# *Capitolo 3. Pesci e Animali Domestici – Le Classi di Java*

Translated by Francesco Orciuoli

I programmi scritti in Java sono formati da *Classi* che rappresentano gli oggetti del mondo reale. Sebbene le persone possano avere preferenze diverse su come scrivere i programmi, la maggior parte di loro concorda che è meglio scrivere il codice con uno stile di programmazione (e non solo) chiamato *object-oriented* (orientato agli oggetti). Questo vuol dire che un "buon" programmatore Java inizia con il decidere quali oggetti devono essere inclusi nel suo programma e di conseguenza quali Classi di Java rappresenteranno tali oggetti. Soltanto dopo, che questa scelta è stata fatta, il programmatore potrà iniziare a scrivere il suo codice.

## Classi e Oggetti

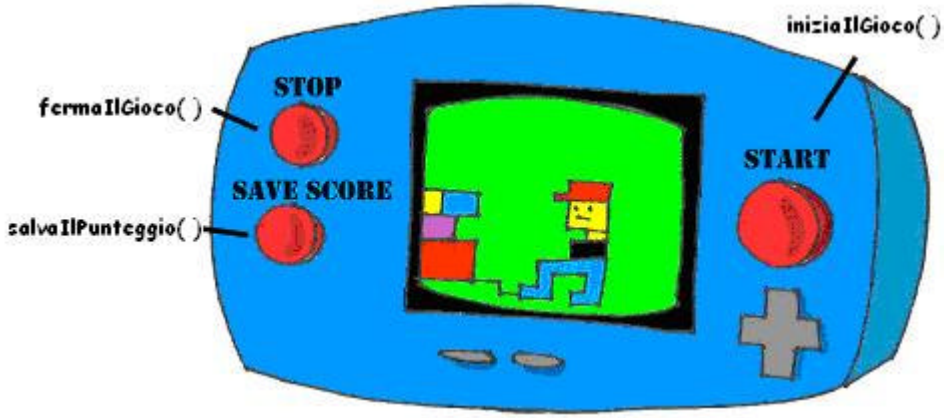
Le Classi in Java possono avere *Metodi* e *Attributi*.

I Metodi definiscono le azioni che una Classe può eseguire.

Gli Attributi descrivono una Classe.

Cerchiamo, ora, di costruire e discutere insieme una Classe chiamata `VideoGioco`. Questa Classe può avere diversi Metodi,

che ci informano su *che cosa possono fare i suoi Oggetti*: Inizia il Gioco, Ferma il Gioco, Salva il Punteggio, e così via. La Classe `VideoGioco`, inoltre, può avere degli Attributi: Prezzo, Colore dello Schermo, Numero di JoyPad ed altri ancora.




In linguaggio Java, la Classe descritta in precedenza si scrive così:

```
class VideoGioco {
  String colore;
  int prezzo;

  void iniziaIlGioco () {
  }
  void fermaIlGioco () {
  }
  void salvaIlPunteggio(String nomeGiocatore,
      int punteggio) {
  }
```

La nostra Classe `VideoGioco` dovrebbe essere simile a tutte le altre Classi che rappresentano video-giochi – tutte loro hanno degli schermi di dimensioni e colori differenti, tutte eseguono le stesse azioni, ed infine tutte hanno un prezzo.

Potremmo essere ancora più precisi e creare un'altra Classe di Java chiamata `GameBoyAdvance`. Questa classe sicuramente appartiene alla famiglia dei video-giochi, ma ha delle proprietà specifiche del modello Game Boy Advance, per esempio può supportare un tipo particolare di cartuccia (meccanismo sul quale viene memorizzato un gioco specifico).

```
class GameBoyAdvance {
  String tipoDiCartuccia;
  int ampiezzaDelloSchermo;

  void iniziaIlGioco() {

  }
  void fermaIlGioco() {

  }
}
```

In questo esempio la Classe `GameBoyAdvance` definisce due Attributi – `tipoDiCartuccia` e `ampiezzaDelloSchermo` e due Metodi – `iniziaIlGioco()` e `fermaIlGioco()`. Ma allo stato attuale, nel nostro esempio, questi Metodi non possono eseguire le azioni indicate, perché non presentano il codice Java, tra le loro parentesi graffe, che è necessario per eseguire tali azioni.

Oltre alla parola *Classe*, utilizzata negli esempi, dovremo prendere confidenza con il significato della parola *Oggetto*.

La frase "creare un'istanza di un Oggetto" significa creare una copia di questo Oggetto nella memoria del computer in accordo con la definizione della sua Classe.

Una descrizione del Game Boy Advance è in relazione col video-gioco vero e proprio allo stesso modo in cui una Classe di Java è in relazione con una sua istanza in memoria. Il processo di

costruire giochi basati su una loro descrizione è simile al processo di costruire Oggetti GameBoyAdvance in Java.


