Lezione 8

Classi e oggetti

La classe è il più importante costrutto sintattico tra quelli disponibili in Java. Non è possibile realizzare alcun programma senza definire almeno una classe e senza servirsi di eventuali altre classi realizzate in proprio o fornite da terze parti. Le classi, d'altra parte, sono il fulcro della programmazione orientata agli oggetti, e Java è un linguaggio completamente orientato agli oggetti. Comprendere i meccanismi di funzionamento delle classi è di fondamentale importanza per manipolare consapevolmente il linguaggio e gli elementi della sua piattaforma.
A partire da questa lezione, dunque, ci si addentra finalmente nella caratteristica più interessante e proficua di Java: la programmazione orientata agli oggetti, intesa nel senso più stretto del termine.

8.1 - Introduzione alle classi

Radunando le nozioni che sono state sparse nel corso delle lezioni precedenti, è possibile inquadrare alcuni punti fissi, utili per decretare cosa sia una classe, come debba essere utilizzata e quale sia l'implicito rapporto tra le classi e gli oggetti:

1. Una *classe* è un modello per una categoria di oggetti. Definendo una classe, si rende automaticamente disponibile un nuovo tipo di dato, che può essere sfruttato e manipolato come fosse parte integrante del linguaggio o della sua piattaforma.
2. Ciascun *oggetto* è una *istanza* della sua classe di appartenenza. Ad esempio, la classe *String*, inclusa nella piattaforma standard di Java 2, definisce il modello di comportamento di ogni oggetto di tipo *String*. Le stringhe, dunque, sono istanze della classe *String*. Lo stesso può avvenire con tutte classi realizzate in proprio: una volta completata la stesura del codice di una classe, diventa possibile usufruire di tutte le istanze che si desiderano.
3. Le classi favoriscono la riusabilità del codice. Se, per un dato programma, è stato necessario definire un nuovo tipo di dato, la medesima definizione può opzionalmente essere reimpiegata in altri progetti. Una esemplare dimostrazione di questo concetto è fornita dalle classi inglobate nella stessa piattaforma standard di Java 2, che ciascun programmatore può sfruttare a proprio piacimento. Quale programma, in fondo, non si serve delle stringhe e, dunque, della classe *String*? Java favorisce la riusabilità del codice imponendo che ogni singola classe venga racchiusa in un sorgente a sé stante, facendo in modo che ciascuna classe sia compilata in un *.class* corrispondente e permettendo il raggruppamento di classi tra loro collegate all'interno dei pacchetti.

Messi in chiaro questi aspetti fondamentali, diventa possibile approfondire l'argomento.

Il costrutto *class*, indispensabile per definire una classe, va impiegato attenendosi al seguente generico modello:

```
class NomeClasse {

  tipo proprietà1;
  tipo proprietà2;
```

```
  ...

  tipo metodo1(argomenti) {
    ...
  }

  tipo metodo2(argomenti) {
    ...
  }

  ...

}
```

Esistono alcune convenzioni che è utile seguire:

1. Un programma Java è comunemente costituito da più classi cooperanti. Il sorgente di ciascuna classe deve essere racchiuso all'interno di un apposito file, da nominare alla stessa maniera della classe, con l'aggiunta dell'estensione *.java*. Il sorgente della classe *Prova*, ad esempio, dovrà essere conservato in un file chiamato *Prova.java*.
2. Alle classi si è soliti assegnare dei nomi completamente in minuscolo, ad eccezione della prima lettera di ogni parola che compone l'identificatore, resa sempre in maiuscolo. *Automobile*, *Persona* e *ContoCorrente*, secondo questa convenzione, sono identificatori validi per una classe.
3. Per gli identificatori che contraddistinguono i metodi e le proprietà di una classe, valgono le medesime convenzioni in vigore per le variabili (cfr. **Lezione 4**).

