Parte quarta: la programmazione orientata agli oggetti (OOP) e i suoi concetti

Ci addentriamo in un argomento molto importante che rappresenta, in poche parole, l'essenza stessa del linguaggio Java: la programmazione orientata agli oggetti. Finora abbiamo visto come creare variabili, inizializzarle, creare array e matrici, utilizzare le strutture di controllo, le iterazioni, le eccezioni ecc. Per comprendere bene la parte essenziale che stiamo per prendere in considerazione è però necessario conoscere cos'è un oggetto, una classe ed un'istanza. Quindi, prima di cominciare con questa parte, se non si ricordano le nozioni basilari sugli oggetti, sarà bene rileggere il paragrafo 3 della prima parte ("Richiami sugli oggetti"). Detto ciò, possiamo cominciare.

1. Classi, attributi, metodi

Poiché tutta l'attività di programmazione Java avviene all'interno di una classe, è importante conoscere a fondo i loro meccanismi di funzionamento.
Una classe è un modello che definisce la forma di un oggetto. Java utilizza una specificazione delle classi per creare gli oggetti, che sono istanze di una classe. La struttura di base di una classe è la seguente:

```
class <nome>
{
   <attributi>
   <metodi>
}
```

A titolo di esempio, creiamo una nuova classe, la classe *Persona*, per comprendere meglio i concetti di attributi e metodi:

```
class Persona
{
         public String nome;
         public String cognome;
         public int eta;
         private double statura;

      public void impostaEta(int age)
      {
         eta = age;
      }

      public int ottieniEta()
      {
         return eta;
      }
}
```

Nella classe Persona nome, cognome, eta e statura sono attributi mentre impostaEta() e ottieniEta() sono metodi. Analizziamo adesso gli attributi.

Attributi: gli attributi devono essere dichiarati nel blocco di una classe ed esternamente ai metodi. Gli attributi vengono anche chiamati variabili istanza, perché ogni oggetto (che è l'istanza di una classe) possiede le proprie variabili, nelle quali memorizza i valori dei propri attributi. Gli attributi vengono dichiarati nello stesso modo in cui vengono dichiarate le semplici variabili. Un attributo viene quindi generalmente dichiarato nel modo seguente:

```
[<livello_visibilità>] <tipo> <nome>;
```

L'indicatore <livello_visibilità> indica il livello di visibilità di un attributo. Un attributo stabilisce un livello di visibilità, ovvero quali sono le classi che possono accedere all'attributo per leggere o modificare il suo valore. Di seguito sono elencati i tre specificatori di accesso che vengono utilizzati: **public**, **private** e **protected**.

`public`	l'attributo è accessibile a qualsiasi altra classe.
`private`	l'attributo non è accessibile ad altre classi e solo la classe che lo contiene può utilizzarlo o modificarne il valore.
`protected`	l'attributo è accessibile alle classi del medesimo package e alle sottoclassi (anche di altri package) della classe che contiene l'attributo.

Riferendoci all'esempio visto sopra, gli attributi nome, cognome ed eta sono public, e possono essere "visti" dalle classi esterne, mentre statura, essendo private, può essere manipolato solo dalla classe Persona e da nessun'altra. Vediamo ora i metodi.

Metodi: un metodo è caratterizzato da un livello di visibilità, un tipo di valore di ritorno, un nome, un elenco di parametri e un blocco di istruzioni. La forma generale di un metodo è quindi la seguente:

```
[<livello_visibilità>] <tipo_val_ritorno> <nome> ([<parametri>])
{
     <istruzioni>
}
```

Un metodo che non restituisce alcun valore di ritorno è un metodo che utilizza la parola chiave **void**. Per restituire un valore di ritorno si utilizza l'istruzione **return** che, oltre a restituire un valore di ritorno, chiude l'esecuzione del metodo. Si può ricorrere all'istruzione return quando si vuole uscire dal metodo prima della sua fine. Ad esempio, la classe Persona utilizza i metodi impostaEta() e ottieniEta(). Il metodo impostaEta() è un metodo di tipo void e, pertanto, non restituisce alcun valore di ritorno mentre il metodo ottieniEta() restituisce come valore di ritorno un numero intero.

Ogni classe può essere rappresentata graficamente in **UML** (Unified Modelling Language). UML è un metodo di analisi e progettazione ad oggetti.

[NOTA: un'analisi approfondita di UML si può trovare nell'appendice B alla fine di questo corso. Per adesso non useremo UML per realizzare diagrammi di classi.]

2. Variabili istanza

Le variabili istanza contengono i valori degli attributi e possono assumere diversi valori all'interno di ogni istanza e, inoltre, permangono anche dopo l'esecuzione di un metodo di quell'istanza.

3. Creazione di oggetti

Esaminiamo adesso come avviene il meccanismo di creazione di un oggetto.

1) **Dichiarazione**: per dichiarare un oggetto si utilizza la seguente sintassi:

```
<nome_classe> <nome_var_oggetto>;
```

Dove <nome_classe> è il nome della classe (ad esempio Persona) mentre <nome_var_oggetto> è il nome della variabile che si desidera associare ad un oggetto. Per esempio, per dichiarare un oggetto di classe Persona:

```
Persona p;
```

La variabile 'p' è quindi associata ad un oggetto di classe Persona.

2) **Allocazione:** per allocare un oggetto si utilizza la seguente sintassi:

```
<nome_var_oggetto> = new <nome_costruttore> ([parametri]);
```

In Java il nome del metodo costruttore deve essere necessariamente corrispondente con il nome della classe. Per esempio:

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

Le due istruzioni viste sopra possono essere raggruppate in un'unica istruzione, avente la seguente generica sintassi:

```
<nome_classe> <nome_var_oggetto> = new <nome_costruttore> ([parametri]);
```

Per esempio:

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

Quindi, quando un oggetto viene dichiarato viene creata una variabile di dimensione tale da contenere un riferimento ad un oggetto di una specifica classe; quando un oggetto viene allocato viene creato in memoria lo spazio necessario a contenere l'oggetto con tutti i suoi attributi e inserisce nella variabile un riferimento a tale spazio di memoria. L'operatore new alloca spazio fisico per un oggetto.