In molti casi, un programma può utilizzare una Classe di Java soltanto dopo che una sua istanza è stata creata. Anche le fabbriche di video-giochi creano migliaia di copie di giochi basate sulla stessa descrizione. Sebbene queste copie rappresentano la stessa Classe, esse hanno differenti *Valori* all'interno dei propri Attributi - alcune di loro sono verdi, mentre altre sono color argento, e così via. In altre parole, un programma può creare *più istanze* di Oggetti GameBoyAdvance.

## Tipi di Dati

Le *Variabili* di Java rappresentano gli Attributi di una Classe, gli Argomenti dei Metodi oppure possono essere utilizzate all'interno dei Metodi per mantenere in memoria alcuni dati per

un breve periodo di tempo. Le variabili devono essere dichiarate prima di scrivere altro codice, e soltanto dopo che sono state dichiarate potranno essere utilizzate.

Ricordate le equazioni come y=x+2? In Java abbiamo bisogno di dichiarare le variabili x and y come *tipi di dati* numerici come ad esempio `integer` o `double` in questo modo:

```
int x;
int y;
```

Le due linee seguenti mostrano come assegnare un *valore* a queste variabili. Se il vostro programma assegna il valore 5 alla variabile `x`, la variabile `y` sarà uguale a 7:

```
x=5;
y=x+2;
```

In Java c'è anche la possibilità di modificare il valore di una variabile in modi differenti da quelli mostrati in precedenza. Le due linee seguenti cambiano il valore della variabile `y` da 5 a 6:

```
int y=5;
y++;
```

Nonostante abbiamo utilizzato i due segni di addizione, la JVM incrementerà il valore della variabile `y` esattamente di una unità a causa del significato particolare che si dà all'operatore ++.

Dopo l'esecuzione del seguente frammento di codice otterremo che il valore della variabile `mioPunteggio` è esattamente 6:

```
int mioPunteggio=5;
mioPunteggio=mioPunteggio+1;
```

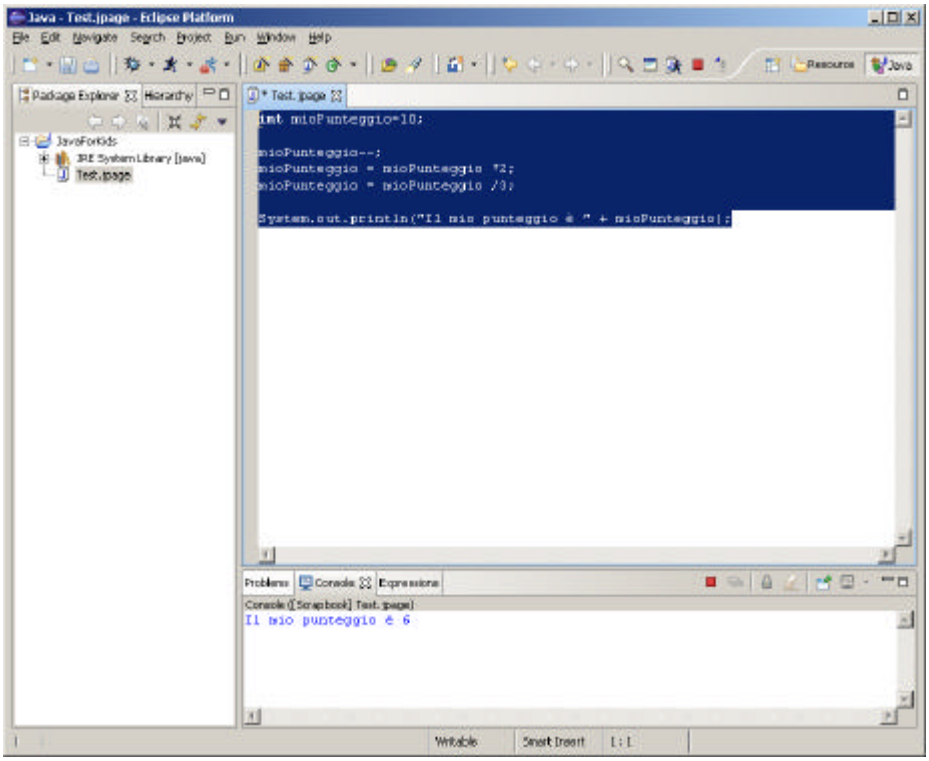
Naturalmente, in Java è anche possible usare la moltiplicazione, la divisione e la sottrazione allo stesso modo. Fate attenzione al frammento di codice seguente:

```
int mioPunteggio=10;
```

```
mioPunteggio--;
mioPunteggio = mioPunteggio *2;
mioPunteggio = mioPunteggio /3;

System.out.println("Il mio punteggio è " + mioPunteggio);
```

Che cosa stamperà sul monitor del vostro computer il frammento di codice precedente? Eclipse ha una interessante funzione chiamata **scrapbook** che permette di eseguire velocemente una piccola sequenza di istruzioni (come quella che abbiamo scritto precedentemente) senza creare una Classe che la contiene. Selezionate i menu *File*, *New*, *Scrapbook Page* e scrivete la parola `Test` quando vi verrà chiesto il nome del vostro file di scrapbook.

