
        v1[i] =  2*i;
```

Quando si inizializza con new gli array di interi contengono 0, quelli di booleani false, quelli di caratteri \0 e quelli di oggetti null.

Tipi array multidimensionali

È possibile definire *array multidimensionali* ovvero array i cui elementi sono a loro volta degli array.

Esempio

```
int matrix [][] = new int[100] [100];
matrix [0][0] = 1;
matrix [0][1] = 2;
int m [] = matrix[2];
```

3.7 Istruzione semplici

Assegnamento

Chiamata di metodo

3.8 Istruzione composta

Un'istruzione composta, anche detta blocco, è costituita da un gruppo di istruzioni chiuse fra parentesi graffe precedute da una parte dichiarativa opzionale.

<IstruzioneComposta> ::=
'**{**' [ParteDichiarativa] Istruzione '**;**' {Istruzione '**;**'} '**}**'

Un'istruzione composta può essere utilizzata ovunque può essere utilizzata un'istruzione semplice.

Le dichiarazioni di un blocco sono locali al blocco stesso e sono visibili nei blocchi annidati

Nei blocchi annidati non è possibile ridefinire una variabile.

Esempio

```
class Esempio{
        int x = 5;
void blocchi()
        { int a,b;
         a = 1;
         b = 3;
        {
                // int a;   //non posso ridefinire la variabiloe in un blocco
        annidato
                int c;
                a = 2;          c=4;
                System.out.println(a);              // a =
                System.out.println(b);              // b =
                System.out.println(c);              // c =

                System.out.println(x);              // x =


         }
         System.out.println(a);             // a =
         System.out.println(b);             // b =
        // System.out.println(c);           // c non è visibile qui

}

public static void main (String args []){

int a=0; int b=1; int c=2;

Esempio e;
e = new Esempio();
e.blocchi();

System.out.println(a);              //  a = */
System.out.println(b);              //  b = */
 System.out.println(c);             //  c = */

}
}
```

3.9 Istruzioni condizionali o di selezione

In molte circostanze è necessario scegliere tra più azioni da compiere, a questo scopo vengono utilizzate le istruzioni di selezione.

Istruzione condizionale di selezione singola *if*

L'istruzione **if** rappresenta la scelta tra due alternative e dal punto di vista sintattico ha due forme:

if (<condition>) <statement1>;

Se <condition> ha valore di verità statement1 viene eseguito, altrimenti non viene seguito

if (<condition>) <statement1>;
else <statement2>;

Se <condition> ha valore di verità statement1 viene eseguito, altrimenti viene seguito <statement2>.

Nel caso di if annidati, ogni else viene riferito all'if più vicino a cui non è già stato associato nessun else.

If non è un operatore, nel senso che non restituisce valori.

Il linguaggio mette però a disposizione un operatore condizionale ternario:

? :

Quando viene valutata l'espressione:

E1 ? E2 : E3

E1 viene valutato. Se il suo valore è diverso da 0 (cioè VERO) viene valutato E2, altrimenti viene valutato E3. L'operatore restituisce il valore di E2 o E3 a seconda di quale dei due viene valutato.

Istruzione condizionale di selezione multipla *switch*

Sintassi:

switch '(' <espressione> ')'
'**{**'
{{**case** <constant expression> '**:**'} <statement>'**;**' [**break**';']}
[**default** '**:**' <statement>'**;**']
'**}**'

L'espressione, che deve essere di tipo può esser di tipo byte, char, short o int, viene valutata. Se il valore ottenuto è uguale ad una delle costanti indicate viene eseguita l'istruzione del ramo corrispondente e quelle di tutti i rami successivi. Se il valore dell'espressione non è stato indicato viene eseguita l'istruzione etichettata da **default** o niente se tale etichetta è assente.

Per eseguire solo l'istruzione corrispondente all'etichetta con valore uguale a quello dell'espressione evitando l'esecuzione delle istruzioni corrispondenti alle etichette successive si usa l'istruzione **break** che ha l'effetto di interrompere l'esecuzione dello switch e di passare all'esecuzione dell'istruzione successiva.

Esempio

```
{ switch (numero)
        case 1 : System.out.println("uno");
        case 2 : System.out.println("due");
        case 3 : System.out.println("tre");
        default : System.out.println("molti");
}
```

```
switch (numero)
{
        case 1 : System.out.println("uno"); break;
        case 2 : System.out.println("due"); break;
        case 3 : System.out.println("tre"); break;
        default : System.out.println("molti");
}
```

3.10 Istruzioni iterative

Specificano che uno statement, o più in generale una sequenza di statement deve essere ripetuta. In Java tutti i costrutti iterativi sono di tipo indeterminato.

Istruzione while

while (<condition>) <statement> “**;**”

L'istruzione **while** ha lo scopo di iterare un'istruzione fintantoché una certa condizione rimane vera. Quando una istruzione **while** viene eseguita la condizione viene valutata. Se la condizione è falsa il controllo passa all'istruzione successiva al costrutto **while**, se la condizione è vera viene eseguito lo statement specificato ed il controllo torna ad eseguire l'intera istruzione **while**. Dato che la valutazione della condizione avviene prima dell'esecuzione dell'istruzione se la condizione è falsa al primo passo l'istruzione non viene eseguita affatto.

Istruzione do while

do <statement> **while** (<condition>) ”**;”**

L'istruzione **do while** ha lo scopo di iterare un'istruzione fintantoché una certa condizione rimane vera. Quando una istruzione do while viene eseguita, viene eseguito lo statement specificato. Si passa poi a valutare la condizione. Se la condizione è falsa il controllo passa all'istruzione successiva al costrutto **do while**, se la condizione è vera ed il controllo torna ad eseguire l'intera istruzione **do while**. Dato che la valutazione della condizione avviene dopo l'esecuzione dell'istruzione questa verrà sicuramente eseguita almeno una volta.

Istruzione for

for "**(“** [<initialization>] "**;”** [<condition>] "**;”** [<increment>] "**)”** <statement>"**;**"

```
for (int i=0; i <100; i++)
        <istruzione>;
```

for (Elemento corrente = testa; corrente != null; corrente = corrente.next) <statement>"**;**"

