Lezione 3
Regole di base, commenti, istruzioni e blocchi

Sintassi e semantica

Sintassi. Letteralmente, è la disposizione delle parole in un discorso. Relativamente ad un linguaggio di programmazione, è l'insieme delle regole che determina come le singole istruzioni debbano essere disposte e come i loro elementi interni debbano essere ordinati. La sintassi è un aspetto che riguarda la forma del codice.

Semantica. Letteralmente, è il significato di un'espressione linguistica. Relativamente ad un linguaggio di programmazione, è l'effetto di ciascun elemento ed il suo contributo al funzionamento complessivo del software. La semantica è un aspetto che riguarda il significato del codice.

In breve, le regole semantiche suggeriscono cosa debba essere usato in ogni particolare situazione, mentre le regole sintattiche indicano come i singoli elementi debbano essere distribuiti e messi in relazione tra loro.

Quando si esamina un linguaggio di programmazione, è difficile isolare un argomento per poterlo trattare indipendentemente dagli altri. Il più delle volte, soprattutto durante le prime fasi dello studio, è necessario fare riferimento a nozioni ancora non discusse.
Con questa lezione comincia l'analisi delle regole sintattiche e semantiche di Java. L'esempio impiegato come filo conduttore sfrutta diversi costrutti non immediatamente descrivibili. Il lettore, concentrandosi soprattutto sugli argomenti cardine di ciascun paragrafo, non deve sentirsi disorientato dalle nozioni e dalle citazioni riferite ad argomenti di stampo più avanzato.

3.1 - Un esempio illustrato

In un file di testo chiamato *Benvenuto.java* (d'ora in avanti si sottintenderà che ad ogni sorgente deve essere associato il nome della classe in esso contenuta), si vada ad introdurre il seguente codice:

```
import java.io.*;

public class Benvenuto {

	public static void main(String[] args) throws Exception {
		// Stampo una domanda sulla periferica di output.
		System.out.print("Come ti chiami? ");
		// Creo un oggetto utile per leggere dalla periferica di input.
		BufferedReader input = new BufferedReader(
			new InputStreamReader(System.in)
		);
		// Leggo una riga dalla periferica di input.
		String nome = input.readLine();
		// Stampo un saluto personalizzato.
		System.out.println(
			"Ciao " + nome + ", benvenuto nel mondo di Java!"
		);
	}

}
```

Si compili il sorgente con il comando:

```
javac Benvenuto.java
```

Quindi, come mostrato in **Figura 3.1**, si avvii il bytecode generato dal compilatore,

digitando:

```
java Benvenuto
```

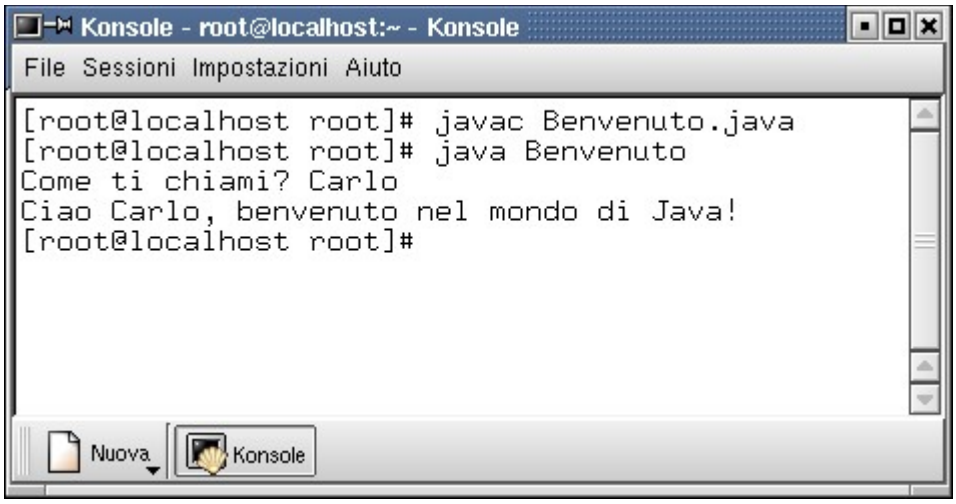

Figura 3.1
Compilazione ed esecuzione della classe *Benvenuto*.