Ora scrivete le linee di codice (quelle che abbiamo mostrato in precedenza), che lavorano con `mioPunteggio`, nello scrapbook. Dopo averle scritte, evidenziatele e fate click sul pulsantino che ha come icona la lente di ingrandimento che trovate sulla barra degli strumenti (toolbar).



Per vedere il risultato del calcolo del punteggio fate click sul tab **console** che troverete nella parte bassa dello schermo:

```
Il mio punteggio è 6
```

In questo esempio, l'argomento del Metodo `println()` è costituito da due pezzi – la frase "Il mio punteggio è " ed il valore della variabile `mioPunteggio`, che nel nostro caso al momento della stampa è 6. La costruzione di una `String` (stringa) a partire da diversi pezzi è chiamata *concatenazione*. Sebbene `mioPunteggio` è un numero, Java è abbastanza "furbo" da convertire questa variabile in una `String`(stringa), e poi da attaccarla alla frase *Il mio punteggio è*.

Vediamo altri modi con i quali cambiare i valori delle variabili:

`mioPunteggio=mioPunteggio*2;` è uguale a `mioPunteggio*=2;`
`mioPunteggio=mioPunteggio+2;` è uguale a `mioPunteggio+=2;`
`mioPunteggio=mioPunteggio-2;` è uguale a `mioPunteggio-=2;`
`mioPunteggio=mioPunteggio/2;` è uguale a `mioPunteggio/=2;`

In Java esistono otto Tipi di Dati semplici o *primitivi*, e siete voi a dover decidere quali di questi utilizzare per i vostri programmi. Di volta in volta dobbiamo scegliere in base al tipo ed alla dimensione dei dati che pensate di memorizzare nelle vostre variabili:

- Quattro Tipi di Dati per memorizzare numeri interi – `byte`, `short`, `int`, e `long`.
- Due Tipi di Dati per i numeri con la virgola – `float` e `double`.
- Un Tipo di Dato per memorizzare un singolo carattere – `char`.

- ✍✍ Un Tipo di Dato *logico* chiamato `boolean` che permette l'uso di due valori: `true` (vero) or `false` (falso).

In Java è possible assegnare un valore iniziale ad una variabile durante la sua dichiarazione e questa operazione è chiamata *inizializzazione di una variabile*:

```
char livello = 'A';
int sedie = 12;
boolean suonoPresente = false;
double guadagnoNazionale = 23863494965745.78;
float prezzoDelGioco = 12.50f;
long numeroTotaleDelleMacchine =4637283648392l;
```

Nelle ultime due linee, `f` significa float e `l` significa long.

Se non inizializziamo le variabili, Java lo farà per voi assegnando zero ad ogni variabile numerica, `false` alle variabili di tipo `boolean`, ed il codice speciale '\u0000' alle variabili di tipo `char`.

Esiste anche la parola speciale `final`, e se questa viene utilizzata in una dichiarazione di variabile, possiamo assegnare un valore a questa variabile soltanto una volta e questo valore non potrà essere modificato successivamente. In alcuni linguaggi le variabili di tipo `final` vengono chiamate *costanti*. In Java, di solito, le variabili di tipo `final` vengono scritte utilizzando sempre le lettere maiuscole come nel seguente esempio:

```
final String CAPITALE="Roma";
```

In aggiunta ai Tipi di Dati primitivi, è possibile usare le Classi di Java per dichiarare le variabili. Ogni Tipo di Dato primitivo ha una Classe *wrapper* corrispondente, per esempio `Integer`, `Double`, `Boolean`, ecc... Queste Classi mantengono al loro interno dei Metodi che sono molto utili per convertire dei valori da un Tipo di Dato ad un altro.

Il Tipo di Dato `char`, come abbiamo già detto, viene utilizzato per memorizzare un solo carattere, Java ha una Classe chiamata `String` per lavorare con parole o frasi, per esempio:

```
String cognome="Rossi";
```

In Java, i nomi delle variabili non possono iniziare con un numero e non possono contenere degli spazi.

Un *bit* è il più piccolo dato che può essere mantenuto in memoria. Esso può assumere il valore 1 oppure il valore 0.

Un `byte` è formato da otto bit.