```
for ( ; ; )
        <statement>";"
```

In Java anche l'istruzione for rappresenta una iterazione indeterminata.

Quando un'istruzione for viene eseguita per prima cosa si valuta l'inizializzazione. Poi si valuta la condizione. Se la condizione è falsa l'esecuzione del for termina, altrimenti si esegue lo statement, poi si valuta l'incremento si torna a valutare la condizione e così via.

3.11 Istruzioni di interruzione

Istruzione break

break ;

Quando l'istruzione break viene eseguita all'interno di un ciclo l'esecuzione del programma continua dalla prima istruzione che segue il ciclo più annidato.

Istruzione continue

continue ;

Quando l'istruzione continue viene eseguita all'interno di un ciclo l'esecuzione del programma prosegue tornando a valutare la condizione del ciclo.

Istruzione return

return [<expression>] ;

Quando l'istruzione return viene eseguita all'interno di un metodo l'esecuzione del metodo termina e questo restituisce il valore dell'espressione specificata.

4
Manipolazione degli oggetti

4.1 Creazione di un oggetto

```
Motocicletta m;              //1) l'oggetto viene dichiarato
m = new Motocicletta();      //2) l'oggetto viene istanziato
m.accendi();                 //3) l'oggetto viene utilizzato
```

```
m = new Motocicletta("Guzzi","bianca");
        //2) l'oggetto viene istanziato; il costruttore specifica anche i
valori iniziali dell'oggetto
```

```
public class Motocicletta {
String marca;
String colore;
boolean accesa = false;

public Motoclicletta(String m, String c){
        marca = m;
        colore = c;
}

public Motoclicletta(String marca, String colore, boolean
accesa){
        this.marca = marca;
        this.colore = colore;
        this.accesa = accesa;
}


// void accendi() { … }

// void mostra() { … }
}
```

4.2 Modi per assegnare un valore ad un oggetto

1) chiamata di costruttore
Motocicletta m2;
m = new Motocicletta(“Guzzi”,”bianca”);

2) assegno ad un oggetto il valore null
Motocicletta m2;
m2 =null;

3) assegno ad un oggetto un valore costante (non nullo) (c’è una chiamata di costruttore implicita)

```
String s = "Ciao";
int [] V = {1,2,3,4};
        // V = new int [4];
        // for (int i = 0; i < V.length; i++)
              V[i] = i;
```

4) assegno ad un oggetto il valore restituito da un metodo (che deve essere un oggetto compatibile)

```
 Motocicletta m2;
    m2 =Honda.metodoFactory(500);
```

5) assegno ad un oggetto il valore di un altro oggetto

```
Motocicletta m3;
m3 =m;
```

```
Motocicletta m;
m = new Motocicletta(“Guzzi”,”bianca”);
Motocicletta m3;
m3 =m;
System.out.println(m3.colore);    //
m3.colore= “rossa”;
System.out.println(m3.colore);   //
System.out.println(m.colore);     //
m3= null;
// System.out.println(m3.colore);   //
System.out.println(m.colore);     //
```

Nel caso sia stato specificato un costruttore si possono specificare argomenti.

Il modello ad oggetti utilizzato può essere considerato un modello ad oggetti per riferimento.

La principale caratteristica consiste nella allocazione delle istanze di una classe, od oggetti.

Una dichiarazione come Motocicletta m; ha come effetto non l'allocazione di un oggetto, ma l'allocazione di un riferimento (un puntatore) ad un oggetto.

Una istanza di oggetto va quindi creata esplicitamente utilizzando un appropriato metodo costruttore

La gestione della memoria è dinamica e automatica

In particolare, la variabile m viene allocata nello stack (pila), mentre l'oggetto verrà allocato nello heap al momento della sua creazione esplicita nel programma.

Oggetti inutilizzati vengono deallocati automaticamente tramite un meccanismo di garbage collection.

Per accedere ai campi di un oggetto e per richiamare un metodo si usa la notazione punto:

```
m.colore = "rossa";
System.out.println("Descrizione: ");
m.mostra();
```

il punto si valuta da sinistra a destra

4.3 Costanti

una costante può essere definita utilizzando la parola final usata per proprietà o variabili

Esempio

```
final double pi = 3.14

final int numeromesi = 12;
final int segnizodiacali = 12;
final int magliaportiere = 12;


final int iva = 21;
```

di una classe final non possono essere definite sottoclassi

un metodo final non può essere ridefinito nelle sottoclassi

4.4 Campi di classe

i campi di classe sono preceduti dalla parola chiave static

rappresentano proprietà che non sono specifiche dei singoli oggetti della classe, ma si riferiscono alla classe stessa

le proprietà statiche esistono e sono utilizzabili anche se non esistono oggetti della classe

sono accessibili con la sintassi:
NomeClasse.nomeproprietà
(o anche così: oggetto.nomeproprietà)

Rappresentano proprietà globali rispetto agli oggetti della classe

Esempio

```
class Famiglia {
        static String cognome = "Rossi";
        String nome;
        int eta;
}

// nel main()
System.out.println(Famiglia.cognome);

Famiglia padre = new Famiglia();
padre.nome=”Mario”;
padre.eta=40;


Famiglia madre = new Famiglia();
```

```
madre.nome=”Elena”;
madre.eta=40;

Famiglia figlio;
figlio.nome=”Carlo”;
figlio.eta=5;


padre.cognome=”Reds”;   // questo si può scrivere anche:
                        // Famiglia.cognome=”Reds”;
System.out.println(madre.cognome);
```

4.5 Metodi di classe

i metodi di classe sono preceduti dalla parola chiave static

utili per metodi che si applicano a una classe e non ad un singolo oggetto (nel senso che non lavorano con le proprietà degli oggetti) o
per raccogliere metodi con caratteristiche comuni (la classe Math di java.lang)

le metodi di classe esistono e sono utilizzabili anche se non esistono oggetti della classe

sono richiamabili con la sintassi:
NomeClasse.nomemetodo();
(o anche così: oggetto nomemetodo();)

un metodo di classe non può riferirsi a campi non di classe

Esempio

```
m = Math.max(a,b);
String s1,s2;
s1 = "ciao";
s2 = s1.valueOf(1);
s2 = String.valueOf(1);

class Famiglia {
        static String cognome = "Rossi";
        String nome;
        int eta;

void certificato()
        {
        System.out.println(cognome);
        System.out.println(nome);
        System.out.println(eta);
        }

}
```

“di classe” sinonimo di “statico”
“non statico” sinonimo di “di istanza” (cioè specifico dei singoli oggetti, cioè delle istanze della classe)

4.6 Confronto fra oggetti

Gli operatori == e != consentono di verificare se 2 oggetti puntano alla stessa locazione di memoria

m = new Motocicletta(“Guzzi”,”bianca”);
m3 = new Motocicletta(“Guzzi”,”bianca”);

m==m3

Il metodo equals() verifica se due stringhe sono uguali