8.2 - Classi e oggetti, un binomio per rappresentare la realtà

La programmazione orientata agli oggetti, in certi aspetti, si avvicina di più alla filosofia che non all'informatica. Questa considerazione è intenzionalmente esagerata e volutamente provocatoria, ma racchiude del vero. I costrutti tipici dei linguaggi orientati agli oggetti dispongono di caratteristiche utili per allestire una efficiente rappresentazione del reale, almeno nei termini in cui l'uomo è capace di schematizzare e catalogare ciò che lo circonda. All'interno di questa ottica, la classe è l'idea che permea una categoria di oggetti, mentre le sue istanze sono gli oggetti realmente esistenti.
Si pensi, in termini generici, ad una chitarra, senza però immaginare un preciso modello. La generica idea della chitarra comprende alcuni elementi che distinguono ogni chitarra da tutti gli altri oggetti esistenti. Ad esempio, una chitarra è uno strumento musicale che ha un manico, un corpo e delle corde. Ogni chitarra può compiere una particolare azione, ossia suonare. Grazie a queste caratteristiche comuni, diventa immediatamente possibile identificare qualsiasi modello di chitarra, anche se non noto a priori.
In Java, l'idea della chitarra può essere implementata con una classe:

```
class Chitarra {

  // Proprietà che rappresenta il numero delle
  // corde di cui è dotata la chitarra.
  int quanteCorde;

  // Proprietà per rappresentare il tipo di
  // chitarra, ad esempio "acustica" oppure
```

```
  // "elettrica".
  String tipo;

  // Metodo che esprime un'azione che la chitarra
  // può compiere, ossia suonare.
  void suona() {
    System.out.println("do re mi fa sol la si");
  }

}
```

La singola idea di chitarra, nella pratica, non compie nulla e non ha un riscontro reale, così come la classe appena vista non rappresenta un programma completo. Il codice visto, pertanto, fornisce semplicemente il nuovo tipo di dato nominato *Chitarra*.
Ora che si dispone di un nuovo modello di dati, è possibile sfruttare la classe per ottenere degli oggetti di tipo Chitarra o, per meglio dire, delle istanze della classe Chitarra. Lo si può fare dall'interno di una seconda classe, chiamata *TestChitarra*, creata al solo scopo di contenere un metodo *main()* che consenta l'esecuzione del programma:

```
class TestChitarra {

  public static void main(String[] args) {

    Chitarra primaChitarra = new Chitarra();
    primaChitarra.quanteCorde = 6;
    primaChitarra.tipo = "elettrica";
    System.out.println(
      "Ho una chitarra con " +
      primaChitarra.quanteCorde + " corde."
    );
    System.out.println(
      "Si tratta di una chitarra " +
      primaChitarra.tipo + "."
    );
    System.out.println("Quando suona fa:");
    primaChitarra.suona();

    Chitarra secondaChitarra = new Chitarra();
    secondaChitarra.quanteCorde = 12;
    secondaChitarra.tipo = "acustica";
    System.out.println(
      "Ho una chitarra con " +
      secondaChitarra.quanteCorde + " corde."
    );
    System.out.println(
      "Si tratta di una chitarra " +
      secondaChitarra.tipo + "."
    );
    System.out.println("Quando suona fa:");
    secondaChitarra.suona();

  }

}
```

CLASSPATH
Durante la compilazione di un programma Java suddiviso in più classi, alcuni sistemi potrebbero restituire un errore del tipo:

```
cannot resolve symbol
```

Lo stesso potrebbe avvenire durante l'esecuzione, con un messaggio del tipo:

```
java.lang.NoClassDefFoundError
```

Se il codice è stato digitato correttamente, ambo i problemi sono causati dalla mancata configurazione della variabile d'ambiente *CLASSPATH*, che non i tutti i sistemi viene impostata automaticamente. Nel caso si dovesse manifestare tale problema, la soluzione è semplice. Seguendo le medesime istruzioni presentate per la configurazione della variabile di ambiente *PATH* (cfr. **Lezione 2**), si vada a verificare il contenuto di *CLASSPATH*. Se tale variabile non esiste, sarà necessario crearla, includendo al suo interno un semplice punto, che indica la directory corrente. Se la variabile *CLASSPATH* è già definita, il punto deve essere aggiunto come primo elemento del percorso, usando un carattere di punto e virgola come separatore:

```
.;PrecedenteContenuto
```

Eventuali altri problemi di configurazione, che possono dipendere tanto dal sistema quanto dalla versione di Java impiegata, possono essere risolti consultando la documentazione ufficiale della piattaforma J2SE.