Un `char` in Java occupa due byte in memoria.

Un `int` e un `float` in Java occupano quattro byte di memoria.

Le variabili di tipo `long` e `double` occupano otto byte ciascuno.

I Tipi di Dati numerici che usano più byte possono memorizzare cifre più grandi rispetto ai Tipi di Dati che usano meno byte.

1 kilobyte (KB) ha 1024 byte.

1 megabyte (MB) ha 1024 kilobyte

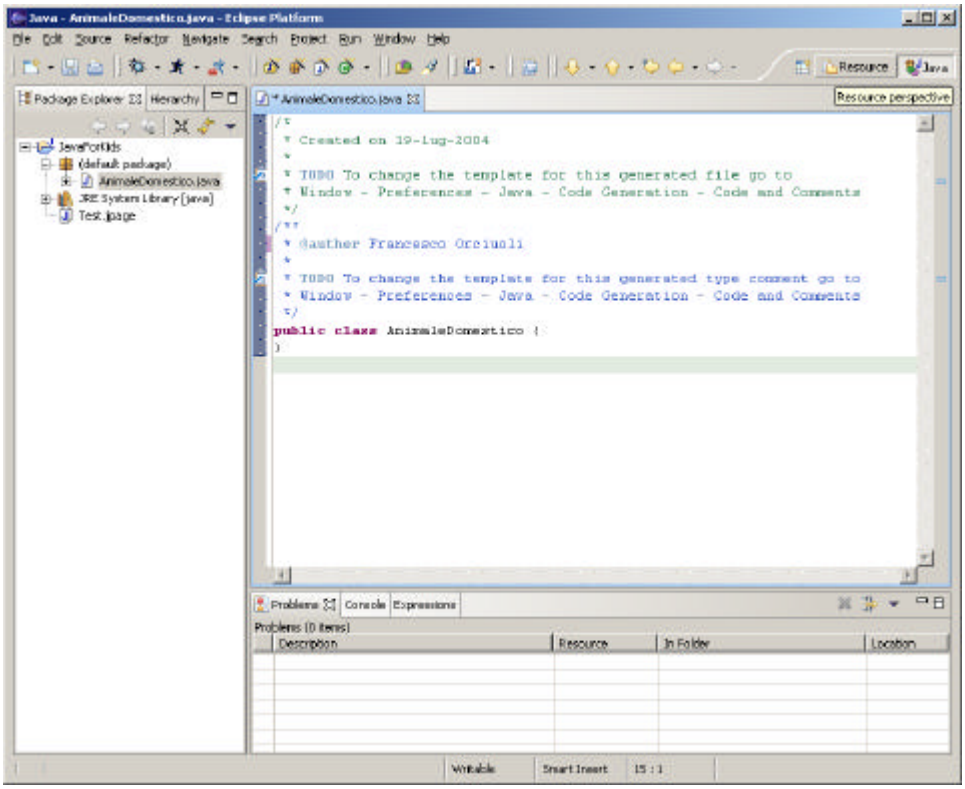
## Creazione di un Animale Domestico

Progettiamo e costruiamo la Classe `AnimaleDomestico`. Per prima cosa abbiamo la necessità di decidere quali azioni sarà capace di eseguire il nostro animale domestico. Stabiliamo che il nostro animale domestico sarà capace di mangiare, di dormire, e di dire qualcosa. Programmeremo queste azioni all'interno dei Metodi della classe `AnimaleDomestico` e assegneremo loro i seguenti Attributi: anni, altezza, peso, e colore.

Iniziamo a creare una nuova Classe di Java chiamata `AnimaleDomestico` in *Il mio primo progetto* come descritto nel

Capitolo 2, senza però selezionare il controllo per la creazione del Metodo `main()`.

Il nostro schermo dovrebbe apparire simile a questo:




Ora siamo pronti per dichiarare gli Attributi ed i Metodi nella Classe `AnimaleDomestico`. Le Classi di Java e i Metodi racchiudono il loro "corpo" all'interno di parentesi graffe. Per ogni parentesi graffa aperta ci deve essere una parentesi graffa chiusa corrispondente:

```
class AnimaleDomestico {
}
```

Per dichiarare le variabili per gli Attributi della Classe bisogna scegliere prima un Tipo di Dato da assegnare loro. E' preferibile

utilizzare il tipo `int` per gli anni, `float` per il peso e l'altezza, e `String` per il colore dell'animale domestico.

```
class AnimaleDomestico{
   int anni;
   float peso;
   float altezza;
   String colore;
}
```

Il prossimo passo consiste nell'aggiungere alcuni Metodi alla Classe creata. Prima di dichiarare un Metodo bisogna decidere se questo dovrà utilizzare degli Argomenti e se dovrà restituire un valore:

- ✍✍Il Metodo `dorme()` dovrà visualizzare sullo schermo la frase *Buona notte, ci vediamo domani* – questo Metodo non ha bisogno di alcun Argomento e non restituirà alcun valore.