```
String  s1,s2;
boolean b, b1, b2;

s1 = " ciao";
s2 = s1;
b = s1==s2;

s2 = new String(s1)
b1=s1==s2;
b2 = s1.equals(s2);
b2 = s1.equalsIgnoreCase(s2);


Motocicletta m1, m2, m3,m4;

m1 = new Motocicletta(“Honda”, “rossa”);
m2 = new Motocicletta(“Honda”, “rossa”);
m3 = new Motocicletta(“Guzzi”, “nera”);
m4 = m1;

m1 == m3          m1 == m2          m1 == m4
```

è possibile ridefinire in una classe il metodo equals() per considerare due oggetti uguali se hanno gli stessi valori delle proprietà

4.7 Verifica della classe di appartenenza ed introspezione

l'operatore instanceof ha due operandi: un oggetto e una classe; restituisce true se il valore dell'oggetto appartiene alla classe od a una delle sue sottoclassi, false altrimenti.

Si applica per verificare se il valore di un oggetto appartiene alla sua classe o ad una sottoclasse

Il metodo getClass() della classe Object restituisce un oggetto Class (a sua volta una classe): la classe dell'oggetto a cui è applicato

Il metodo getName() della classe Class restituisce il nome della classe

String nome = oggetto.getClass().getName();

public native boolean isInstance(Object obj)
restituisce true se obj è non null e si può fare un cast con la classe a cui il metodo è applicato

public native boolean isArray()
restituisce true se la classe a cui è applicato è un array

public native boolean isPrimitive()
restituisce true se la classe a cui è applicato è un tipo primitivo

public native Class getSuperclass()
restituisce la sovraclasse della classe a cui è applicato

public native Class[] getInterfaces()
determina le interfacce implementate dalla classe

public native int getModifiers()
restituisce i modificatori per la classe

il package java.lang.reflect contiene le definizioni delle seguenti classi:
Field, Method, Costructor, Modifier

public Field[] getFields()
restituisce il vettore dei campi pubblici dell'oggetto a cui è applicato

public Method[] getMethods()
restituisce il vettore dei metodi pubblici dell'oggetto a cui è applicato

public Constructor[] getConstructors()
restituisce il vettore dei costruttori pubblici dell'oggetto a cui è applicato

public Field getField(String name)
restituisce il campo specificato dal parametro

public Method getMethod(String name, Class parameterTypes[])
restituisce il metodo specificato dal parametro

public Constructor getConstructor(Class parameterTypes[])
restituisce il costruttore specificato dal parametro

Esempio

```
import java.lang.reflect.*;

class Motocicletta {
public String marca;
public String colore;
public boolean accesa = false;
public void accendi() { …}
public void mostra() {…}
}

 class specchio {
public static void main (String[] arguments)
        {
Motocicletta c = new Motocicletta();
        Class classe = c.getClass();
        System.out.println(classe.getName());
        Method [] metodi = classe.getMethods();
        Field [] campi = classe.getFields();
        for (int i =0; i < campi.length; i++)
                System.out.println(campi[i]);
        for (int i =0; i < metodi.length; i++)
                System.out.println(metodi[i]);
        }
}

Motocicletta

public java.lang.String Motocicletta.marca
public java.lang.String Motocicletta.colore
public boolean Motocicletta.accesa

public final native java.lang.Class java.lang.Object.getClass()
public native int java.lang.Object.hashCode()
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
public final native void java.lang.Object.wait(long)
public final void java.lang.Object.wait(long,int)
public final void java.lang.Object.wait()
public void Motocicletta.accendi()
public void Motocicletta.mostra()
```

4.8 Ereditarietà, interfacce e packages (2)

Quando vogliamo definire una nuova classe possiamo decidere se crearla da zero o partire da una classe già esistente; la classe già esistente viene modificata ed arricchita secondo le esigenze del programmatore.

E' possibile definire una nuova classe come sottoclasse discendente da una sovraclasse;

La sottoclasse eredita campi e metodi della sovraclasse (gli elementi privati esistono, ma non sono direttamente accessibili).

Esempio
public class Ciao2Applet **extends** java.applet.Applet

approccio di tipo bottom-up (da parti di codice (classi) più semplici ne costruisco di più complicate); favorisce il riuso del codice

se devo definire una serie di classi per la mia applicazione (per esempio voglio sviluppare una applicazione per la gestione dei dipendenti di una azienza; i dipendenti sono di vario tipo: impiegato, venditore, l'operaio, …) inveche che definire le classi in modo indipendente l'una dall'altra, posso oganizzarle in una gerarchia, definendo delle sovraclassi che rappresentano proprietà e servizi comuni a più sottoclassi (per esempio una sovraclasse Dipendente che rappresenta le proprietà comuni: nome, cognome, numeroMatricola,…)

aproccio di tipo top-down (definisco gli elementi più generici e gli elementi più specifici) favorisce il procedimento di astrazione.
Che forma assume la gerarchia delle classi di Java?
Un albero

Ogni classe può avere una sola sovraclasse (cioè Java usa l'ereditarietà singola);
per avere un liguaggio più semplice cioè semplifica la scrittura di compilatore ed interprete e quindi rende il linguaggio più sicuro.

Le interfacce sono collezioni di dichiarazioni di metodi che possono essere aggiunti alle classi;
una classe può estendere solo un'altra classe, ma può implementare tutte le interfacce che vuole
consentono di avere comportamenti analoghi in rami diversi della gerarchia delle classi.

class MiaApplet extends Applet, Thread, MouseAdapter: questo NON si può fare

class MiaApplet extends Applet implements Runnable, MouseListener: questo si può fare

Un package è una collezione di classi e interfacce. Consente di rendere disponibili gruppi di classi quando è necessario e consente di eliminare conflitti potenziali fra i nomi delle classi.

La libreria delle classi del Java Developer's Kit è contenuta nel package java, l'unico che è garantito presente in tutte le implementazioni.

Math
java.lang.Math

Per default sono disponibili solo le classi contenute nel package java.lang

Altre classi vanno riferite esplicitamente o importate: java.awt.Color

Esempio

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

// per importare tutto il package:
/*
import java.awt.*;
*/

public class Ciao2Applet extends java.applet.Applet {


Font f = new Font("TimesRoman",Font.BOLD,36);

public void paint (Graphics g) {
        g.setFont(f);
        g.setColor(Color.RED);
        g.drawString("Hello again!",5,25);
                                        }
}
```