Escludendo i costrutti per il momento non interessanti, è possibile osservare come il programma muova tre passi:

1. Sulla periferica di output predefinita (generalmente il monitor) viene mostrato il testo "Come ti chiami?", seguito da uno spazio vuoto.
2. Sfruttando la periferica di input predefinita (generalmente la tastiera), è possibile rispondere alla domanda posta, digitando il proprio nome.
3. Sulla periferica di output predefinita viene introdotto un saluto personalizzato, basato sul nome digitato in precedenza.

Queste sono le operazioni percepite dall'utente del programma. Andando a sondare il codice, però, si scopre che i passi compiuti sono quattro:

```
1. System.out.print("Come ti chiami? ");

2. BufferedReader input = new BufferedReader(
        new InputStreamReader(System.in)
   );

3. String nome = input.readLine();

4. System.out.println(
        "Ciao " + nome + ", benvenuto nel mondo di Java!"
   );
```

Le operazioni 1, 3 e 4 corrispondono a quanto riscontrato dall'utente. Il secondo passo è impiegato come meccanismo interno, necessario per eseguire l'operazione successiva.
L'esempio, oltre a mostrare la natura di un generico programma Java, insegna come ciascuna applicazione possa essere realizzata sommando delle operazioni di natura elementare, alcune destinate all'interazione con l'utente, altre necessarie per il corretto funzionamento interno.

3.2 - Commenti

Java, analogamente alla maggior parte degli altri linguaggi di programmazione, consente l'inserimento di commenti all'interno del codice. Il programma *Benvenuto* contiene quattro commenti, ognuno dei quali illustra l'effetto dell'operazione che lo segue.

```
// Stampo una domanda sulla periferica di output
System.out.print("Come ti chiami? ");
// Creo un oggetto utile per leggere dalla periferica di input
BufferedReader input = new BufferedReader(
      new InputStreamReader(System.in)
);
// Leggo una riga dalla periferica di input
String nome = input.readLine();
// Stampo un saluto personalizzato
System.out.println(
      "Ciao " + nome + ", benvenuto nel mondo di Java!"
);
```

Ricordati di ...

I commenti, mentre si lavora allo sviluppo di una classe, possono essere impiegati per aggiungere delle note temporanee utili per non dimenticarsi di un'aggiunta prevista, di una revisione necessaria o di un blocco di codice di debug da eliminare a lavoro ultimato. Mi capita spesso, mentre programmo, di utilizzare delle note del tipo:

```
// Carlo, ricordati di ...
```

Questo genere di commenti è particolarmente proficuo se combinato con l'uso di editor avanzati o di ambienti integrati. Alcuni editor, ad esempio, evidenziano con colori differenti i commenti destinati a lavorare come richiami, oppure introducono in un apposito box dei collegamenti verso le righe che li contengono. L'importante è rispettare la sintassi prevista dal particolare editor adottato. Il più delle volte, è sufficiente usare una forma del tipo:

```
// TODO: fai questo
```

In altri casi, l'opzione è liberamente personalizzabile. Pertanto, si faccia riferimento alla documentazione dell'ambiente utilizzato per scoprire in che modo venga supportata questa caratteristica.

