# F

# Linguaggio Java: compendio sintattico

Come nel resto del libro, in questo compendio sintattico usiamo un tipo di carattere equispaziato per le parole riservate del linguaggio Java, come `while`, e per i nomi di variabili, metodi, classi, e così via. Un carattere corsivo indica costrutti del linguaggio, come *condizioni* o *variabili*. Entità racchiuse tra parentesi quadre sono opzionali. Entità separate da barre verticali sono alternative tra loro. Non inserite nel vostro codice queste parentesi quadre o queste barre verticali!

Il compendio si riferisce alle parti del linguaggio Java che sono state trattate in questo libro. Per una panoramica completa sulla sintassi di Java, consultate `http://download.oracle.com/javase/10/docs/api/`.

Occorre fare molta attenzione nel distinguere i puntini che indicano qualcosa che viene omesso dalla "parola riservata" costituita da tre puntini, che compare due volte in questa appendice, nella trattazione dell'argomento "numero di parametri variabile" relativo ai metodi.

## Tipi

Un tipo è un tipo primitivo o un tipo riferimento. I tipi primitivi sono

- i tipi numerici `int`, `long`, `short`, `char`, `byte`, `float`, `double`
- il tipo `boolean`

I tipi riferimento sono

- classi, come `String` o `Employee`
- tipi enumerativi, come `enum Gender { FEMALE, MALE }`
- interfacce, come `Comparable`
- tipi di array, come `Employee[]` o `int[][]`

## Variabili

Le dichiarazioni di variabili locali hanno la forma

```
[final] Tipo nomeVariabile [= valoreIniziale];
```

Esempi

```
int n;
double x = 0;
String harry = "Harry Handsome";
Rectangle box = new Rectangle(5, 10, 20, 30);
int[] a = { 1, 4, 9, 16, 25 };
```

I nomi delle variabili sono costituiti soltanto da lettere, cifre e caratteri di sottolineatura; devono iniziare con una lettera o un carattere di sottolineatura. Nei nomi, la distinzione tra lettere maiuscole e minuscole è rilevante: `totalscore`, `TOTALSCORE` e `totalScore` sono tre variabili diverse.

L'ambito di visibilità di una variabile locale si estende dal punto della sua definizione alla fine del blocco che la racchiude.

Una variabile che venga dichiarata `final` può ricevere il proprio valore soltanto una volta.

Le variabili di esemplare saranno presentate nel paragrafo relativo alle classi.

## Espressioni

Una *espressione* è una variabile, un'invocazione di metodo o una combinazione di sotto-espressioni connesse da operatori. Ecco alcuni esempi

```
x
Math.sin(x)
x + Math.sin(x)
x * (1 + Math.sin(x))
x++
x == y
x == y && (z > 0 || w > 0)
p.x
e.getSalary()
v[i]
```

Gli operatori possono essere *unari*, *binari* o *ternari*. Un operatore unario agisce su una singola espressione, come `x++`. Un operatore binario combina due espressioni, come `x +`

y. Un operatore ternario combina tre espressioni: Java ha un solo operatore ternario, `? :` (trattato in Argomenti avanzati 5.1).

Gli operatori unari possono essere prefissi o postfissi. Un operatore prefisso viene scritto prima dell'espressione su cui opera, come `--x`. Un operatore postfisso viene scritto dopo l'espressione su cui opera, come `x++`.

Gli operatori sono classificati in base a livelli di *precedenza*: operatori con una precedenza più alta si abbinano ai propri operandi più strettamente di quanto facciano operatori con una precedenza più bassa. Ad esempio, * ha una precedenza più alta di +, per cui `x + y * z` ha lo stesso valore di `x + (y * z)`, anche se l'operatore + compare per primo.

La maggior parte degli operatori sono *associativi a sinistra*: operatori aventi la stessa precedenza vengono valutati procedendo da sinistra verso destra. Ad esempio, l'espressione `x - y + z` viene valutata come `(x - y) + z`, e non come `x - (y + z)`. Costituiscono eccezione gli operatori unari prefissi e gli operatori di assegnamento, che sono associativi a destra. Ad esempio, `z = y = Math.sin(x)` ha lo stesso significato di `z = (y = Math.sin(x))`.