4.9 La libreria delle classi

java.lang
: Object, String, System, Math, Thread,le classi dei tipi primitivi, …

java.util
: Date, Vector, Hashtable, …

java.awt
: Frame, Menu, Button, Font, CheckBox, Image, …

javax.swing
: JFrame, JMenu, JButton, JFont, JCheckBox, …

java.awt.event

java.applet

java.io

java.net
: ServerSocket, Socket, URL, …

4.10 Incapsulazione:

nascondere le parti interne di un oggetto,
permettendo l'accesso all'oggetto solo attraverso le interfacce definite

Un nome viene cercato in tutti i file sorgente possibile

Package: gruppo di classi collegate

Protezione package: metodi e campi sono accessibili alle classi nello stesso package

Le classi non insertite esplicitamente in un package vengono inderite nello stesso package di default

private: metodi e campi privati possono essere utilizzati o richiamati solo all'interno della stessa classe;

public: metodi e campi pubblici possono essere utilizzati o richiamati anche all'esterno della classe o del package;

: protezione a livello di package

protected: metodi e campi protetti possono essere utilizzati da tutte le classi del package e dalle sottoclassi anche fuori dal package; (*)

non è possibile definire due classi pubbliche nello stesso file

Visibilità	public	protected		private
nella stessa classe	si	si	si	si
da una classe nello stesso package	si	si	si	no
da una classe fuori dal package	si	no (si solo nelle sottoclassi)*	no	no

nelle sottoclassi:

un metodo pubblico rimane pubblico
un metodo protetto rimane protetto o può diventare pubblico
un metodo privato non viene visto
un metodo senza protezione può diventare privato

In generale si dichiarano i campi privati e si manipolano tramite metodi di accesso (pubblici) (setProprieta (tipo valore); getProprieta();)

Se eredito una proprietà privata, lei c'è, ma non la vedo. La posso manipolare utilizzando i motodi accesso setProprieta (tipo valore); getProprieta();

Nel momento in cui ho definito le intestazioni dei metodi pubblici per la manipolazione di un oggetto, ne posso modificare l'implementazione senza che questo influenzi il resto del programma

Esercizio

lista concatenata

```
class Elemento{
        int info;
        Elemento next;
}


class Lista {
        Elemento testa;
}
```

Lista è:
struttura vuota
elemento (la testa) a cui è associata una lista (la coda)

Lista	Array
Dinamico	dinamico
(la dimensione decisa a tempo	(la dimensione decisa a tempo
Di esecuzione)	Di esecuzione)
Posso inserire od estrarre	una volta che l'ho istanziato
Elementi	la dimensione rimane fissa
Accesso sequenziale	accesso casuale

4.11 Polimorfismo sui dati

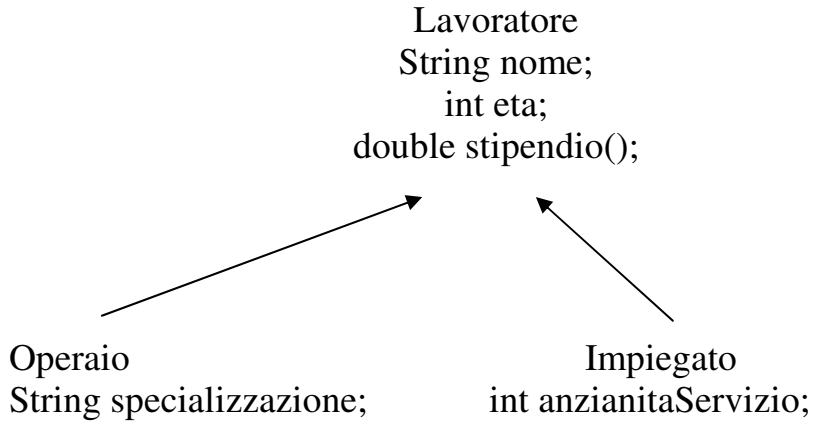
Un oggetto può assumere valori della sua classe, ma anche di sue sottoclassi.

(si può dichiarare un oggetto appartenente ad una interfaccia (invece che ad una classe); allora l'oggetto può assumere come valore quello di una qualsiasi classe che implementa quell'interfaccia)

Operaio
String nome;
int eta;
String specializzazione;
double stipendio();

Impiegato
String nome;
int eta;
int anzianitaServizio;
double stipendio();

Astrazione (evidenzio in una sovraclasse le caratteristiche comuni delle classi della mia applicazione):




```
Lavoratore l1, l2, l3;
Operaio o1, o2;
Impiegato i1,i2;

l1 = new Lavoratore();
o1 = new Operaio();
i1 = new Impiegato();

l2 = new Impiegato();
l3 = o1;
```

Questo mi permette anche di definire strutture dati polimorfe;

```
Lavoratore [] azienda;

azienda = new Lavoratore [100];
for (int i = 0; i < azienda.length/2; i++)
        azienda[i]=new Operaio();
for (int i = azienda.length/2; i < azienda.length; i++)
        azienda[i]=new Impiegato();
```

l3 dell'esempio sopra: esso è: un lavoratore oppure un operaio?
L'oggetto l3 viene trattato come un Lavoratore dal compilatore

Cioè il compilatore considera gli oggetti secondo la loro dichiarazione, il compilatore considera l3 un Lavoratore e **controlla** che l3 sia utilizzato conformemente al fatto di essere un Lavoratore

L'oggetto l3 viene trattato come un Operaio dall'interprete, ovvero in fase di esecuzione gli oggetti vengono trattati secondo il valore che essi assumono.

```
Lavoratore l1, l2, l3;
Operaio o1, o2;

o1 = new Operaio();
l3 = o1;                    //ok per il polimorfismo sui dati
l3.nome = "mario";  //ok perché per la classe Lavoratore è definita la proprietà nome
l3.specializzazione = "perito chimico";  //errore in compilazione: per la classe Lavoratore la proprietà specializzazione non è definita
```

4.12 Casting

L'operazione di casting converte un valore di un tipo primitivo in un altro tipo primitivo

(nometipo) valore

Esempio

```
float x=3.14, y=2.73;
int n = (int) (x/y);
int m = (int) (Math.random()*100);
```

```
int x=3, y=2;
double z = (double) x/y;
```

quando si assegna un valore a una variabile sufficientemente grande da accoglierlo senza perdita di precisione non c'è bisogno di un cast esplicito.
double x = 3;

Typecasting fra oggetti

Polimorfismo sui dati: un oggetto può assumere un valore che appartiene alla sua classe o a una sua sottoclasse.

Operazioni di casting sono possibili fra oggetti purché l'uno sia discendente dell'altro.