- ✍✍Il Metodo `mangia()` ha le stesse caratteristiche del Metodo `dorme()` ma `dovrà` visualizzare sullo schermo il Messaggio *Sono affamato... dammi uno snack!*.

- ✍✍Il Metodo `say()` dovrà visualizzare sullo schermo un messaggio sullo schermo ovvero la frase che noi vogliamo che l'animale domestico "dica". Per fare ciò, passeremo una frase al Metodo `say()` con un *Argomento di Metodo*. Il Metodo costruirà una frase usando questo Argomento e la restituirà al programma chiamante (quello che ha ivocato il Metodo).

La nuova versione della Classe `AnimaleDomestico` diventa:

```
public class AnimaleDomestico {
 int anni;
 float peso;
 float altezza;
 String colore;

 public void dorme(){
      System.out.println(
                  "Buona notte, ci vediamo domani");
 }

 public void mangia(){
  System.out.println(
  "Ho fame…dammi uno snack!");
 }

 public String dice(String unaFrase){
      String risposta = "OK!! OK!! " + unaFrase;
            return risposta;
 }
}
```

Questa Classe rappresenta una creatura amichevole del nostro mondo reale:

Discutiamo, ora, sulla "firma" del Metodo `dorme()`:

```
public void dorme()
```

La "firma" precedente ci dice che questo Metodo può essere utilizzato da altre Classi di Java (`public`) e che non restituisce alcun valore (`void`). Le parentesi tonde vuote significano che questo Metodo non ha alcun Argomento, perchè non ha bisogno di alcuna informazione proveniente dal mondo esterno – esso visualizza sullo schermo sempre la stessa frase.

Invece, la "firma" del Metodo `dice()` è la seguente:

```
public String dice(String unaFrase)
```

Questo Metodo può essere invocato da altre Classi di Java, ma, a differenza del precedente, restituisce un testo (frase). Ci accorgiamo della restituzione del testo guardando la sua "firma" ed in particolare grazie alla parola `String` riportata prima del nome del Metodo. Inoltre, il Metodo aspetta un testo dal mondo esterno che gli viene comunicato dall'Argomento `String unaFrase`.

Come si decide se un Metodo deve o non deve restituire un valore? Se un Metodo esegue una manipolazione su alcuni dati e deve, in qualche modo, "consegnare" il risultato di questa manipolazione alla Classe che lo ha invocato allora dovrà restituire un valore. A questo punto i lettori più attenti affermeranno che la Classe `AnimaleDomestico` non ha alcuna

Classe che la invoca. Questo è corretto! Ed infatti costruiremo la Classe `Padrone`. Questa Classe deve avere al suo interno un Metodo `main()` che contiene il codice per comunicare con la Classe `AnimaleDomestico`. Creiamo un'altra Classe chiamata `Padrone`, selezionando, questa volta, l'opzione che aggiunge il Metodo `main()`. Ricordate che senza questo Metodo non si può *eseguire* (*run)* questa Classe come un programma. Modifichiamo il codice generato con Eclipse come l'esempio seguente:

```
public class Padrone {

  public static void main(String[] args) {

   String reazione;

   AnimaleDomestico mioAnimaleDomestico = new
AnimaleDomestico ();

   mioAnimaleDomestico.mangia();
   reazione = mioAnimaleDomestico.dice("Tweet!!
Tweet!!");
   System.out.println(reazione);

   mioAnimaleDomestico.dorme();

  }
}
```

Non dimenticate di premere *Ctrl-S* per salvare e compilare questa Classe!
Per eseguire la Classe `Padrone` si deve fare click sui menu di Eclipse *Run*, *Run…* , *New* e scrivere il nome della Classe principale: `Padrone`. Fate click sul pulsante *Run* e il programma visualizzerà il seguente risultato:

```
Sono affamato…dammi uno snack!
OK!! OK!! Tweet!! Tweet!!
Buona notte, ci vediamo domani
```

`Padrone` è la *Classe chiamante*, e inizia con il creare una *istanza* di Oggetto della Classe `AnimaleDomestico` dichiarando una variabile `mioAnimaleDomestico` ed utilizzando l'operatore di Java chiamato `new`:

```
AnimaleDomestico mioAnimaleDomestico = new AnimaleDomestico();
```

La linea precedente dichiara una variabile di tipo `AnimaleDomestico` (in generale potete trattare ogni Classe creata da voi come un nuovo Tipo di Dato in Java). Ora, la variabile `mioAnimaleDomestico` sa in quale posto della memoria del computer è stata creata l'istanza di `AnimaleDomestico`, ed è possibile usare questa variabile per invocare qualsiasi Metodo della Classe `AnimaleDomestico`, per esempio:

```
mioAnimaleDomestico.mangia();
```