4. Creazione di metodi costruttori

Tornando al primo esempio, creiamo il costruttore della classe Persona:

```
class Persona
{
         public String nome;
         public String cognome;
         public int eta;
         private double statura;

      public Persona(String n)            // costruttore
      {
         nome = n;
         eta = 0;
      }
}
```

Dove n rappresenta il nome da specificare per l'oggetto di classe Persona che viene creato. Quindi per creare un nuovo oggetto Persona scriveremo:

```
Persona p = new Persona("Tizio");
```

Il livello di visibilità per i metodi costruttori è public. Se una classe viene creata senza specificare il metodo costruttore, ad essa viene associato un costruttore vuoto e al momento della creazione dell'oggetto non viene eseguita alcuna operazione. Se un costruttore viene dichiarato come private non sarà possibile creare istanze. Proviamo a scrivere:

```
private Persona(String n)
{
   nome = n;
   eta = 0;
}
```

Se proviamo a compilare, il compilatore ci segnalerà questo errore:

```
test.java:4: Persona(java.lang.String) has private access in Persona
                Persona p = new Persona("Tizio");
                            ^
1 error
```

Prendiamo ad esempio la classe Math del package java.lang, che considereremo più avanti nel corso: se tentiamo di creare un oggetto di classe Math

```
Math m = new Math();
```

il compilatore ci segnalerà questo errore:

```
test.java:4: Math() has private access in java.lang.Math
                Math m = new Math();
                         ^
1 error
```

Come vedremo in seguito, la classe Math non necessita di istanze, in quanto è composta da metodi ed attributi statici. Fra poco considereremo proprio attributi e metodi statici. Per ora ci occorre solamente sapere che questi attributi e metodi possono essere richiamati indipendentemente dall'esistenza o meno di un'istanza di una classe.

Tornando al discorso precedente, la variabile p conterrà un riferimento, o indirizzo o puntatore, allo spazio di memoria in cui sono allocati i valori degli attributi.
Un'istanza di una classe può contenere o un riferimento ad un oggetto oppure un valore particolare, indicato dalla parola chiave **null**, con la quale non si fa riferimento ad alcun oggetto.

Possiamo anche aggiungere più metodi costruttori nella stessa classe, a condizione che abbiano tutti diverso numero o tipo di parametri. Per esempio:

```
class Persona
{
         public String nome;
         public String cognome;
         public int eta;
         private double statura;

      public Persona(String n)
      {
         nome = n;
         eta = 0;
      }

      public Persona(String n, String c)
      {
         nome = n;
         cognome = c;
      }

      public Persona(String n, String c, int e)
```

```
        {
            nome = n;
            cognome = c;
            eta = e;
        }
}
```

Un'ultima precisazione: relazionando variabili con oggetti, possiamo affermare che alle variabili viene associato un tipo, agli oggetti, invece, una classe.

Classi nello stesso file o in file separati? Un file è capace di contenere più classi, ma solo una di esse potrà contenere il metodo main. La classe principale è la classe che contiene main(). La classe principale dovrà essere preceduta dalla parola chiave public.

5. Interazione fra oggetti

Gli oggetti possono tra loro interagire. Vediamo i meccanismi delle loro comunicazioni.

Invocare un metodo: per invocare un metodo si utilizza la seguente forma generale:

```
<nome_oggetto>.<nome_metodo> ([parametri]);
```

Ad esempio, sempre considerando la classe Persona, se volessimo impostare l'età per l'oggetto 'p' scriveremo:

```
p.impostaEta(18);
```

Abbiamo inoltre passato al metodo impostaEta() un parametro, il numero 18, che indica l'età di Persona.
[NOTA: per essere richiamati, i metodi devono possedere un livello di visibilità public. Se un metodo viene dichiarato private non può essere richiamato nel modo visto sopra.]

Accedere agli attributi: per accedere direttamente ad un attributo è sufficiente usare la stessa notazione usata per invocare un metodo:

```
<nome_oggetto>.<nome_attributo>;
```

Ad esempio:

```
p.nome;
```

Gli attributi possono anche essere modificati direttamente. Il seguente esempio mostra come modificare gli attributi di p:

```
Persona p = new Persona("Tizio");
p.eta = 18
p.cognome = "Caio";
```

Esiste tuttavia un modo più "pulito" per modificare il valore degli attributi: si aggiunge un nuovo metodo set o get per ogni attributo. set imposta il valore di un attributo mentre get ne ottiene il valore. Ad esempio, avremmo potuto inserire nella classe Persona dei metodi per modificare gli attributi, precisamente setEta(), getEta(), setNome(), getNome(), setCognome(), getCognome(), setStatura() e getStatura(). Nell'esempio, l'attributo statura ha accesso privato, quindi solamente Persona, la classe che lo contiene, può leggerne o modificarne il valore. Proviamo infatti a modificarne il valore:

```
p.statura = 180.5;
```

Quando compileremo, il compilatore ci segnalerà questo errore:

```
test.java:6: statura has private access in Persona
          p.statura = 180.5;
            ^
1 error
```

Il compilatore ci segnala che statura ha un livello di accesso privato. Se proviamo a cercare di leggerne il valore:

```
System.out.println(p.statura);
```

Il compilatore ci segnalerà ancora errore:

```
test.java:10: statura has private access in Persona
        System.out.println(p.statura);
                           ^
1 error
```

Il segnale di errore del compilatore ci indica ancora che statura ha livello di accesso privato. Tuttavia, possiamo leggere ed impostare il valore di questa variabile anche se il livello di accesso è privato, utilizzando i metodi setStatura() e getStatura(). Apportando tutte le modifiche viste finora, vediamo la nuova classe Persona:

```
class Persona
```

```
{
        public String nome;
        public String cognome;
        public int eta;
        private double statura;


    public Persona()
    {

    }

    public Persona(String n)
    {
        nome = n;
        cognome = "";
        eta = 0;
    }

    public Persona(String n, String c)
    {
        nome = n;
        cognome = c;
        eta = 0;
    }

    public Persona(String n, String c, int e)
    {
        nome = n;
        cognome = c;
        eta = e;
    }

    public void setNome(String name)
    {
        nome = name;
    }

    public String getNome()
    {
        return new String(nome);
    }

    public void setCognome(String surname)
    {
        cognome = surname;
    }

    public String getCognome()
    {
        return new String(cognome);
    }

    public void setEta(int age)
    {
        eta = age;
    }

    public int getEta()
    {
        return eta;
    }

    public void setStatura(double height)
    {
        statura = height;
```

```
      }

      public double getStatura()
      {
         return statura;
      }
}
```

Per provare, creiamo il file test.java ed inseriamo il codice seguente:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();       // nessun parametro

         p.setNome("Tizio");
         p.setCognome("Caio");
         p.setEta(45);
         p.setStatura(165.5);

                                          // utilizziamo i metodi get per
                                          // ottenere i valori impostati