In Appendice B trovate un elenco di tutti gli operatori del linguaggio Java.

## Classi

La sintassi per la dichiarazione di una *classe* è

```
[public] [abstract|final] class NomeClasse
      [extends NomeSuperclasse]
      [implements NomeInterfaccia1, NomeInterfaccia2, ...]
{
   caratteristica1
   caratteristica2
   ...
}
```

Ciascuna *caratteristica* è una dichiarazione di questo tipo

```
modificatori costruttore|metodo|variabile|classe
```

oppure un blocco di inizializzazione

```
[static] { corpo }
```

Consultate il paragrafo dedicato ai costruttori per avere maggiori informazioni sui blocchi di inizializzazione.

I possibili *modificatori* sono `public`, `private`, `protected`, `static` e `final`.

La dichiarazione di una *variabile* è

```
Tipo nomeVariabile [= valoreIniziale];
```

Un *costruttore* ha questa forma

```
NomeClasse(parametro1, parametro2, ...)
      [throws TipoEccezione1, TipoEccezione2, ...]
{
```

```
   corpo
}
```

Un *metodo* è così definito

```
Tipo nomeMetodo(parametro1, parametro2, ...)
      [throws TipoEccezione1, TipoEccezione2, ...]
{
   corpo
}
```

mentre un *metodo astratto* è

```
abstract Tipo nomeMetodo(parametro1, parametro2, ...);
```

Ecco un esempio:

```
public class Point
{
   private double x; // variabile di esemplare
   private double y;

   public Point() // costruttore senza argomenti
   {
      x = 0; y = 0;
   }

   public Point(double xx, double yy) // costruttore
   {
      x = xx; y = yy;
   }

   public double getX() // metodo
   {
      return x;
   }

   public double getY()
   {
      return y;
   }
}
```

Una classe può avere sia variabili di esemplare sia variabili `static`. Ogni oggetto della classe ha una propria copia delle variabili di esemplare, mentre esiste un'unica copia delle variabili `static` per tutti gli oggetti della classe.

Una classe che viene dichiarata `abstract` non può essere usata per creare oggetti.

Una classe che viene dichiarata `final` non può essere estesa.

## Interfacce

La sintassi per una *interfaccia* è

```
[public] interface NomeInterfaccia
     [extends NomeInterfaccia1, NomeInterfaccia2, ...]
{
   caratteristica1
   caratteristica2
   ...
}
```

Ciascuna *caratteristica* ha questa forma

```
modificatori metodo|variabile
```

I possibili modificatori sono `default`, `public`, `static` e `final`. I metodi sono automaticamente `public` e le variabili sono automaticamente `public static final`. I metodi predefiniti (`default`) e i metodi statici hanno un corpo.

La dichiarazione di una *variabile* è

```
Tipo nomeVariabile = valoreIniziale;
```

mentre un *metodo* è così definito

```
Tipo nomeMetodo(parametro1, parametro2, ...);
```

Ecco un esempio

```
public interface Measurable
{
   int CM_PER_INCH = 2.54;
   int getMeasure();
   static boolean isSmallerThan(Measurable other)
   {
      return getMeasure() < other.getMeasure();
   }
}
```

## Tipi enumerativi

La sintassi per un tipo enumerativo è

```
[public] enum NomeTipoEnumerativo
{
   costante1, costante2, ...;
   caratteristica1
   caratteristica2
   ...
}
```

Ciascuna *costante* è un nome di costante, seguito da parametri di costruzione opzionali.

```
nomeCostante[(parametro1, parametro2, ...)]
```

Il punto e virgola dopo le costanti è necessario soltanto se il tipo enumerativo definisce ulteriori caratteristiche, che possono essere le stesse caratteristiche attribuibili ad una classe. Ciascuna *caratteristica* è una dichiarazione di questo tipo

```
modificatori metodo|variabile
```

I possibili modificatori sono `public`, `static` e `final`. Tutti i costruttori sono privati.