Dopo aver creato i due file *Chitarra.java* e *TestChitarra.java*, e dopo averli riposti nella medesima directory, è possibile tentare la compilazione:

```
javac TestChitarra.java
```

Non è necessario specificare esplicitamente la compilazione di *Chitarra.java*: il compilatore di Sun risolve automaticamente le dipendenze, andando a ricercare per proprio conto tutti i sorgenti necessari. Nel caso si dovessero ricevere degli errori che indicano l'impossibilità di trovare la classe *Chitarra*, si verifichi il contenuto della variabile di ambiente chiamata *CLASSPATH* (cfr. **box laterale**).

Effettuata la compilazione, è possibile eseguire *TestChitarra*:

```
java TestChitarra
```

L'output mostrato sarà il seguente:

```
Ho una chitarra con 6 corde.
Si tratta di una chitarra elettrica.
Quando suona fa:
do re mi fa sol la si
Ho una chitarra con 12 corde.
Si tratta di una chitarra acustica.
Quando suona fa:
do re mi fa sol la si
```

Ciò che è stato fatto, a livello logico, è illustrato in **Figura 8.1**.

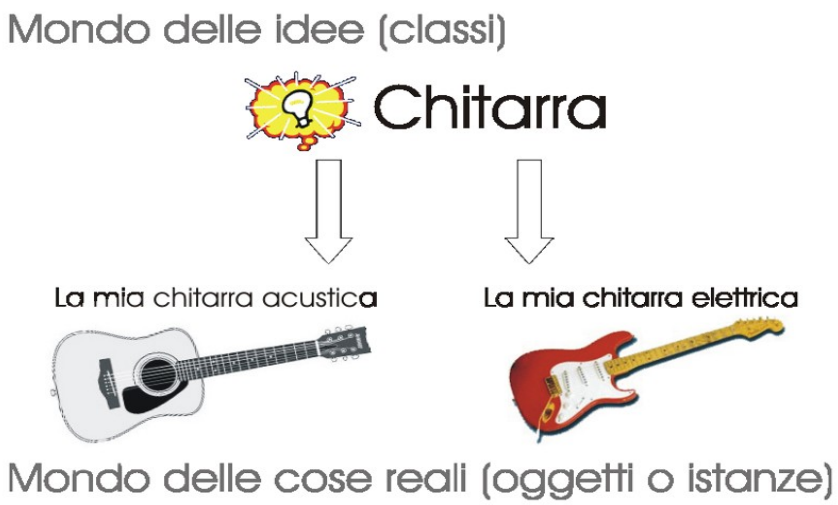

Figura 8.1
La classe *Chitarra* stabilisce un modello generico per la rappresentazione di ogni chitarra. Le singole istanze della classe, ossia gli oggetti di tipo *Chitarra*, rappresentano le peculiari chitarre realmente esistenti. Il rapporto che corre tra una classe e le sue istanze, riassumendo, è lo stesso che corre tra l'idea di una categoria di oggetti e gli oggetti stessi.

Per prima cosa, attraverso la classe *Chitarra*, è stata generata l'idea di base che può potenzialmente essere impiegata per rappresentare qualsiasi chitarra esistente. Quindi, a scopo dimostrativo, sono state create ed impiegate due istanze della classe, per rappresentare dei modelli realmente esistenti e pertanto concretamente impiegabili.

A livello di codice, è essenziale notare la sintassi adoperata per istanziare la classe *Chitarra*:

```
Chitarra primaChitarra = new Chitarra();
```

L'operatore *new* serve per generare una nuova istanza di una classe.