         System.out.println("Nome: " + p.getNome());
         System.out.println("Cognome: " + p.getCognome());
         System.out.println("Eta': " + p.getEta());
         System.out.println("Statura: " + p.getStatura());
      }
}
```

Abbiamo visto che, nonostante statura sia un membro ad accesso privato, abbiamo potuto modificarlo e leggerlo tramite i metodi setStatura() e getStatura(), che hanno accesso pubblico. Quindi, utilizzando questi meccanismi, potremmo creare delle classi che possiedono solo attributi ad accesso privato, che possono essere modificati unicamente attraverso metodi. Questo è un aspetto importante di Java. Immaginiamo ad esempio una classe che possieda un attributo molto importante da definire. Per definire questo attributo è utile effettuare dei controlli. Se tale attributo può essere definito attraverso l'accesso diretto, non si avrebbe nessun genere di controllo e, se venisse inserito un valore errato, comprometterebbe la funzionalità del programma stesso. Quindi, in tal caso, è molto utile l'utilizzo di metodi specifici. A titolo esemplificativo, creiamo una classe che funzioni in modo analogo.

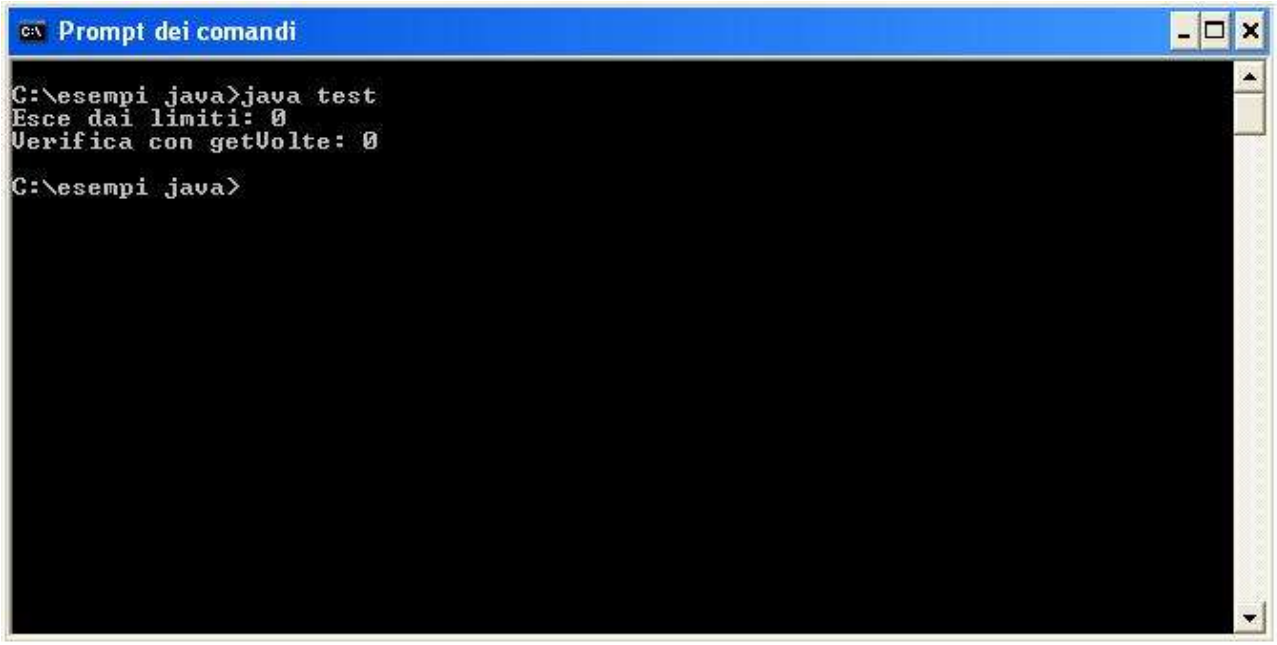
```
class Ciclo
{

         private int nvolte;

      public Ciclo()
      {

      }

      public void setVolte(int volte)
      {
         if(volte > 10 || volte <= 0)
            {
            System.out.println("Esce dai limiti: " + volte);
            }
```

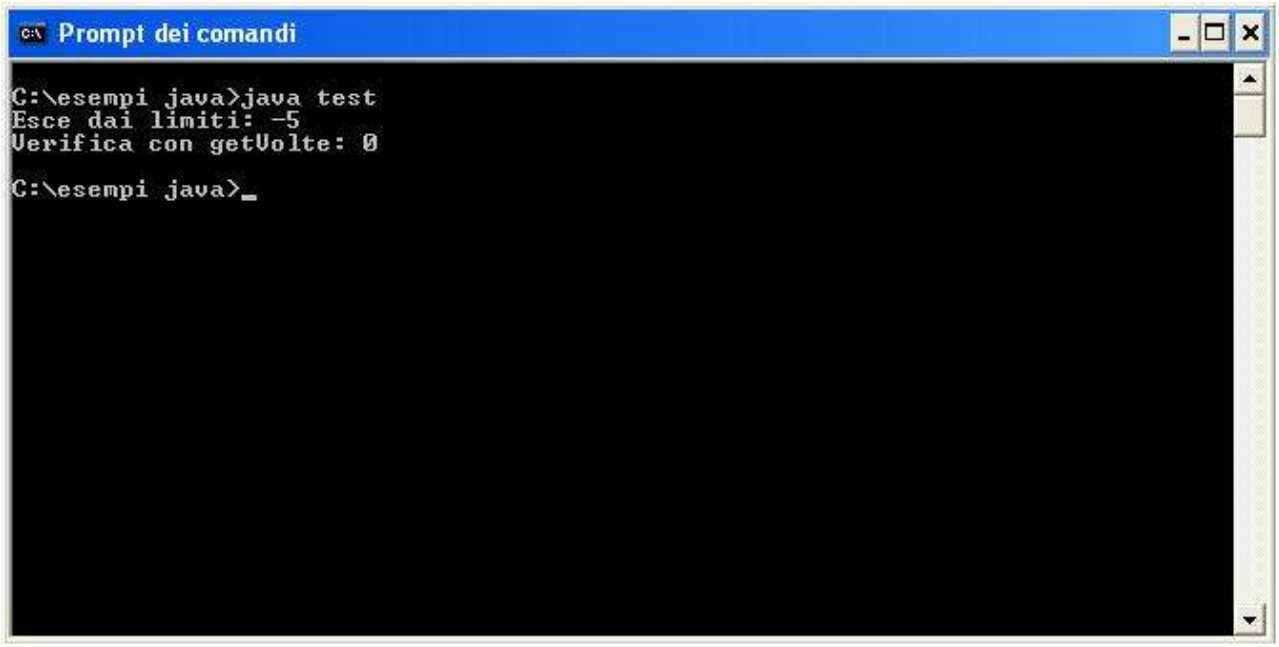
```
         else
            {
            nvolte = volte;
            System.out.println("Numero di volte impostate: " + nvolte);
            }
      }

      public int getVolte()
      {
         return nvolte;
      }

}
```

Modifichiamo adesso il precedente file test.java inserendo il codice seguente:

```
class test
{
      public static void main(String[] args)
      {
         Ciclo ciclo = new Ciclo();
         ciclo.setVolte(15);
         System.out.println("Verifica con getVolte: " + ciclo.getVolte());
      }
}
```

Vediamo ora come si comporta il programma. Supponiamo che la classe Ciclo rappresenti per l'appunto un ciclo che deve essere eseguito un numero di volte compreso fra 1 e 10. Se, per errore, venisse impostato un numero di volte minore o uguale a 0 o maggiore di 10, si potrebbe causare un errore. Quindi, per evitare di generare tale errore, inseriamo il numero di volte utilizzando il metodo setVolte(). Come possiamo notare dal codice, questo metodo controlla il valore che gli viene passato. Se il valore inserito rispetta le condizioni da noi imposte, viene accettato altrimenti viene rifiutato. Se avessimo inserito tale valore scrivendo, ad esempio, `ciclo.nvolte = <valore>` non avremmo avuto alcun tipo di controllo, mentre utilizzando il metodo setVolte() viene effettuato un rigoroso controllo dei valori inseriti. Inoltre, con getVolte() controlliamo lo stato di nvolte. Una precisazione: ovviamente, anche se avessimo voluto modificare o leggere i valori dell'attributo nvolte non avremmo potuto ed il compilatore ci avrebbe segnalato errore. Questo perché nvolte ha un livello di accesso privato. Per verificare, compiliamo e testiamo il file test.java. Proviamo inserendo come parametro di setVolte() quattro numeri: 0, -5, 19 e 7. [NOTA: ovviamente, ogni volta che inseriamo un nuovo valore nel file test.java dovremo ricompilare per rendere effettive le modifiche apportate.] I risultati prodotti sono i seguenti:


Prompt dei comandi
C:\esempi java>java test
Esce dai limiti: 0
Verifica con getVolte: 0
C:\esempi java>


Prompt dei comandi
C:\esempi java>java test
Esce dai limiti: -5
Verifica con getVolte: 0
C:\esempi java>

Come vediamo dagli schemi, tramite il metodo getVolte() verifichiamo ogni volta lo stato di nvolte. Notiamo che nei primi tre casi, avendo inserito valori non ammessi, il programma ci segnala il numero errato che abbiamo inserito. Il valore di nvolte rimane sempre 0; questo perché il programma non ne modifica il valore, a causa del valore da noi inserito non ammesso. Nell'ultimo caso, invece, il programma ci segnala che il numero inserito è ammesso e, con getVolte(), notiamo che il valore di nvolte viene impostato secondo il valore da noi stabilito. Questa è una strategia per evitare che vengano passati valori non ammessi agli attributi. Vediamo adesso come utilizzare l'oggetto corrente this.

L'oggetto corrente *this*: è possibile definire un costruttore della classe Persona usando come nome del parametro lo stesso nome dell'attributo. Il codice seguente ha lo scopo di chiarire questo concetto:

```
class Persona
{

         public String nome;
         public String cognome;
         public int eta;
         private double statura;


      public Persona()
      {

      }

      public Persona(String nome)
      {
         this.nome = nome;
         cognome = "";
         eta = 0;
      }

      public Persona(String nome, String cognome)
      {
         this.nome = nome;
         this.cognome = cognome;
         eta = 0;
      }

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

}
```

Per evitare di inventare nomi diversi di variabile, usiamo this. Con questa parola chiave possiamo utilizzare lo stesso nome, rendendo più semplice ed immediata la lettura del codice. this è un riferimento all'oggetto corrente; in questo modo l'oggetto è in grado di "capire" che ci stiamo riferendo a lui stesso. Può essere utile utilizzare this in classi con metodi costruttori sovraccarichi. Il codice seguente fornisce un esempio:

```
class Persona
{

         public String nome;
         public String cognome;
         public int eta;
         private double statura;


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

      public Persona(String nome)
      {
         this(nome, "", 0);
      }
```

```
    public Persona(String nome, String cognome)
    {
       this(nome, cognome, 0);
    }

}
```

Vediamo che in questo caso this richiama il costruttore Persona che prende tre argomenti. Inoltre, un costruttore può fornire un altro oggetto. Ad esempio:

```
public Persona(Persona persona)
{
   this(persona.nome, persona.cognome, persona.eta);
}
```

Vedremo adesso brevemente alcuni concetti chiave, quali incapsulamento, information hiding etc.

6. Interfacce con l'esterno

Come abbiamo potuto constatare con l'esempio del ciclo, grazie ai metodi possiamo leggere e modificare anche gli attributi con livello di accesso private. Quindi, per poter accedere anche agli attributi nascosti, occorre usare obbligatoriamente dei metodi. Tali, metodi, però devono essere dichiarati public, altrimenti sarebbero anch'essi nascosti. Possiamo affermare che questi metodi appartengono all'interfaccia con l'esterno di un oggetto.

7. Incapsulamento

Per quanto riguarda il concetto di incapsulamento, ci limitiamo alla seguente definizione. *Un oggetto è costituito da un insieme di metodi e di attributi che sono incapsulati nell'oggetto stesso.* E' importante che un oggetto esponga solamente la porzione di codice e di dati che il programma deve utilizzare. Qualsiasi altro dato e codice deve rimanere nascosto, affinché l'oggetto possa mantenere uno stato "consistente".

8. Information Hiding

Abbiamo visto che con la parola chiave private possiamo nascondere informazioni all'esterno. Si realizza in tal modo l'*information hiding*, cioè non si consente di "vedere" come vengono implementati i metodi, ma se ne consente solamente l'utilizzo.

9. Variabili e metodi statici

Può capitare di voler definire un membro di una classe da usare in modo indipendente da qualunque altro oggetto di tale classe. Normalmente, si accede ad un membro di una classe tramite un oggetto della sua classe e per mezzo della *dot-notation*. Si può anche creare un membro da usare da solo, cioè senza riferimento ad una specifica istanza. Proviamo a modificare la classe Persona:

```
class Persona
{

         public String nome;
         public String cognome;
         public int eta;
         private double statura;
         public static int ngambe = 2;


      public Persona()
      {

      }

      public Persona(String nome)
      {
         this.nome = nome;
         cognome = "";
         eta = 0;
      }

      public Persona(String nome, String cognome)
      {
         this.nome = nome;
         this.cognome = cognome;
         eta = 0;
      }

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

}
```

Fra gli attributi di Persona ne abbiamo aggiunto uno, ngambe, che rappresenta il numero di gambe. Tale attributo è preceduto dalla parola chiave static. La forma generale per definire una variabile statica è la seguente:

```
static <tipo> <nome>;
```

Proviamo a richiamare dalla classe Persona l'attributo statico ngambe. Modifichiamo il file test.java con il seguente codice:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();

         System.out.println(p.ngambe);
         System.out.println(Persona.ngambe);
      }
}
```

Il risultato che il compilatore ci restituisce è il seguente:

```
2
2
```

Dal codice del file test.java notiamo che ngambe può essere richiamato sia creando un'istanza di Persona che non creandola.
Se, invece, volessimo impedire di richiamare l'attributo ngambe dalle istanze, dovremmo impostare il livello di visibilità del metodo costruttore su private. Per provare, modifichiamo la classe Persona inserendo il codice seguente:

```
class Persona
{

         public static int ngambe = 2;

      private Persona()
      {

      }
}
```

Modifichiamo anche il file test.java nel seguente modo:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();
      }
}
```

Compiliamo quindi il file test.java. In fase di compilazione, il compilatore ci segnalerà il seguente errore:

```
test.java:5: Persona() has private access in Persona
           Persona p = new Persona();
                       ^
1 error
```

In questo modo non possiamo richiamare ngambe dall'oggetto p. Quindi tale attributo potrà essere richiamato unicamente direttamente dalla classe Persona. Modifichiamo opportunamente il file test.java inserendo il seguente blocco di codice:

```
class test
{
      public static void main(String[] args)
      {
         System.out.println(Persona.ngambe);
      }
}
```

Se proviamo a compilare, ci accorgeremo che il compilatore non ci segnalerà alcun errore. Lo stesso discorso vale per i metodi statici. La forma generale per definire un metodo statico è la seguente:

```
static <tipo_val_ritorno> <nome> ([<parametri>])
{
      <istruzioni>
}
```

Quindi, supponendo di avere il metodo contaGambe() nella classe Persona:

```
class Persona
{

         public static int ngambe = 2;

      public Persona()
      {

      }

      public static int contaGambe()
      {
         return ngambe;
      }
}
```

Proviamo a richiamare il metodo contaGambe(). Modifichiamo la classe test.java inserendo il seguente codice:

```
class test
```

```
{
      public static void main(String[] args)
      {
         System.out.println("Valore di ngambe: " + Persona.contaGambe());
      }
}
```

10. Ereditarietà

In Java è possibile creare gerarchie di classi e di sottoclassi. Con l'ereditarietà si può creare una classe generale che definisce tratti comuni ad un gruppo di elementi collegati. Questa classe può quindi essere ereditata da altre classi più specifiche, ognuna delle quali aggiunge ad essa elementi univoci. In Java, una classe ereditata viene definita *superclasse*, mentre la classe che compie l'ereditarietà viene definita sottoclasse. Per creare una sottoclasse si utilizza la parola chiave **extends** subito dopo il nome della classe, seguita dal nome della superclasse. La forma generale è la seguente:

```
class <nome> extends <nome_superclasse>
{

}
```

Tornando all'esempio della classe Persona, la potremmo estendere, ad esempio, alla classe EssereUmano. Proviamo quindi a creare la classe EssereUmano e a modificare la classe Persona per dimostrare l'ereditarietà:

```
class EssereUmano
{

         public int ngambe = 2;
         public int nbraccia = 2;

      public EssereUmano()
      {

      }


      public void cammina(int numkm)
      {
         System.out.println("Camminato per " + numkm + " chilometri.");
         System.out.println("Un po' di esercizio fa bene!");
      }

      public void dormi()
      {
         System.out.println("Che sonno... sto dormendo");
      }
}
```

Vediamo che nella nuova classe EssereUmano abbiamo definito gli attributi ngambe e nbraccia ed i metodi cammina() e dormi(). Modifichiamo ora anche la classe Persona:

```
class Persona extends EssereUmano
{

      public Persona()
      {

      }
}
```

Vediamo che la classe Persona contiene solamente il metodo costruttore e la parola chiave extends che estende la sottoclasse Persona alla superclasse EssereUmano. Modifichiamo anche il file test.java inserendo il seguente codice per verificare se le classi precedentemente dichiarate sono corrette:

```
class test
{
      public static void main(String[] args)
      {
         EssereUmano eu = new EssereUmano();
         Persona p = new Persona();

         System.out.println("*** ATTRIBUTI DI ESSEREUMANO ***");
         System.out.println("Attributo ngambe: " + eu.ngambe);
         System.out.println("*** METODI DI ESSEREUMANO ***");
         System.out.print("Metodo cammina(): ");
         eu.cammina(125);
         System.out.println("Metodo dormi(): ");
         eu.dormi();
          System.out.println("\n\nApplicazione dei metodi e degli attributi di
EssereUmano");
         System.out.println("a Persona per ereditarietà:");
           System.out.println("***  RICHIAMO  DI  ATTRIBUTI  E  METODI  DA  PERSONA
***");
         System.out.println("ngambe da Persona: " + p.ngambe);
         System.out.println("Metodo cammina() da Persona: ");
         p.cammina(200);
         System.out.println("Metodo dormi() da Persona: ");
         p.dormi();
      }

}
```

Il risultato prodotto è il seguente:

Vediamo che, nel file test.java, creiamo l'oggetto eu di classe EssereUmano e l'oggetto p di classe Persona. Dall'oggetto eu richiamiamo attributi e metodi. Se guardiamo all'interno del file sorgente della classe Persona, notiamo solamente il metodo costruttore e nient'altro. Eppure, nella classe di prova, richiamiamo attributi e metodi (es. `p.cammina(200)` e `p.dormi()`). Questi metodi non sono definiti all'interno di Persona ma di EssereUmano. Per ereditarietà Persona eredità questi metodi.

Quando una classe eredita dalle classi superiori nella gerarchia attributi e metodi, vuol dire che, se la superclasse definisce già un comportamento richiesto dalla classe, non occorre ridefinirlo e copiarne il codice. Infatti, la classe eredita automaticamente il comportamento dalla superclasse. Quando una sottoclasse possiede un metodo con stesso nome, stessi parametri di ingresso, stesso valore di ritorno della superclasse, ma con implementazione (codice) diverso, il metodo della superclasse non viene più ereditato. In questo caso, si dice che la classe, con operazione di overriding, ridefinisce il metodo della superclasse. Quindi, ridefinendo un metodo nascondiamo la definizione del metodo della superclasse. Per richiamare il metodo originale e non quello ridefinito all'interno della classe, si utilizza la parola chiave **super**, che consente di risalire nella gerarchia. In definitiva, una classe può ereditare metodi e attributi da una superclasse, estendere nuovi metodi ed attributi e ridefinire attributi e metodi (overriding). Vediamo subito un esempio, sempre utilizzando le classi Persona ed EssereUmano.