Il compilatore ignora i commenti, come se non esistessero. Per questo è possibile impiegare il linguaggio naturale nel modo che si preferisce, senza dover sottostare a particolari regole di dizione. I commenti servono per illustrare il funzionamento di un programma a chiunque disponga del corrispondente codice sorgente. E' bene fare uso abituale dei commenti, giacché costituiscono un efficace strumento per la condivisione del codice e la manutenzione dei programmi. Non si deve credere che i commenti siano utili solo nel caso in cui si preveda di consegnare ad altri il codice sorgente di un proprio software. I programmi di vaste dimensioni, il più delle volte, hanno bisogno di costante manutenzione e di frequenti aggiornamenti. Con il trascorrere del tempo e l'incrementarsi delle righe di codice, lo sviluppatore potrebbe dimenticare in che modo lavori una particolare procedura, scritta magari sei mesi prima. I commenti, lavorando come promemoria, possono essere impiegati per facilitare le revisioni, gli ampliamenti e le correzioni.
Nell'esempio presentato, con l'intenzione di chiarire al lettore ciascuna operazione effettuata, si è deliberatamente esagerato con la quantità dei commenti introdotti, descrivendo una ad una le istruzioni Java impiegate. In casi reali, un tale eccesso di zelo non è assolutamente richiesto. Inquadrare e descrivere i passaggi salienti, ad ogni modo, è una buona abitudine.
Java supporta tre tipi di commenti: su riga singola, su più righe e commenti javadoc. I commenti su riga singola cominciano con una coppia di barre e terminano alla fine della riga. Tutto ciò che è compreso in tale intervallo viene ignorato dal compilatore:

```
// Questo è un commento su riga singola
```

Un uso come il seguente è del tutto lecito:

```
System.out.print("Ciao"); // Stampo "Ciao"
```

I commenti su più righe cominciano con una barra ed un asterisco e terminano con la combinazione inversa, vale a dire con un asterisco ed una barra. Per questo motivo, possono essere distribuiti su più righe:

```
/* Questo è
un commento
su più righe */
```

Anche i commenti multiriga possono essere accodati ad un'istruzione Java, senza alterarne il funzionamento:

```
System.out.print("Ciao"); /* Stampo
                              "Ciao" */
```

I commenti javadoc, attivabili usando come sequenza di apertura una barra e due asterischi, sono una significativa variante dei commenti multiriga:

```
/** commento javadoc,
che formalmente è una
variante dei commenti
su più righe */
```

Per il compilatore, non sussiste alcuna differenza tra i commenti javadoc ed i commenti multiriga: entrambi vengono ignorati alla medesima maniera. I commenti javadoc vengono invece considerati e valutati da un apposito strumento di documentazione, chiamato per l'appunto *javadoc*. Rispettando una particolare sintassi, è possibile far creare automaticamente dei documenti che descrivano l'impiego ed il funzionamento di ciascuna classe realizzata.

3.3 - Regole sintattiche di base

Java, per quanto riguarda le regole sintattiche più basilari, assomiglia molto a C e a C++. Per questo motivo, si dice che Java è un linguaggio *C-like* (letteralmente, *somigliante a C*). Questo comporta alcune conseguenze che sono esaminate di seguito.
Per prima cosa, Java è un linguaggio *case-sensitive*. Il compilatore distingue le lettere maiuscole da quelle minuscole. Scrivere *class*, *Class*, *CLASS* o *CLasS*, dunque, non è affatto la stessa cosa. Tale regola si applica a qualsiasi digitazione, sia essa un comando Java oppure un identificatore definito dal programmatore (il nome di una classe o di una variabile, ad esempio).
I caratteri di spaziatura (spazio, tabulazione e ritorno a capo) non ricoprono speciali significati, tranne quando vengono impiegati per separare una parola dalla successiva. Nell'esempio presentato, infatti, un comando è stato distribuito su tre righe:

```
BufferedReader input = new BufferedReader(
      new InputStreamReader(System.in)
);
```

L'operazione

```
String nome = input.readLine();
```

è del tutto equivalente alle seguenti:

```
String nome=input.readLine();

  String   nome    =  input . readLine (  ) ;

 String
     nome    =
    input
      .
readLine   (
     )
  ;
```

Tra le diverse varianti che è possibile escogitare, le uniche incorrette sono quelle che uniscono il termine *String* con l'identificativo *nome*:

```
Stringnome = input.readLine(); // Non valido!
```