8.3 - Gestione della memoria e garbage collection

La memoria utilizzata per la gestione dei valori impiegati in un programma Java è suddivisa in due aree. La prima è nominata *stack*, ed in essa trovano posto i valori di tipo predefinito (*byte*, *short*, *int*, *long*, *char*, *float*, *double* e *boolean*). Gli ambiti di visibilità delle variabili, ed i blocchi di codice che ne determinano i confini, sono strettamente legati al funzionamento dello stack. Quando si dichiara una variabile di tipo *int*, ad esempio, all'interno dello stack vengono riservati ed occupati i 32 bit necessari per la rappresentazione del valore. Nel momento in cui tale variabile conclude il proprio ciclo vitale, ossia quando l'esecuzione del blocco che la contiene è ultimato, il riferimento va perduto, e la memoria ad esso collegata viene automaticamente ed istantaneamente ripulita. Così avviene per tutti i valori di tipo predefinito: non appena le variabili che li contengono escono dal loro ambito di visibilità, la memoria associata viene ripulita.
La seconda area di memoria si chiama *heap*, e qui trovano spazio gli oggetti. Le informazioni memorizzate nello heap, a differenza dei valori di tipo predefinito, godono di un ciclo vitale che non dipende dall'ambito di visibilità delle variabili ad esse collegate. In pratica, le variabili muoiono appena escono dal loro ambito di visibilità, ma i dati cui esse si riferiscono continuano ad esistere in memoria. Di tanto in tanto, secondo l'impegno della CPU e del quantitativo di memoria residua, l'ambiente di esecuzione di Java avvia uno speciale processo chiamato *garbage collection* (*raccolta dei rifiuti*), il cui compito è rimuovere dallo heap tutti gli oggetti non più utilizzabili. Il sofisticato algoritmo che si occupa di questa operazione, chiamato *garbage collector*, può anche essere attivato manualmente:

```
System.gc();
```

Il più delle volte, conviene affidarsi al giudizio dell'ambiente di esecuzione, lasciando ad esso il compito di stabilire quando sia conveniente attivare il processo.

Questa distinzione tra stack e heap, benché di per sé trasparente poiché completamente gestita dall'ambiente di esecuzione, ha un importante effetto sul linguaggio Java. I dati di tipo predefinito, infatti, vengono gestiti *per valore*, mentre gli oggetti sono gestiti *per riferimento*.

Le variabili di tipo valore (che risiedono nello stack) sono un tutt'uno con il valore in esse conservato: quando si crea una variabile, si deve inizializzare il suo valore; quando la stessa viene distrutta, il valore associato è automaticamente ed istantaneamente rimosso dalla memoria; quando si assegna un valore ad un variabile, tale valore viene copiato nell'area di memoria associata alla variabile. Tutto ciò può essere facilmente dimostrato:

```
int a = 5;
int b = a;
System.out.println("a: " + a);
System.out.println("b: " + b);
b += 2;
System.out.println("a: " + a);
System.out.println("b: " + b);
```

L'output prodotto è il seguente:

```
a: 5;
b: 5;
a: 5;
b: 7;
```

Nel momento in cui il contenuto di *b* è stato incrementato, *a* non ha subito alcuna modifica. L'istruzione

```
b = a;
```

esegue semplicemente una copia su *b* del valore contenuto in *a*. Conclusa l'operazione di copia, non vi è più alcuna relazione tra il valore di *a* ed il valore di *b*.