Se un Metodo restituisce un valore, abbiamo bisogno di invocarlo in maniera differente dall'esempio precedente. Per prima cosa si deve dichiarare una variabile che ha lo stesso Tipo di Dato del valore di ritorno del Metodo e poi si deve assegnare questo valore alla variabile. Ora possiamo invocare il metodo `dice()`:

```
String reazione;

reazione = mioAnimaleDomestico.dice("Tweet!! Tweet!!");
```

A questo punto, il valore restituito viene memorizzato nella variabile `reazione` e se vogliamo vedere cosa succede allora dobbiamo utilizzare l'istruzione di stampa sullo schermo e fornire, come Argomento, il valore restituito dal Metodo `dice()`:

```
System.out.println(reazione);
```


## Ereditarietà – un Pesce è un Animale Domestico

La nostra Classe `AnimaleDomestico` ci aiuterà ad apprendere un'altra caratteristica importante del linguaggio Java che è chiamata *ereditarietà* (*inheritance*). Nella vita reale, ogni persona eredità alcune caratteristiche dai suoi genitori. Allo stesso modo, nel mondo Java è possible creare una nuova Classe basata su una Classe esistente.

La Classe `AnimaleDomestico` ha dei comportamenti e degli Attributi che sono condivisi da molti animali – essi mangiano, dormono, alcune volte producono suoni, la loro pelle ha differenti colori, e così via. Da un altro punto di vista, gli animali domestici sono differenti - i cani abbaiano, i pesci nuotano e sono muti, i pappagalli parlano meglio dei cani ✍. Ma tutti gli animali citati mangiano, dormono ed hanno un peso ed un'altezza. Questo è il motivo per cui è più semplice creare una

Classe `Pesce` che *eredita* alcuni comportamenti e alcuni Attributi comuni dalla Classe `AnimaleDomestico`, piuttosto che creare le Classi `Cane`, `Pappagallo` o `Pesce` ogni volta, da zero, senza tener conto di quello che è stato fatto in precedenza.

La parola speciale `extends` fa in modo che una Classe erediti da un'altra Classe esistente:

```
class Pesce extends AnimaleDomestico{

}
```

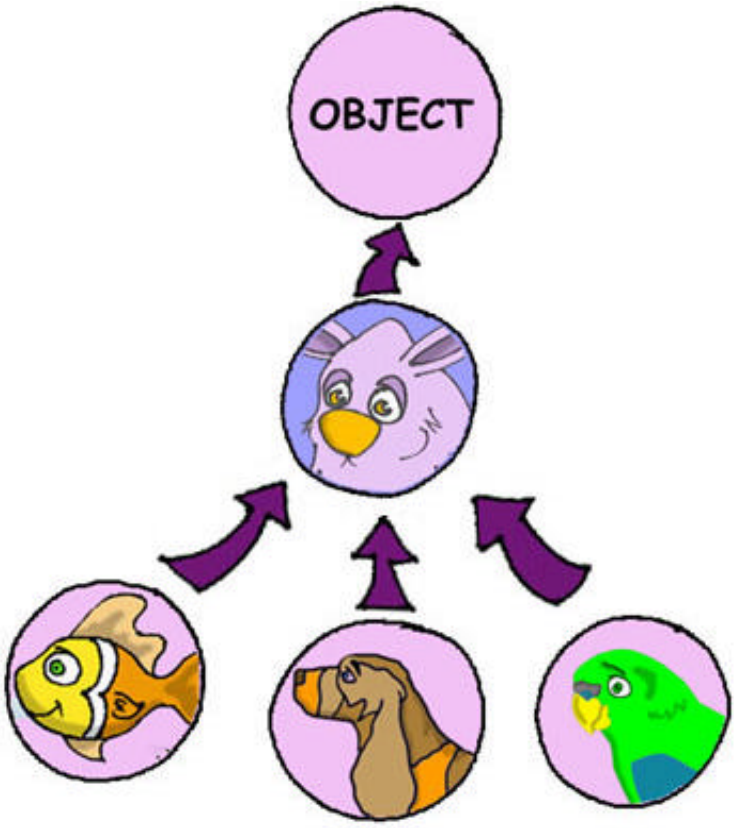
Possiamo anche dire che la Classe `Pesce` è una *sottoclasse* della Classe `AnimaleDomestico`, e la Classe `AnimaleDomestico` è una *superclasse* della Classe `Pesce`. In altre parole, potete usare la Classe `AnimaleDomestico` come modello per creare la Classe `Pesce`.