```
class EssereUmano
{

         public int ngambe = 2;
         public int nbraccia = 2;

      public EssereUmano()
      {

      }


      public void cammina(int numkm)
```

```
      {
         System.out.println("Camminato per " + numkm + " chilometri.");
         System.out.println("Un po' di esercizio fa bene!");
      }

      public void dormi()
      {
         System.out.println("Che sonno... sto dormendo");
      }
}                 // nessuna modifica a EssereUmano
```

La classe EssereUmano non è stata modificata. Modifichiamo, invece, la classe Persona:

```
class Persona extends EssereUmano
{

      public Persona()
      {

      }

      public void cammina(int numkm)
      {
         if(numkm > 100)
            System.out.println("Troppa strada!");
         else
            System.out.println("Ho camminato per " + numkm + " chilometri.");
      }

      public void mangia(String cibo, int numhg)
      {
         System.out.println("Mangiato " + numhg + " etti di " + cibo);
      }

}
```

Modifichiamo ora il file test.java inserendo il seguente codice:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();

         p.mangia("Pasta", 2);
         p.cammina(150);
         p.cammina(70);
         p.dormi();
      }
}
```

Analizziamo il codice. La classe EssereUmano non ha subito alcuna modifica. La classe Persona eredita da EssereUmano il metodo dormi(), definisce un nuovo metodo, il metodo mangia() e ridefinisce il metodo cammina(). Soffermiamoci su quest'ultimo. Il metodo cammina() non viene ereditato da EssereUmano, in quanto viene ridefinito in Persona. Se fosse stato ereditato da EssereUmano, il risultato prodotto sarebbe stato simile al seguente: "Camminato per 150 chilometri. Un po' di esercizio fa bene!". Invece, notiamo che il risultato prodotto è ben diverso. Se avessimo

voluto chiamare il metodo originale, avremmo usato la parola chiave super. Proviamo ad utilizzarla. Modifichiamo il codice sorgente di Persona:

```
class Persona extends EssereUmano
{

      public Persona()
      {

      }

      public void cammina(int numkm)
      {
         if(numkm > 100)
            System.out.println("Troppa strada!");
         else
            System.out.println("Ho camminato per " + numkm + " chilometri.");
      }

      public void superCammina(int numk)
      {
         super.cammina(numk);
      }

      public void mangia(String cibo, int numhg)
      {
         System.out.println("Mangiato " + numhg + " etti di " + cibo);
      }
}
```

Modifichiamo ora anche il file test.java:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();

         System.out.println("Usando il metodo cammina() di Persona:");
         p.cammina(150);
         p.cammina(75);
         System.out.println("Usando il metodo cammina() di EssereUmano:");
         p.superCammina(150);
      }
}
```

All'interno del metodo superCammina() di Persona vi è l'istruzione `super.cammina(numk);`. Come già detto, questa istruzione permette di richiamare il metodo contenuto nella superclasse. La parola chiave super può anche essere utilizzata per richiamare i metodi costruttori della superclasse. Vediamo un esempio concreto di come una sottoclasse richiama un costruttore della superclasse. Modifichiamo la classe EssereUmano con il codice riportato di seguito:

```
class EssereUmano
{

         public int ngambe = 2;
         public int nbraccia = 2;
         public String razza;
```

```
      public EssereUmano()
      {

      }

      public EssereUmano(String razza)
      {
         this.razza = razza;
      }


      public void cammina(int numkm)
      {
         System.out.println("Camminato per " + numkm + " chilometri.");
         System.out.println("Un po' di esercizio fa bene!");
      }

      public void dormi()
      {
         System.out.println("Che sonno... sto dormendo");
      }
}
```

Modifichiamo anche la classe Persona:

```
class Persona extends EssereUmano
{

         public String razza;


      public Persona(String razza)
      {
         super(razza);
         this.razza = razza;
      }

      public String getRazza()
      {
         return new String("Razza impostata: " + razza);
      }
}
```

Quindi, per fare una prova, modifichiamo anche il file test.java:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona("Europeo");
         System.out.println(p.getRazza());
      }
}
```

Infine, ricordiamo che super può essere utilizzato anche per richiamare gli attributi della superclasse. Per esempio, all'interno della classe Persona, scrivendo `super.ngambe` si richiama l'attributo ngambe. Proviamo inserendo il metodo getNumeroGambe() all'interno di Persona:

```
public int getNumeroGambe()
{
   return super.ngambe;
}
```

Vediamo ora la composizione.

11. Composizione

Una classe può avere come membri dei riferimenti ad oggetti di altre classi. Tale capacità viene chiamata composizione. Per esempio, se aggiungessimo alla classe Persona un metodo vaiDalDentista(), potremmo includere un riferimento ad un oggetto Dentista come membro dell'oggetto Persona. Il codice seguente mostra in maniera pratica tale esempio. Di seguito è riportato il codice del metodo vaiDalDentista() ed il codice della classe Dentista.

```
class Persona extends EssereUmano
{

      public Persona()
      {

      }

      public void vaiDalDentista(Dentista dentista)
      {
         System.out.println("Vado dal dentista dr. " + dentista.getNome());
      }

}
```

Vediamo ora il codice della classe Dentista:

```
class Dentista
{

         public String nome;

      public Dentista(String nome)
      {
         this.nome = nome;
      }

      public String getNome()
      {
         return new String(nome);
      }
}
```

Per fare una prova, modifichiamo il file test.java inserendo il codice seguente:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();
         p.vaiDalDentista(new Dentista("Giovanni Rossi"));
      }

}
```

oppure, dato che il risultato che si ottiene è lo stesso, il seguente codice:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();
         Dentista d = new Dentista("Giovanni Rossi");
         p.vaiDalDentista(d);
      }
}
```

Esamineremo adesso metodi e classi astratte.