Le variabili di tipo oggetto, al contrario, funzionano diversamente, giacché sono gestite per riferimento. Riusando la definizione della classe *Chitarra* presentata nel paragrafo precedente, si sperimenti la seguente dimostrazione:

```
class TestRiferimento {

  public static void main(String[] args) {
    // Dichiarazione ed inizializzazione
    // della chitarra c1.
    Chitarra c1 = new Chitarra();
    c1.quanteCorde = 6;
    c1.tipo = "elettrica";
    // Si definisce una seconda variabile di
    // tipo Chitarra, usando il valore di c1
    // come inizializzatore.
    Chitarra c2 = c1;
    // Si stampano i dati in output.
    System.out.println(
      "c1 ha " + c1.quanteCorde + " corde."
    );
    System.out.println(
      "c1 e' una chitarra " + c1.tipo + "."
    );
    System.out.println(
      "c2 ha " + c2.quanteCorde + " corde."
    );
    System.out.println(
      "c2 e' una chitarra " + c2.tipo + "."
    );
    // Ora vengono variate le proprietà di c2.
```

```
    c2.quanteCorde = 12;
    c2.tipo = "acustica";
    // Si ripete la stampa dei dati in output.
    System.out.println(
      "c1 ha " + c1.quanteCorde + " corde."
    );
    System.out.println(
      "c1 e' una chitarra " + c1.tipo + "."
    );
    System.out.println(
      "c2 ha " + c2.quanteCorde + " corde."
    );
    System.out.println(
      "c2 e' una chitarra " + c2.tipo + "."
    );
  }

}
```

L'output prodotto è il seguente:

```
c1 ha 6 corde.
c1 e' una chitarra elettrica.
c2 ha 6 corde.
c2 e' una chitarra elettrica.
c1 ha 12 corde.
c1 e' una chitarra acustica.
c2 ha 12 corde.
c2 e' una chitarra acustica.
```

Come è possibile osservare, l'aver assegnato il valore di *c1* alla variabile *c2* non ha sottinteso una copia dell'oggetto. Variando le proprietà di *c2*, infatti, si modificano anche quelle di *c1*, e viceversa. Le variabili *c1* e *c2* si riferiscono alla medesima area di memoria presente nello heap, e pertanto sono due alias di un unico oggetto. Le variabili di tipo oggetto, in definitiva, contengono dei riferimenti verso gli oggetti che rappresentano. Copiando il contenuto di una variabile di questo tipo si ottiene una copia del riferimento, e non una copia dell'oggetto. Per copiare gli oggetti, è necessario impiegare dei metodi appositamente studiati. Chi conosce C e C++ può pensare alle variabili di tipo oggetto come ad una sorta di puntatori impliciti.

Il *modus operandi* del garbage collector, a questo punto, dovrebbe apparire più chiaro. Quando una variabile di tipo oggetto esce dal proprio ambito di visibilità, viene automaticamente perso il riferimento in essa contenuto, ma non l'oggetto riferito. Ad un oggetto possono corrispondere più alias, pertanto sarebbe incorretto ripulire la memoria alla semplice scomparsa di un riferimento. La pulizia della memoria è completamente affidata al garbage collector, il cui compito è scorrere lo heap di tanto in tanto, ricercando ed eliminando tutti quegli oggetti rimasti completamente privi di alias e, per questo, definitivamente inservibili.

8.4 - Proprietà e metodi a prima vista

Un primo approccio alle proprietà e ai metodi degli oggetti è stato svolto nel corso della lezione precedente. Riassumendo:

Inizializzazione automatica delle proprietà
Le proprietà di un oggetto, se non specificato diversamente, vengono inizializzate in automatico dall'ambiente di esecuzione. Le proprietà di carattere numerico (*byte*, *char*, *short*, *int*, *long*, *float* e *double*) vengono inizializzate con il valore 0. Le proprietà booleane sono automaticamente impostate su *false*. I riferimenti agli oggetti ricevono per default il valore *null*. L'affermazione può essere facilmente dimostrata:

```
class TestInizializzazione {

  byte a;
  char b;
  short c;
  int d;
  long e;
  float f;
  double g;
  boolean h;
  String i;

  public static void main(String[] args) {
    TestInizializzazione t =
      new TestInizializzazione();
    System.out.println(t.a);
    System.out.println((int)t.b);
    System.out.println(t.c);
    System.out.println(t.d);
    System.out.println(t.e);
    System.out.println(t.f);
    System.out.println(t.g);
    System.out.println(t.h);
    System.out.println(t.i);
  }
}
```