Si può usare un valore di una sottoclasse ogni volta che è richiesto il valore di un oggetto. (Non è richiesto un cast esplicito)

(Classe) oggetto

Cioè il compilatore considera gli oggetti secondo la loro dichiarazione

Lavoratore L;

per il compilatore L è un lavoratore; ma il programmatore può assegnare ad L un valore che appartiene ad uan delle due sottoclassi, Operaio od Impiegato.

Come fa il programmatore ad informate il compilatore che il valore di L non è un Lavoratore, ma un Impiegato?
Con un typecasting fra oggetti

```
Lavoratore L;
L = new Operaio();

L.nome = "Mario";
L.eta = 30;
// L.specializzazione = "Saldatore";   //errore in compilazione!
((Operaio) L).specializzazione = "Saldatore";
```

Nel typecasting di oggetti il programmatore indica (al compilatore) la classe del valore che l'oggetto contiene. In questo modo nei suoi controlli statici il compilatore non tiene conto della classe a cui si è dichiarato che l'oggetto appartiene ma della classe su cui abbiamo fatto il typecasting.

Se la classe non è compatibile (cioè non è una sottoclasse) con la classe dell'oggetto ci sarà lo stesso un errore in compilazione.

```
Operaio o2;
o2 = …
((Impiegato) o2).anzianitaServizio = 5; //errore in compilazione perché Impiegato non è una sottoclasse di Operaio
```

Se durante l'esecuzione il valore dell'oggetto non risulta compatibile con la classe su cui si è fatto il typecasting ci sarà un errore in esecuzione.

```
Lavoratore L2;
L2 = new Lavoratore ();

L2.nome = "Marco";
L2.eta = 35;
 ((Operaio) L2).specializzazione = "Saldatore";      //questo codice viene compilato correttamente; però questo codice produrrà un errore in esecuzione di typecasting scorretto
```

Conversione fra tipi

E' possibile fare conversioni fra i tipi primitivi e le classi corrispondenti

Esempi di conversione

```
String s1 = "120";
int n1 = (int) s1;   //non si può fare
int n1 = Integer.parseInt(s1);
float x = Float.parseFloat("3.14");
double y = Double.parseDouble("3.14");

String s = String.valueOf(35);
String s1 = Integer.toString(35);

Integer m = new Integer(35);
int n = m.intValue();
Integer p = new Integer(5);
int n = p+m ;    //ERRATO!! (in teoria, ma è possibile farlo nelle più recenti versioni di Java e quindi n vale 40)
```

4.13 Definizione di una classe

class Classe1
{. . . }

public class ClasseFiglia extends Classe1 implements Interfaccia1, Interfaccia2
{. . . }

4.14 Definizione dei metodi

un metodo è caratterizzato da:
nome, i parametri (segnatura, firma) (signature)

modificatori:
public, protected, , private,
static
final

throws

modificatori tipovalorerestituito nomeMetodo (tipo1 arg1, tipo2 arg2, …) {. . . }

Esempio

```
class Range {


int [] makeRange (int min, int max) {
        int vet [] = new int [max - min+1];      // max-(min-1)
        for (int i = 0; i < vet.length; i++) {
                vet[i] = min++;
        }
        return vet;
}

public static void main (String args[]) {
        int [] vet;
        Range r ;
        r= new Range();
        vet = r.makeRange(1,10);

        System.out.print("vet = [");
        for (int i = 0; i < vet.length; i++)
                System.out.print(vet[i] +" ");
        System.out.println("]");


int estremominore = (int) (Math.random()*20);
int ampiezza = (int) (Math.random()*15)+1;
vet = r.makeRange(estremominore, estremominore+ampiezza);

        System.out.print("vet = [");
        for (int i = 0; i < vet.length; i++)
                System.out.print(vet[i] +" ");
        System.out.println("]");

        }
}
```

4.15 La parola chiave this

La parola chiave this rappresenta un parametro implicito per i metodi di istanza (cioè non statici)

La parola chiave this permette di riferirsi all'oggetto corrente (cioè all'oggetto a cui il metodo è applicato o all'oggetto che un metodo costruttore restituirà)

Metodi di classe non hanno l'identificatore this implicito perché…

Esempio

```
this.nome = nome;
this.metodo();
oggetto.metodo(this);
return this;
```

4.16 Visibilità

Le variabili locali sono visibili nei blocchi in cui sono definiti e nei blocchi annidati (non è possibile ridefinire una variabiloe locale in un blocco annidato)

Campi e metodi sono visibili nella classe in cui sono definiti e nelle sottoclassi (purché nopn siano privati)

Dichiarazioni omonime di variabili mascherano le dichiarazioni dei campi

Data una variabile se ne cerca la definizione nel blocco corrente, poi in quello appena più esterno e così via;
se non è una variabile locale si cerca la definizione come campo della classe corrente o di una sovraclasse

Esempio

```
class Vi {
        int a = 10;
        int b = 40;
        static int c = 50;

void scrivi() {
        int a = 20;
        System.out.println(a);
        System.out.println(this.a);
        System.out.println(b);
        System.out.println(c);
}

public static void main (String args[]) {
        int a = 30;
        Vi r = new Vi();
        r.scrivi();
        System.out.println(a);
        System.out.println(c);
        System.out.println(this.a);

        }
}
```

4.17 Passaggio dei parametri

In Java i parametri sono passati per valore

Un parametro corrisponde ad una variabile locale del metodo.

Quando un metodo viene richiamato:

1) i parametri attuali (cioè quelli su cui viene richiamato effettivamente il metodo) vengono valutati
2) i valori ottenuti vengono assegnati al corrispondente parametro formale

Il modello di oggetti per riferimento fa si che, mentre i puntatori agli oggetti sono anche essi passati per valore,
gli oggetti puntati (cioè l'area di memoria che contiene le proprietà dell'oggetto) sono passati per riferimento.
Lo stesso vale per i vettori, essendo anche essi degli oggetti.