Pur lasciando la classe `Pesce` come è stata scritta in precedenza, possiamo usare tutti i Metodi e gli Attributi ereditati dalla Classe `AnimaleDomestico`. Diamo un'occhiata:

```
Pesce mioPesce = new Pesce();
mioPesce.dorme();
```

Sebbene non abbiamo dichiarato alcun Metodo nella Classe `Pesce`, possiamo comunque invocare il Metodo `dorme()` dalla sua superclasse!

La creazione delle sottoclassi in Eclipse è un'operazione semplicissima! Selezionate i menu *File*, *New*, *Class*, e scrivete `Pesce` come nome della Classe. Sostituite `java.lang.Object` nel campo che riguarda la superclasse con la parola `AnimaleDomestico`.




Non dimenticate, comunque, che stiamo creando una sottoclasse di `AnimaleDomestico` per aggiungere alcune caratteristiche, che solo i pesci hanno, e riutilizzare parte del codice scritto per gli animali domestici in generale.

E'giunto il tempo di svelare un segreto – tutte le Classi in Java ereditano dalla superclasse Object anche se non si usa la parola speciale extends.

Le Classi di Java non possono avere due genitori dai quali ereditare.
Se volgiamo continuare a paragonare le Classi alle persone, possiamo dire che tutti gli uomini discendono da Adamo, e tutte le donne discendono da Eva ✍.

Non tutti gli animali domestici possono andare sott'acqua, ma i pesci di certo lo possono fare. Aggiungiamo il nuovo Metodo `siImmerge()` alla Classe `Pesce`.

```
public class Pesce extends AnimaleDomestico {

   int profonditaCorrente=0;

   public int siImmerge(int quantoProfondo){
      profonditaCorrente=profonditaCorrente +
                                        quantoProfondo;
      System.out.println("Mi immergo per " +
                              quantoProfondo + " metri");
      System.out.println("Sona a " +
          profonditaCorrente + " metri sotto il mare");
      return profonditaCorrente;
   }
}
```

Il Metodo `siImmerge()` ha un *Argomento* chiamato `quantoProfondo` che ci dice di quanti metri il pesce si è immerso. Abbiamo dichiartato anche la variabile `profonditaCorrente` che memorizza e modifica la profondità corrente del pesce ogni volta che viene invocato il Metodo `siImmerge()`. Questo Metodo restituirà, alla Classe chiamante, anche il valore corrente (dopo le modifiche) della variabile `profonditaCorrente`.

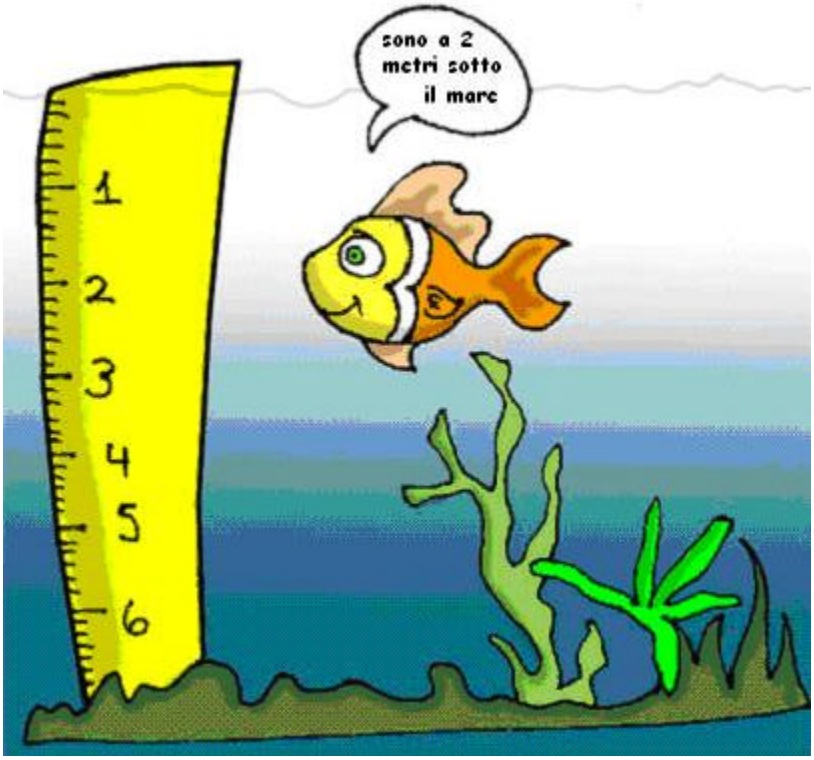
Ora creiamo un'altra Classe e chiamiamola `PadroneDelPesce` che dovrebbe essere scritta come l'esempio seguente:

```
public class PadroneDelPesce {

	public static void main(String[] args) {

		Pesce mioPesce = new Pesce();

		mioPesce.siImmerge(2);
		mioPesce.siImmerge(3);

		mioPesce.dorme();
	}
}
```