1. Le proprietà sono delle variabili proprie di un oggetto, ad esso strettamente collegate.
2. I metodi sono delle azioni, ossia dei blocchi di codice, che l'oggetto può eseguire su richiesta.
3. Per accedere alle proprietà e ai metodi di un oggetto deve essere sfruttata la notazione puntanta, nelle forme:

```
riferimento.nomeProprietà
riferimento.nomeMetodo(argomenti)
```

Sulle proprietà non c'è molto altro da dire, in questo momento. Semplicemente, possono essere variabili qualsiasi, indifferentemente di tipo predefinito oppure di tipo oggetto. L'universo dei metodi, invece, ancora deve essere esplorato.
Un metodo completamente definito si presenta sempre alla seguente maniera:

```
tipoRestituito nomeMetodo(argomenti) {
  // corpo del metodo
}
```

Escludendo il corpo, si ottiene la firma del metodo:

```
tipoRestituito nomeMetodo(argomenti)
```

Il tipo restituito specifica il gruppo di appartenenza del valore elaborato dal metodo. Un metodo può restituire un valore di tipo predefinito oppure un riferimento ad un oggetto, in base alle esigenze e a discrezione del programmatore. Non sempre, ad ogni modo, si ha la necessità di restituire un valore al codice chiamante. In casi come questo, il tipo restituito sarà *void*. Un metodo di tipo *void* non restituisce nulla.
La lista degli argomenti è un insieme di coppie tipo identificatore, separate da virgole. Gli argomenti (o parametri) sono speciali variabili il cui contenuto cambia in base ai valori specificati quando il metodo è richiamato. Un metodo può anche non accettare alcun parametro. In questo caso, la lista degli argomenti sarà vuota, e la firma del metodo ricalcherà la seguente:

```
tipoRestituito nomeMetodo()
```

Il corpo di un metodo è un blocco di codice che viene eseguito nel momento in cui il metodo è richiamato. All'interno di tale blocco è possibile riferirsi agli argomenti forniti in tutta

naturalezza, come se fossero variabili definite all'interno del blocco stesso. Stando in questa posizione, è possibile accedere alle proprietà e agli altri metodi dell'oggetto semplicemente richiamandone il nome, senza dover usare un riferimento seguito da notazione puntata. All'interno della classe *Chitarra*, ad esempio, potrebbe essere definito il seguente metodo:

```
void scriviQuanteCorde() {
  System.out.println(quanteCorde);
}
```

Come è possibile osservare, si è fatto riferimento alla proprietà *quanteCorde* semplicemente digitandone l'identificatore. Dall'esterno della classe, invece, è necessario scrivere:

```
nomeOggetto.quanteCorde
```

La parola chiave *return* ha un duplice scopo: interrompe l'esecuzione del metodo che la contiene (è un'istruzione di salto) e restituisce un valore compatibile con il tipo dichiarato nella firma del metodo. Si vada ad arricchire ulteriormente la classe *Chitarra*:

```
int dimmiQuanteCordeHai() {
  return quanteCorde;
}
```

Fatto ciò, dall'interno di un'altra classe, diventa possibile scrivere:

```
Chitarra c = new Chitarra();
// ...
int corde = c.dimmiQuanteCordeHai();
System.out.println("Numero corde: " + corde);
```

I metodi *void* possono usare *return* come semplice istruzione di salto, senza specificare alcun valore di ritorno:

```
void nomeMetodo() {
  System.out.println("Vengo eseguito");
  return;
  System.out.println("Non vengo eseguito");
}
```

Un metodo come quello appena mostrato, in realtà, non può essere compilato, giacché il compilatore si accorge del salto incondizionato imposto da return, giudicando inutile l'istruzione successiva. Il salto *return* deve apparire sempre in forme condizionali del tipo:

```
void nomeMetodo() {
  // ...
  if (qualcosa) return;
  // ...
}
```