Esempi

```
class  Prova {

int m (int p, int q)
{
p=p+1;
q=q+1;
return p+q;
   }

public static void main (String args[]) {
        int a = 1;
        int b = 2;
        int c;
        Prova p = new Prova();
        c=p.m(a,b);
        System.out.println(a);    //
        System.out.println(b);    //
        System.out.println(c);    //
        c=p.m(b-1,1+1);
        System.out.println(a);   //
        System.out.println(b);   //
        System.out.println(c);   //
```

```
}
}
```

```
class  Persona {
String nome;
int eta;


public Persona (String nome, int eta){
        this.nome = nome;
        this.eta = eta;
}

  public void stampa()
   {
   System.out.println("Mi chiamo "+nome);
   System.out.println("ho "+eta+" anni");
   }

}
```

```
public class Applicazione {

  public static void main(String argv[])
  {
  Persona lui, lei;
  lui= new Persona("Marco", 25);
  lei= new Persona("Anna", 30);

  compleanno(lui);

  lui.stampa();       //
  lei.stampa();       //

  scambia(lui,lei);
  lui.stampa();       //marco
  lei.stampa();       //anna

  scambia1(lui,lei);
  lui.stampa();       // anna
  lei.stampa();       //marco
  }

public static void compleanno(Persona p1)
   {
   p1.eta++;
 }
```

```
public static void scambia(Persona p1, Persona p2)
  {
   Persona p3;
   p3 = p1;
   p1 = p2;
   p2=p3;
 }


public static void scambia1(Persona p1, Persona p2)
  {

   String nome = p1.nome;
    int eta = p1.eta;

   p1.nome = p2.nome;
   p1.eta = p2.eta;

   p2.nome = nome;
   p2.eta = eta;
 }

}
```

Ricapitolando:
In Java i parametri sono passati per valore
Siccoma Java usa un modello di oggetti per riferimento un metodo può

1) modificare le proprietà dell'oggetto che gli viene passato
2) modificare le proprietà dell'oggetto a cui viene applicato (passando implicitamente o esplicitamente per il parametro this)

```
class PassByReference {

int oneToZero(int arg[]) {
      int count = 0;
        for (int i = 0; i < arg.length; i++) {
           if (arg[i] == 1) {
              count++;
              arg[i] = 0;
           }
        }
        return count;
    }

static void visualizza (int vettore []){
       System.out.print("Values of the array: [ ");
       for (int i = 0; i < arr. vettore; i++) {
          System.out.print(vettore [i] + " ");
       }
       System.out.println("]");
}

 public static void main (String arg[]) {

       PassByReference test = new PassByReference();
       int numOnes;
      arg[] = { 1, 3, 4, 5, 1, 1, 7 };

       visualizza(arr);
      numOnes = test.onetoZero(arr);
      System.out.println("Number of Ones = " + numOnes);
      visualizza(arr);  // [0,3,4,5,0,0,7]

    }
}
```

Altri linguaggi ammettono anche il passaggio per riferimento (nel linguaggio c per esempuio usando &). In questo caso il parametro attuale deve essere una variabile ed il parametro formale diventa n secondo riferimento alla stessa locazione di memoria a cui si riferisce il parametro attuale.

4.18 Applicazioni Java

Una applicazione consiste di una o più classi.

Esattamente una classe deve contenere il metodo main:

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

Le altre classi possono trovarsi in file nella stessa directory della classe principale o nelle directory elencate in CLASSPATH

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

4.19 Argomenti della linea comandi

Gli argomenti passati nella linea comandi diventano le componenti del parametro args del metodo main

Esempi

```
class EchoArgs {
   public static void main(String args[]) {
         System.out.println("Numero parole immesse = "  +args.length);
       for (int i = 0; i < args.length; i++) {
          System.out.println("Argument " + i + ": " + args[i]);
       }
   }
}
```

==========

```
class SumAverage {
   public static void main (String args[]) {
      int sum = 0;

      for (int i = 0; i < args.length; i++) {
                        sum += Integer.parseInt(args[i]);
      }

      System.out.println("Sum is: " + sum);
      System.out.println("Average is: " +
         (float)sum / args.length);
   }
}
```

4.20 Overloading di metodi (sovraccarico)

E' possibile, all'interno della stessa classe, creare metodi con lo stesso nome, ma con differenti intestazioni e definizioni.
Il numero ed il tipo degli argomenti sono utilizzati per distinguere i metodi.

```
public static void quicksort  (int v[]) {
        quicksort(v,0,v.length-1);
}

private static void quicksort  (int v[], int inizio, int fine) {…}
```

Esempio

```
import java.awt.Point;

class MyRect {
   int x1 = 0;    int y1 = 0;   int x2 = 0;    int y2 = 0;

   MyRect buildRect(int x1, int y1, int x2, int y2) {
      this.x1 = x1;        this.y1 = y1;        this.x2 = x2;        this.y2 = y2;
      return this;     }

   MyRect buildRect(Point topLeft, Point bottomRight) {
      x1 = topLeft.x;        y1 = topLeft.y;        x2 = bottomRight.x;
      y2 = bottomRight.y;        return this;      }

   MyRect buildRect(Point topLeft, int w, int h) {
      x1 = topLeft.x;        y1 = topLeft.y;        x2 = (x1 + w);        y2 = (y1 + h);
      return this;     }

   MyRect buildRect(Point topLeft, int lato) {
      x1 = topLeft.x;        y1 = topLeft.y;        x2 = (x1 + lato);        y2 = (y1 +
lato);
      return this;     }

   MyCircle buildRect(Point topLeft, int raggio) {
      x1 = topLeft.x;        y1 = topLeft.y;        x2 = (x1 + raggio);        y2 = (y1
+ raggio);
      return this;     }  //QUESTO NON SI PUO' FARE: C'E' AMBIGUITA'
CON QUELLO SOPRA


  void printRect(){
      System.out.print("MyRect: <" + x1 + ", " + y1);
      System.out.println(", " + x2 + ", " + y2 + ">");                  }

   public static void main(String args[]) {
      MyRect rect = new MyRect();

      System.out.println("Calling buildRect with coordinates 25,25 50,50:");
      rect.buildRect(25, 25, 50, 50);        rect.printRect();

      System.out.println("Calling buildRect w/points (10,10), (20,20):");
      rect.buildRect(new Point(10,10), new Point(20,20));
```

```
    rect.printRect();

    System.out.print("Calling buildRect w/1 point (10,10),");
    System.out.println(" width (50) and height (50)");
    rect.buildRect(new Point(10,10), 50, 50);       rect.printRect();
}}
```

4.21 Costruttori

Un costruttore è un metodo richiamato per istanziare un nuovo oggetto.
Se il programmatore non definisce propri costruttori ha a disposizione un costruttore di default senza parametri.

Il costruttore può essere usato anche per inizializzare le proprietà del nuovo oggetto.

Un costruttore ha lo stesso nome della classe e non restituisce valori.