Il Metodo `main()` istanzia un Oggetto di tipo Pesce e invoca due volte il suo Metodo `siImmerge()` con differenti Argomenti. Dopo di che, invoca il Metodo `dorme()`. Quando eseguiremo (*run*) la Classe `PadroneDelPesce`, appariranno sullo schermo i seguenti messaggi:

```
Mi immergo per 2 metri
Sono a 2 metri sotto il mare
Mi immergo per 3 metri
Sono a 5 metri sotto il mare
Buona notte, ci vediamo domani
```

Avete notato che, insieme con i Metodi definiti nella Classe `Pesce`, la Classe `PadroneDelPesce` invoca anche dei Metodi della superclasse `AnimaleDomestico`? Questo è il punto cruciale dell'ereditarietà – non si deve copiare ed incollare codice della Classe `AnimaleDomestico` – ma si deve, soltanto, utilizzare la parola speciale `extends`, e la Classe `Pesce` potrà avere i Metodi di `AnimaleDomestico`!




Sebbene il Metodo `siImmerge()` restituisce il valore `profonditaCorrente`, la Classe `PadroneDelPesce` non usa tale valore. Ed è giusto! perchè `PadroneDelPesce` non ha bisogno di questo valore, ma potrebbero esistere altre Classi, che invocano i Metodi di `Pesce` ed hanno bisogno del valore `profonditaCorrente`. Per esempio, pensate alla classe `VigileDeiPesci` che ha bisogno di conoscere la posizione di tutti i pesci sotto il mare per evitare degli incidenti ✍.

## Sovrascrittura (Overriding) dei Metodi

Come è noto, i pesci non parlano (o almeno non lo fanno ad alta voce ✍). Ma la Classe `Pesce` è stata ereditata dalla Classe `AnimaleDomestico` che fornisce il Metodo `dice()`. Questo

significa che non c'è niente che ci vieti di scrivere qualcosa del genere:

```
mioPesce.dice();
```

Bene! Il nostro pesce inizierà a parlare… Se non vogliamo che questo accada, la Classe `Pesce` dovrà *sovrascrivere* il Metodo `dice()` di `AnimaleDomestico`. Come? Vediamo un po': se dichiariamo un Metodo in una sottoclasse con esattamente la stessa "firma" di uno della sua superclasse, allora il Metodo della sottoclasse sarà utilizzato al posto di quello della superclasse. Aggiungiamo il Metodo `dice()` alla Classe `Pesce`.

```
public String dice(String unaFrase){
  return "Non sai che i pesci non parlano?";
}
```

Ora possiamo aggiungere la linea seguente al Metodo `main()` della Classe PadroneDelPesce:

```
mioPesce.dice("Ciao");
```

Eseguendo (*run*) il programma vedremo che il risultato sarà la visualizzazione della frase:

```
Non lo sai che i pesci non parlano?
```

Questo prova che il Metodo `dice()` di `AnimaleDomestico` è stato *sovrascritto* (*overridden*), o in altre parole sostituito.

Se la "firma" di un Metodo include la parola speciale `final`, allora questo Metodo non può essere sovrascritto, per esempio:

```
final public void dorme(){…}
```

Wow! Abbiamo imparato molte cose in questo capitolo! Ora prendiamoci una pausa!

## Approfondimenti

1.Tipi di Dati in Java:
http://www.java-net.it/jmonline/cap3/variabili.htm

2.Ereditarietà:
http://www.java-net.it/jmonline/cap1/erditarieta.htm

## Esercizi

1. Crea una nuova Classe `Automobile` con i seguenti Metodi:

```
public void parte()
public void siFerma()
public int guida(int perQuantoTempo)
```

Il Metodo `guida()` deve restituire il totale della distanza percorsa dall'automobile nel tempo specificato. Usa la formula seguente per calcolare di volta in volta la distanza percorsa:

```
distanza = perQuantoTempo*60;
```

2. Scrivi un'altra Classe chiamata `Proprietario` che crea un'istanza di un Oggetto di tipo `Automobile` ed invoca i suoi Metodi. Il risultato di ogni invocazione deve essere scritto sullo schermo utilizzando l'istruzione `System.out.println()`.

## Esercizi avanzati

Crea una sottoclasse di `Automobile` chiamata AutomobileDi`JamesBond` e sovrascrivi il Metodo `guida()`. Usa la seguente formula per calcolare la distanza percorsa nel tempo specificato:

```
distanza = perQuantoTempo*180;
```

Sii creativo, scrivi qualche messaggio divertente!