L'utilizzo degli argomenti è piuttosto intuitivo. Ecco un paio di generici esempi, destinati alla classe *Chitarra*:

```
void cambiaNumeroCorde(int n) {
  // Cambia il numero delle corde.
  quanteCorde = n;
}

void suonaNota(int qualeNota, int quanteVolte) {
  // Questo metodo suona una nota (espressa come
  // intero che spazia da 0 a 6) per un certo numero
  // di volte.
  String[] note = {
    "Do", "Re", "Mi", "Fa", "Sol", "La", "Si"
  };
  // Se il parametro qualeNota esprime un valore non
  // valido (minore di 0 o maggiore di 6) non viene
  // eseguito nulla.
  if (qualeNota < 0 || qualeNota > 6) return;
  // Il parametro quanteVolte deve necessariamente
  // essere maggiore di 0.
  if (quanteVolte <= 0) return;
  // Se l'esecuzione giunge a questo punto, la nota
  // può essere suonata come richiesto.
  for (int i = 0; i < quanteVolte; i++)
    System.out.println(note[qualeNota]);
}
```

Dall'esterno, è possibile richiamare i nuovi metodi appena definiti:

```
Chitarra c = new Chitarra();
// Imposta su 6 il numero delle corde.
c.cambiaNumeroCorde(6);
// Suona quattro volte la nota re.
c.suonaNota(1, 4);
```

8.5 - Costruttori

I *costruttori* sono speciali metodi il cui compito è inizializzare gli oggetti. Vengono eseguiti non appena l'oggetto è stato creato, ed il codice contenuto al loro interno può essere sfruttato per fornire una prima configurazione all'oggetto. Un costruttore si dichiara definendo un metodo senza alcun tipo di ritorno (neanche *void*), che abbia lo stesso nome della classe che lo contiene (rispettando minuscole e maiuscole):

```
class NomeClasse {

  // Il seguente è un costruttore.
  NomeClasse() {
    // Corpo del costruttore.
  }

}
```

La classe *Chitarra*, tornando al consueto esempio della lezione, dispone di due proprietà: *quanteCorde* e tipo. Quando si crea un oggetto di tipo *Chitarra*, ambo le proprietà devono essere impostate manualmente, affinché l'oggetto abbia senso compiuto:

```
Chitarra c = new Chitarra();
```

```
c.quanteCorde = 6;
c.tipo = "elettrica";
```

Un'operazione del genere non è al massimo della praticità. Una buona idea, pertanto, è stabilire dei valori di default, che un costruttore può impostare automaticamente:

```
Chitarra() {
  quanteCorde = 6;
  tipo = "elettrica";
}
```

Come già detto, il costruttore è eseguito automaticamente dopo la creazione dell'oggetto. Pertanto, per ottenere una chitarra elettrica con sei corde, è ora sufficiente scrivere:

```
Chitarra c = new Chitarra();
```

Tutte le chitarre istanziate, grazie al nuovo costruttore, saranno di tipo elettrico con sei corde. Ovviamente, è probabile che l'utente della classe *Chitarra* intenda utilizzare anche altri tipi di strumento. Lasciando invariato il costruttore appena visto, bisognerebbe tornare a modificare manualmente le due proprietà disponibili:

```
// c è una chitarra elettrica a sei corde
Chitarra c = new Chitarra();
// Ora viene mutata in un'acustica a 12
c.quanteCorde = 12;
c.tipo = "acustica";
```

La pratica, ancora una volta, è poco elegante. Conviene sostituire il costruttore realizzato in precedenza con una variante basata su una lista di due argomenti:

```
Chitarra(int qc, String t) {
  quanteCorde = qc;
  tipo = t;
}
```

Da questo momento in poi, il numero di corde ed il tipo della chitarra dovranno sempre essere specificati direttamente alla creazione dell'oggetto:

```
Chitarra c1 = new Chitarra(6, "elettrica");
Chitarra c2 = new Chitarra(12, "acustica");
```

Avendo eliminato il vecchio costruttore, quello privo di argomenti, non sarà più possibile scrivere:

```
Chitarra c = new Chitarra();
```

A questo punto, infatti, non esiste più un costruttore senza argomenti cui appellarsi. In generale, valgono le seguenti norme:

1. Se una classe non viene dotata di alcun costruttore, automaticamente è reso

disponibile un costruttore senza argomenti, che non compie alcuna operazione, ma che consente di istanziare l'oggetto senza problemi.
2. Se si realizza un costruttore con argomenti, il costruttore implicito senza argomenti non viene più reso disponibile.