Esempio

```
class Person {
   String name;
   int age;

  Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   void descrivi(){
      System.out.print("Hi, my name is " + name);
      System.out.println(". I am " + age + " years old.");
   }

 public static void main (String args[]) {
    Person p;
    p = new Person("Laura", 20);
    p. descrivi ();
    System.out.println("--------");
    p = new Person("Tommy", 3);
    p. descrivi ();
    System.out.println("--------");
    Person q;
    q = new Person("Anna", 30);
    q. descrivi ();
    System.out.println("--------");

  }
}
```

Quando definiamo una sottoclasse e definamo un costruttore della sottoclasse è necessario (come prima istruzione) richiamare il costruttore della sovraclasse con la seguente sintassi:
super(arg1, arg2, …)

```
class Studente extends Person {
        String facolta;
        int matricola;

Studente (String name, int age, String f, int matricola) {
      super(name, age);
      facolta = f;
        this.matricola = matricola;
   }

class  Informatico extends Studente {
        double media;

        Informatico (String name, int age, String f, int m, double media) {
                super(name, age,  f,  m);
                this.media = media;
        }
```

E' possibile fare overloading di costruttori:

```
import java.awt.Point;

class MyRect2 {
   int x1 = 0;    int y1 = 0;    int x2 = 0;    int y2 = 0;

   MyRect2(int x1, int y1, int x2, int y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
   }


   MyRect2(Point topLeft, int w, int h) {
      x1 = topLeft.x;
      y1 = topLeft.y;
      x2 = (x1 + w);
      y2 = (y1 + h);
   }
```

E' possibile richiamare il costruttore della classe all'interno di un altro costruttore con la seguente sintassi:
this(arg1, arg2, …)

4.22 Ridefinizione di metodi (overriding)

E' possibile ridefinire in una sottoclasse un metodo definito in una sovraclasse

Esempio

```
class Person {
   String name;
   int age;

  Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   void descrivi(){
      System.out.print("Hi, my name is " + name);
      System.out.println(". I am " + age + " years old.");
   }



class Studente extends Person {
        String facolta;
        int matricola;

Studente (String name, int age, String f, int m) {
      super(name, age);
      facolta = f;
        matricola = m;
```

```
/*
   void descrivi(){
      System.out.print("Hi, my name is " + name);
      System.out.println(". I am " + age + " years old.");
      System.out.println("La mia Facoltà: " + facolta);
      System.out.println("Matricola: " + matricola);

   }
*/
   void descrivi(){
      super.descrivi();
      System.out.println("La mia Facoltà: " + facolta);
      System.out.println("Matricola: " + matricola);

   }


    public static void main (String args[]) {
    Person p;
    p = new Person("Laura", 20);
    p.descrivi ();
    System.out.println("--------");
    Studente s  = new Studente("Tommy", 23, “Scienze”, 12345);
    s.descrivi ();
    System.out.println("--------");

Person p1;
p1= new Studente("Anna", 23, “Scienze”, 12345);
p1.descrivi();  //viene eseguito il metodo descrivi degli studenti
  }

 }
```

4.23 Metodi finalizzatori

La deallocazione dei metodi avviene automaticamente tramite la garbage collection.

Se vuole il programmatore può richiamare esplicitamente la garbage collection:

System.gc();

Prima che un oggetto sia deallocato viene eseguito automaticamente il metodo finalize().

Esso può essere ridefinito (di solito allo scopo di effettuare operazioni di liberazione di risorse collegate con l'oggetto):

```
protected void finalize () throws Throwable {
        //istruzioni
        for (int i = 0; i < dati.length; i++)
                dati[i].finalize();
        super.finalize();
}
```

È consigliabile deallocare esplicitamente oggetti che occupano risorse di sistema richiamando il metodo **dispose**()

Esempio:

```
Graphics g;
Button b = new Button("ok");
g = b.getGraphics();
…
g.dispose();
```

Esempio (nell'esercizio delle liste):

```
public class Elemento {
   int valore;
   Elemento next;

   public Elemento(int valore,Elemento next){
      this.valore=valore;
      this.next=next;
   }
   protected void finalize () throws Throwable {
      System.out.println("finalizzazione dell'elemento "+valore);
   }
}
```

```
Lista l=new Lista(30);
System.out.println("Cancella 2 =" + l.cancellaEle(2));
System.gc();

 l2 = l.listaInversa();
 l2.visualizza();
 l2 = l.listaInversa(); //queste chiamate danno tempo alla garbage
                        // collection di terminare
l2.visualizza();
```

4.24 Chiamata dei metodi (polimorfismo)

Un oggetto può assumere valori della sua classe o di sue sottoclassi (polimorfismo sui dati)

Early binding: il compilatore decide quale metodo chiamare in base alla classe a cui si è dichiarato che l'oggetto appartiene

Late binding: l'interprete decide quale metodo chiamare in base alla classe a cui appartiene il valore dell'oggetto a cui il metodo viene applicato (polimorfismo sui metodi)