12. Metodi e classi astratte

Una volta organizzate le classi in una gerarchia di ereditarietà, si presume che le classi superiori della gerarchia siano quelle più generali ed astratte, mentre quelle poste inferiormente nella gerarchia siano quelle più specifiche e concrete. Quando si progetta un insieme di classi, si raggruppano attributi e metodi comuni in una superclasse condivisa. Se il motivo principale dell'esistenza di questa superclasse è quello di operare come centro di riferimento comune e condiviso e si pensa di ridefinire attributi e metodi nelle sue sottoclassi, tale classe viene detta astratta e viene dichiarata con abstract. La forma generale per dichiarare una classe astratta è la seguente:

```
abstract class <nome>
{
      <attributi>
      <metodi>
}
```

Le classi astratte non possono essere istanziate. Possono però contenere tutto quello che una classe normale può contenere. Inoltre può contenere dei metodi astratti. Anche questi si comportano in maniera simile, in quanto possiedono una segnatura ma non l'implementazione, che si suppone sia fornita nelle sottoclassi. La forma generale per dichiarare un metodo astratto è la seguente:

```
abstract <tipo_val_ritorno> <nome> ([<parametri>])
{
      <istruzioni>
}
```

Vediamo un esempio di classi e metodi astratti, sempre riferendoci alle classi EssereUmano e Persona.

```
abstract class EssereUmano
{
         public int ngambe = 2;
         public int nbraccia = 2;

         public abstract void vaiDalDentista(Dentista dentista);
                                        // metodo astratto, solo
                                        // la segnatura. L'implementazione
                                        // sarà presente nelle sottoclassi

         public void cammina(int numkm)
         {
            System.out.println("Camminato per " + numkm + " chilometri.");
            System.out.println("Un po' di esercizio fa bene!");
         }

         public void dormi()
         {
            System.out.println("Che sonno... sto dormendo");
         }

                  // i metodi cammina() e dormi() sono concreti

}
```

Vediamo che all'interno della classe astratta EssereUmano è dichiarato il metodo astratto vaiDalDentista(). Il metodo possiede solamente la segnatura. Vedremo adesso che l'implementazione (il codice) del metodo verrà dichiarato all'interno della classe Persona.

```
class Persona extends EssereUmano
{

      public Persona()
      {

      }

      public void vaiDalDentista(Dentista dentista)
      {
         System.out.println("Vado dal dentista dr. " + dentista.getNome());
      }

}
```

Modifichiamo ancora il file test.java per testare l'esempio appena visto:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();

         System.out.println("Numero di gambe: " + p.ngambe);
         System.out.println("Numero di braccia: " + p.nbraccia);

         p.vaiDalDentista(new Dentista("Giovanni Rossi"));
         p.cammina(100);
         p.dormi();
      }
}
```

[NOTA: ovviamente il programma non funzionerà se la classe Dentista dovesse mancare.]

13. Overloading di metodi

All'interno di una classe è possibile dichiarare metodi che hanno lo stesso nome ma differente numero di parametri. Quando viene richiamato il metodo, la scelta corretta del metodo da eseguire viene effettuata contando il numero e verificando il tipo di parametri. Questa situazione è nota come *overloading* dei metodi. Un esempio di tale situazione lo abbiamo già visto parlando di metodi costruttori.

14. Polimorfismo

Nella prima parte abbiamo espresso il seguente concetto: "il polimorfismo è la qualità che permette ad un'interfaccia di accedere ad una classe generale di azioni. In altre parole un'interfaccia è polimorfica nel senso che cambia forma a seconda della classe in cui si trova". Un metodo si definisce polimorfico nel senso che cambia forma a seconda della classe in cui si trova. Per esempio, immaginiamo una classe EsseriCheCamminano: in questa classe potrebbe essere contenuto un metodo come cammina(). Nelle possibili sottoclassi della superclasse EsseriCheCamminano, ad esempio le classi Cane e Persona, il metodo cammina() viene ridefinito, in quanto Persona non cammina nello stesso modo di Cane. Diciamo quindi che il metodo si è adeguato alle particolari esigenze delle due sottoclassi.

15. Attributi, metodi e classi finali

Introduciamo adesso attributi e metodi costanti. Un attributo che deve essere costante viene preceduto dalla parola chiave **final**. Questo attributo può assumere un unico valore durante l'inizializzazione, e manterrà questo valore inalterato in tutte le istanze. In alcuni casi gli attributi final vengono dichiarati con livello di accesso privato: in questo modo non si corre il rischio che possano essere modificati. Per esempio, se dovessimo creare una classe che contiene il valore di una costante, ad esempio il valore di π, scriveremo:

```
public static final double PIGRECO = 3.14;
```

Un metodo final indica invece un metodo che non può essere ridefinito nelle sottoclassi.
Una classe final indica che la classe non ammette sottoclassi. Quindi nessun metodo potrà essere ridefinito. Possiamo affermare quindi che tutti i metodi di una classe final sono final.

16. Interfacce

Vediamo adesso le interfacce. Può capitare spesso in Java di dover utilizzare l'ereditarietà. Va saputo, però, che in Java l'ereditarietà è singola e, quindi, una sottoclasse può essere estesa solamente in una superclasse. Può altresì capitare di dover estendere una sottoclasse in più superclassi. Questo, come già detto, non è possibile. Per risolvere il problema, si utilizzano le interfacce. Vediamo come dichiarare un'interfaccia. La forma generale per dichiarare un'interfaccia è la seguente:

```
public interface <nome>
{

}
```

Una possibile dichiarazione di interfaccia è la seguente:

```
public interface MiaInterfaccia
{
      public void faiQualcosa();
      public static final int unNumero = 120;
}
```

Come notiamo dal codice, l'interfaccia ha solo segnature di metodi. Inoltre, gli attributi sono static final. I metodi possono essere dichiarati unicamente public o abstract.
Per implementare un'interfaccia all'interno di una classe occorre far seguire al nome della classe la parola chiave **implements** seguita dal nome dell'interfaccia. La forma generale è la seguente:

```
class <nome> implements <nome_interfaccia> [, <nome_interfaccia>]
{
      <attributi>
      <metodi>
}
```

Vediamo un esempio. Modifichiamo la classe Persona:

```
class Persona implements MiaInterfaccia
{

      public Persona()
      {

      }

      public void faiQualcosa()
      {
         System.out.println("Faccio qualcosa... che noia!");
      }

}
```

Adesso, per provare, modifichiamo il file test.java:

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();

         p.faiQualcosa();

         System.out.println("Un numero (da Persona): " + p.unNumero);
         System.out.println("Un numero (da MiaInterfaccia): " +
                                                    MiaInterfaccia.unNumero);
      }
}
```

Vediamo come usare un'interfaccia.
Non è possibile costruire oggetti di un tipo interfaccia. Ad esempio

```
MiaInterfaccia miainterfaccia = new MiaInterfaccia();
```

genererà un errore. Invece, scrivendo

```
MiaInterfaccia p = new Persona();
```

non genererà alcun errore. Infatti l'interfaccia MiaInterfaccia è implementata in Persona. Affermiamo quindi che è possibile dichiarare oggetti di un tipo di interfaccia. Possiamo inoltre chiamare i metodi, ad esempio:

```
p.faiQualcosa();
```

Infine, affermiamo che un'interfaccia è un modello del comportamento di un oggetto. Abbiamo visto che utilizzando le interfacce possiamo condividere quindi un comportamento tra varie classi di una determinata gerarchia. Vedremo ora come creare package.

17. Creazione di package

Per creare un pacchetto occorre inserire un comando **package** all'inizio del file sorgente Java. Le classi dichiarate all'interno del file apparterranno così al pacchetto specificato. Inoltre, è possibile creare una gerarchia di pacchetti. Basta separare ogni nome di pacchetto con un punto. La forma generale di una dichiarazione di pacchetti a più livelli è la seguente:

```
package <nome>.<nome>...
```

Vediamo subito un esempio:

```
package esempi;

class PackageTest
{
      public static void main(String[] args)
      {

      }
}
```

Per fare un test, creiamo un file PackageTest.java e salviamo al suo interno il codice riportato sopra. Adesso compiliamo il file utilizzando il seguente comando:

```
javac –d . PackageTest.java
```

Notiamo che nella cartella dove noi stiamo lavorando (ad esempio desktop), è comparsa un'altra cartella, la cartella esempi. Apriamola. Al suo interno troviamo la classe PackageTest. Per importare un pacchetto utilizziamo import. Vediamo un esempio pratico:

```
package resources.system;

import java.io.*;


public class SystemInformation
{

      private SystemInformation()
      {

      }

      public static String getOSName()
      {
         return new String(System.getProperty("os.name"));
      }

}
```

Viene creata la cartella resources. All'interno di questa cartella vi è un'altra cartella, system. Al suo interno è contenuta la classe SystemInformation. Per adesso tralasciamo le istruzioni contenute all'interno della classe, che saranno prese in esame più avanti nel corso. Ci basta per ora sapere che l'istruzione contenuta nel metodo di classe getOSName() restituisce una stringa contenente il nome del sistema operativo in uso. Vediamo ora come importare la classe:

```
import resources.system.SystemInformation;

class PackageTest
{
      public static void main(String[] args)
      {
         System.out.println(SystemInformation.getOSName());
      }
}
```

Compiliamo ed eseguiamo. Il risultato prodotto sarà il nome del sistema operativo che si sta utilizzando.

18. L'operatore instanceof

Può capitare di dover verificare se un'istanza appartiene ad una determinata classe. Per far ciò utilizziamo l'operatore instanceof. Nel esempio che segue, viene creato un oggetto p di classe Persona ed un oggetto a di classe Automobile. [NOTA: la classe Automobile non è mai stata utilizzata fino ad ora. Creeremo questa classe solamente per mostrare l'utilizzo dell'operatore instanceof. Tale classe contiene quindi solamente il metodo costruttore.]

File Automobile.java:

```
class Automobile
{

      public Automobile()
      {

      }

}
```

File Persona.java

```
class Persona
{

      public Persona()
      {

      }

}
```

File test.java

```
class test
{
      public static void main(String[] args)
      {
         Persona p = new Persona();
         Automobile a = new Automobile();

            if(p instanceof Persona)
               System.out.println("p è istanza di Persona");
            else
               System.out.println("p non è istanza di Persona");
            if(a instanceof Automobile)
               System.out.println("a è istanza di Automobile");
            else
               System.out.println("a non è istanza di Automobile");
      }
}
```

Compiliamo ed eseguiamo. Il risultato prodotto sarà il seguente:

```
p è istanza di Persona
a è istanza di Automobile
```

19. Metodi conclusivi

Abbiamo visto che un metodo costruttore alloca ed inizializza un oggetto. Un metodo conclusivo, invece, è un metodo richiamato a svolgere le azioni finali prima di deallocarne la memoria. Java possiede un metodo conclusivo chiamato finalize(), che può essere direttamente richiamato dal programmatore per eseguire le azioni necessarie ad evitare una distruzione brusca dell'oggetto. Il metodo finalize() viene richiamato nel seguente modo:

```
protected void finalize()
{
   <istruzioni>
}
```

20. Garbage collection

In Java, quando i riferimenti ad un oggetto vengono eliminati, o non esistono riferimenti ad un oggetto, tale oggetto non è più necessario e la memoria da esso occupata viene liberata. Questa memoria riciclata può quindi essere usata per un'assegnazione successiva. A grandi linee, questo è il sistema di funzionamento della garbage collection di Java. La garbage collection avviene solamente qualche volta durante l'esecuzione del programma. Inoltre la garbage collection richiede del tempo, quindi il sistema runtime di Java la utilizza solamente quando è necessario. Il metodo finalize() appena visto serve proprio per chiudere eventuali file lasciati aperti, pulire alcuni variabili etc. prima che un oggetto venga rimosso dalle procedure di garbage collection.

Conclusioni

A questo punto possiamo affermare di conoscere il funzionamento del linguaggio Java. Nelle prossime parti apprenderemo aspetti molto più interessanti. Vedremo come gestire le stringhe nei dettagli, come operare sui vettori, utilizzare thread, applet, grafica e svariate funzionalità di Java.

Nelle parti successive si dà per scontato che si conoscano gli argomenti visti fino ad ora, per cui, prima di continuare, è bene conoscere a fondo gli argomenti delle prime quattro parti di questo corso. Vengono allegati alcuni esercizi: lo scopo di questi è di aiutare a capire se si sono compresi gli argomenti finora trattati. Alla fine di ognuna delle prossime parti sono allegati esercizi. Nella prossima lezione vedremo un aspetto avanzato della programmazione Java: la ricorsività.