Una classe, ad ogni modo, può essere dotata di più costruttori che differiscono nella lista degli argomenti accolti:

```
class Chitarra {

  int quanteCorde;
  String tipo;

  Chitarra() {
    // Chitarra elettrica a sei corde.
    quanteCorde = 6;
    tipo = "elettrica";
  }

  Chitarra(int qc, String t) {
    // Chitarra personalizzata.
    quanteCorde = qc;
    tipo = t;
  }

}
```

A questo punto, ambo i costruttori possono essere indifferentemente sfruttati:

```
// Chitarra di default, a sei corde elettrica.
Chitarra c1 = new Chitarra();
// Chitarra "parametrica".
Chitarra c2 = new Chitarra(12, "acustica");
```

8.6 - La parola chiave this

Talvolta può capitare che un oggetto debba far riferimento a se stesso. A tale scopo è stata introdotta la parola chiave *this*. Con *this* (in inglese, *questo*) si intende sempre l'oggetto corrente. Tale parola chiave può essere sfruttata all'interno del corpo di un metodo o di un costruttore. Si consideri la seguente variante dell'ultimo costruttore esaminato per la classe Chitarra:

```
Chitarra(int qc, String t) {
  this.quanteCorde = qc;
  this.tipo = t;
}
```

In casi come questo, l'utilizzo di *this*, benché superfluo, migliora la leggibilità del codice. Ad ogni modo, esistono anche dei casi in cui l'impiego di *this* è assolutamente indispensabile, ad esempio quando l'oggetto corrente deve essere fornito come argomento ad un metodo di un altro oggetto.
Un'altra reale utilità di *this* si riscontra nel momento in cui sorge un conflitto di nomi tra le

proprietà di un oggetto e le variabili utilizzate localmente nel corpo di un metodo o di un costruttore. Java, in questa situazione, non impedisce l'impiego di identificatori potenzialmente in conflitto. Si consideri il seguente esempio:

```
class TestThis {

  int n;

  TestThis(int n) {
    this.n = n;
  }

}
```

L'argomento intero accettato dal costruttore di questa classe si chiama *n*, esattamente come la proprietà dell'oggetto dichiarata poco sopra il costruttore stesso. Ad ogni modo, il conflitto è facilmente risolvibile: rimanendo nel corpo del costruttore, l'identificatore n sarà associato al parametro ricevuto, mentre *this.n* farà riferimento alla proprietà n dell'oggetto corrente.

8.7 - La parola chiave null

Particolare significato ricopre la parola chiave *null*. Nel momento in cui si dichiara una variabile di tipo oggetto, è possibile inizializzare il suo contenuto con *null*:

```
Chitarra c = null;
```

Le inizializzazioni automatiche delle variabili oggetto impiegano sempre il valore *null*. Un esempio è fornito dagli array (cfr. **Lezione 7**):

```
Chitarra[] tanteChitarre = new Chitarra[5];
for (int i = 0; i < tanteChitarre.length; i++)
  System.out.println(tanteChitarre[i]);
```

Il codice appena mostrato stampa in output la parola *null* per cinque volte di seguito. Questo avviene poiché gli elementi dell'array *tanteChitarre* sono stati automaticamente posti su *null*.
La parola *null* indica l'assenza di riferimento. Una variabile oggetto che contiene il valore *null*, quindi, non si riferisce a nessun oggetto realmente esistente. Pertanto, un codice come il seguente è scorretto:

```
Chitarra c = null;
c.suona();
```

Il codice può essere compilato, giacché sintatticamente corretto. Il problema, pertanto, emergerà durante l'esecuzione. L'errore che si riceve è mostrato di seguito:

```
java.lang.NullPointerException
```

In definitiva, quindi, per le variabili oggetto vale la seguente regola: "prima dichiari, poi inizializzi con un oggetto valido realmente esistente, poi utilizzi".