In Java:
metodi di istanza ←→ late binding

metodi statici (di classe) ←→ early binding

metodi di classi final ←→ early binding

metodi finali ←→ early binding

In altri linguaggi:
metodi virtuali (dinamici) ←→ late binding

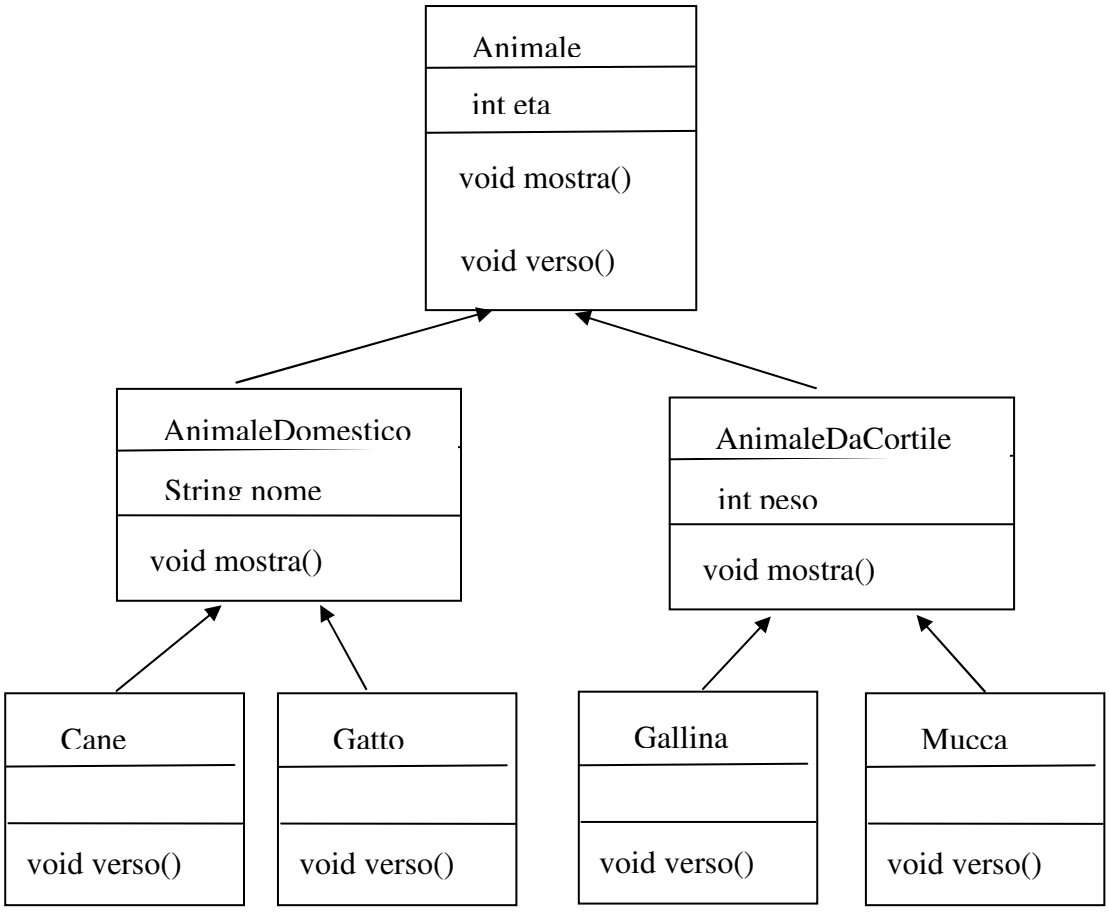
metodi statici ←→ early binding

In Java il compilatore effettua comunque un controllo statico (cioè basato sulle dichiarazioni) sulla chiamata del metodo, verificando che il metodo sia definito nella (o ereditato dalla) classe a cui abbiamo dichiarato che l'oggetto appartiene.
Il compilatore controlla in modo analogo i riferimenti ai campi.

Esempio

```
class Animale {
   int eta;
   void mostra(){… }
  void verso(){… }
}


 class AnimaleDaCortile extends Animale{
   int peso;
   void mostra(){… }
}

class Gallina extends AnimaleDaCortile{
   void verso(){… }
}

class Mucca extends AnimaleDaCortile{
   void verso(){… }
}

  public static void main(String [] arg) {
       Animale V[];
       V = new Animale[4];


       V[2]= new Gallina();
       V[2].eta=6;


       ((Gallina) V[2]).peso=6; //Assegnamento corretto
       ((java.awt.Button) V[2]).peso=6; //Assegnamento scorretto: errore in
compilazione
       ((Mucca) V[2]).peso=6; //Assegnamento scorretto: errore in esecuzione
```

Esempio



Voglio lavorare con 4 classi che rappresentano degli animali: Cane, Gatto (nome), Gallina, Mucca (peso), per tutte 4 voglio rappresentare anche l'età. Per tutte 4 voglio definire un metodo che produce la descrizione e un metodo che produce il verso.
Voglio realizzare una struttura dati polimorfa in cui inserire tutti gli animali della mia Fattoria.

Quindi ho bisogno di una sovraclasse di quelle 4: La classe Animale: in questo modo posso realizzare la struttura dati polimorfa;
Astrazione: definire i dati a vari livelli in cui ad ogni livello rappresentiamo caratteristiche comuni di gruppi di oggetti (in questo modo possiamo definire una sola proprietà o un solo metodo che viene ereditato da un gruppo di oggetti)

```
abstract class Animale {
    int eta;
    void mostra(){
       System.out.println("io ho "+eta+"anni");
       }

    Animale (int e){
            eta = e;
            }

     abstract void verso();
}
```

```
abstract class AnimaleDomestico extends Animale{
   String nome;

      AnimaleDomestico(String nome, int e){
            super(e);
            this.nome = nome;
      }

   void mostra(){
   System.out.println("mi chiamo "+nome);
   //System.out.println("e ho "+eta+" anni");
      super.mostra();
   }
}

abstract class AnimaleDaCortile extends Animale{
   int peso;

      AnimaleDaCortile(int e, int p){
            super(e);
            peso=p;
      }

   void mostra(){
   super.mostra();
   System.out.println("e peso "+peso+" chili");
   }
}
```

```
class Cane extends AnimaleDomestico{

      Cane(String nome, int eta){
            super(nome, eta);
      }

    void verso(){
   System.out.println("Bau! ");
   }
}

class Gatto extends AnimaleDomestico{

      Gatto(String nome, int eta){
            super(nome, eta);
      }

   void verso(){
   System.out.println("Miao! ");
   }
}
```

```
class Gallina extends AnimaleDaCortile{

      Gallina(int eta, int peso){
            super(eta,peso);
      }

   void verso(){
   System.out.println("Coccode! ");
   }
}
class Mucca extends AnimaleDaCortile{

      Mucca(int eta, int peso){
            super(eta,peso);
      }
   void verso(){
   System.out.println("Muuuuu! ");
   }
}
```

```
class Zoo {

      Animale V [];

   /** Creates a new instance of Zoo */
   public Zoo(int n){

      V= new Animale [n];
// Animale[0] = new Animale(10);  // NO perché
animale è astratto
                  for (int i = 0; i < V.length; i++)
                        if (i%4 == 0)
                              V[i] = new Cane("Cane"+i ,
(int) (Math.random()*10));
                        else       if (i%4 == 1)
                              V[i] = new Gatto("Gatto"+i,
(int) (Math.random()*10));

                        else       if (i%4 == 2)
                              V[i] = new Gallina((int)
(Math.random()*10),(int) (Math.random()*3));


                        else       V[i] = new Mucca((int)
(Math.random()*10),(int) (Math.random()*100));

            }
```

```
void alloZoo() {
            for (int i = 0; i < V.length; i++)
                  V[i].verso();
      }

      void mostra(){
            for (int i = 0; i < V.length; i++)
                  V[i].mostra();
      }

   }



public class ApplicazioneZoo {


public static void main(String [] arg) {

                  Zoo z;
                  z = new Zoo(10);
                  z.alloZoo();
                  z.mostra();
      }

}
```

4.25 Classi e metodi astratti

abstract: una classe astratta ha lo scopo di definire caratteristiche comuni alle sottoclassi; non è possibile istanziare oggetti di una classe astratta

metodi astratti hanno intestazione, ma non implementazione. Vengono implementati nelle sottoclassi

metodi astratti possono essere definiti solo in classi astratte