In questo caso, infatti, le due parole vengono fuse, ed il compilatore non può più distinguere l'una dall'altra. I caratteri di spaziatura, dunque, assumono rilevanza quando due *token*, ossia due termini indipendenti, non possono essere separati con altri mezzi.
Il grado di libertà concesso, ad ogni modo, non implica che ci si debba sentire autorizzati ad impiegare i caratteri di spaziatura nella più totale anarchia. La caratteristica, al contrario, va utilizzata per disporre il codice in maniera ordinata, pulita e leggibile. Esistono particolari convenzioni che regolano la stesura del codice Java. Alcune di queste riguardano proprio l'utilizzo degli spazi. Benché non sia obbligatorio seguirle, la stragrande maggioranza dei programmatori le rispetta. In questo corso sono osservate le più importanti convenzioni suggerite da Sun Microsystems. Prendendo a modello gli esempi presentati, il lettore avrà modo di assorbirle ed applicarle in maniera naturale e proficua.
Alcuni altri linguaggi limitano l'uso degli spazi e dei ritorni a capo, giacché li impiegano come elementi sintattici. Java, invece, si serve di numerosi separatori specifici, come la virgola, il punto, il punto e virgola, i due punti e le parentesi tonde, quadre e graffe. Non importa, ad esempio, su quante righe sia contenuto un comando, l'importante è che termini sempre con un punto e virgola.

Attenzione alle righe lunghe!
Talvolta, può capitare di doversi servire di istruzioni particolarmente lunghe. Poiché in alcuni sistemi a carattere non esiste la possibilità di avere delle barre di scorrimento orizzontali, si consiglia di non superare mai l'ottantesima colonna, ripartendo su più righe le istruzioni che valicano il limite stabilito.

3.4 - Istruzioni e blocchi

Invadendo il campo di interesse della chimica, è possibile pensare alle istruzioni che formano un programma Java come a degli atomi appartenenti ad un corpo di grandi dimensioni. Un'istruzione, infatti, è un comando operativo che determina un effetto. Ad esempio:

```
System.out.print("Come ti chiami? ");
```

Le istruzioni semplici, come quella appena presentata, devono essere obbligatoriamente concluse con un punto e virgola.
Più atomi, insieme, possono essere miscelati per formare delle molecole. Allo stesso modo, più istruzioni semplici possono essere raccolte in istruzioni complesse. Le istruzioni complesse, dette altrimenti blocchi, sono delimitate da una coppia di parentesi graffe:

```
{
  istruzione_semplice;
  istruzione_semplice;
  istruzione_semplice;
  ...
}
```

I blocchi devono essere impiegati per chiarire l'appartenenza di una o più istruzioni ad un medesimo contesto. In Benvenuto, ad esempio, un blocco racchiude le quattro operazioni cardine del programma, associandole allo speciale comparto *main*.
Ciascun blocco, a sua volta, può essere contenuto all'interno di un super-blocco di dimensioni maggiori:

```
{
  istruzione_semplice;
  {
    istruzione_semplice;
    istruzione_semplice;
  }
  istruzione_semplice;
}
```

Il blocco *main*, tornando all'esempio principale, è contenuto nel blocco *class Benvenuto*.

Indentare il codice
La possibilità di utilizzare arbitrariamente i caratteri di spaziatura va impiegata per indentare il codice. Utilizzando, in ogni riga, dei rientri verso destra, le singole istruzioni possono essere allineate in modo da risultare meglio leggibili. Il principale uso che si fa dei rientri riguarda i blocchi delimitati da parentesi graffe:

```
{
  istruzione;
  istruzione;
  {
    istruzione;
    istruzione;
    ...
  }
  ...
}
```

Per convenzione, il contenuto di ogni blocco deve essere fatto rientrare verso destra con due, quattro o otto spazi (oppure con una tabulazione) rispetto alla posizione del blocco stesso. Gli editor facilitati aiutano lo sviluppatore in questo compito, sollevandolo dalla noia di dover digitare i rientri manualmente all'inizio di ogni nuova riga.