Ecco due esempi

```
public enum Suit { HEARTS, DIAMONDS, SPADES, CLUBS };

public enum Card
{
   TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6),
         SEVEN(7), EIGHT(8), NINE(9), TEN(10),
         JACK(10), QUEEN(10), KING(10), ACE(11);
   private int value;
   private Card(int aValue) { value = aValue; }
   public int getValue() { return value; }
}
```

## Metodi

La definizione di un *metodo* ha questa forma

```
modificatori Tipo nomeMetodo(parametro1, parametro2, ..., parametroN)
      [throws TipoEccezione1, TipoEccezione2, ...]
{
   corpo
}
```

Il tipo del valore restituito, *Tipo*, può essere qualsiasi tipo del linguaggio Java oppure il tipo speciale `void`, che indica che il metodo non restituisce alcun valore.

Ciascun *parametro* è così definito

```
[final] Tipo nomeParametro
```

Un metodo ha *un numero variabile di parametri* se il suo ultimo parametro ha questa forma speciale:

```
Tipo... nomeParametro
```

Un metodo così definito può essere invocato con una sequenza di parametri di lunghezza qualsiasi contenente valori del tipo specificato: la variabile parametro di cui viene definito il nome è un array di quel tipo, che contiene i valori forniti come parametri. Ad esempio, il metodo

```
public static double sum(double... values)
{
   double s = 0;
   for (double v : values) { s = s + v; }
   return s;
}
```

può essere così invocato:

```
double result = sum(1, -2.5, 3.14);
```

In Java, tutti i parametri vengono passati per *valore*. Ciascun parametro è una variabile locale: il suo ambito si estende fino alla fine del corpo del metodo e viene inizializzata con una copia del valore fornito nell'invocazione. Tale valore può essere di un tipo primitivo o di un tipo riferimento: se è di un tipo riferimento, invocando un metodo modificatore sul riferimento si modificherà l'oggetto il cui riferimento è stato passato al metodo.

Modificando il valore della variabile parametro non si ha alcun effetto al di fuori del metodo; contrassegnando il parametro come `final` si impedisce del tutto tale modifica. Questo viene solitamente fatto per consentire a una classe definita all'interno del metodo di accedere al parametro.

Il linguaggio Java distingue metodi *di esemplare* da metodi *statici*. I metodi di esemplare hanno un parametro speciale, il parametro *implicito*, fornito nell'invocazione del metodo con la sintassi seguente

```
valoreDelParametroImplicito.nomeMetodo(valoreParametro1, valoreParametro2, ...)
```

Esempio:

```
harry.setSalary(30000)
```

Il tipo del parametro implicito deve essere uguale al tipo della classe che contiene la definizione del metodo, mentre un metodo statico non ha parametro implicito.

Nel corpo del metodo, la variabile `this` viene inizializzata con una copia del valore del parametro implicito. Usare il nome di una variabile di esemplare senza qualificarlo significa accedere all'omonima variabile di esemplare del parametro implicito. Ad esempio

```
public void setSalary(double s)
{
   salary = s; // cioè, this.salary = s
}
```

Le invocazioni dei metodi sono, per impostazione predefinita, *risolte dinamicamente*: la macchina virtuale determina la classe a cui appartiene l'oggetto che funge da parametro implicito e invoca il metodo definito in tale classe. Tuttavia, se un metodo viene invocato con la speciale variabile `super`, allora viene invocato il metodo definito nella superclasse, usando `this` come parametro implicito. Ad esempio

```
public class MyPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
```

```
      super.paintComponent(g);
      // invoca JPanel.paintComponent
      ...
   }
   ...
}
```

L'enunciato `return` provoca la terminazione immediata dell'esecuzione di un metodo. Se il tipo del valore restituito dal metodo non è `void`, allora occorre restituire un valore. La sintassi è

```
return [valore];
```

Ad esempio

```
public double getSalary()
{
   return salary;
}
```

Un metodo può invocare se stesso e, in tal caso, viene detto *ricorsivo*:

```
public static int factorial(int n)
{
   if (n <= 1) return 1;
   return n * factorial(n - 1);
}
```

## Costruttori

La definizione di un costruttore ha questa forma

```
modificatori NomeClasse(parametro1, parametro2, ...)
      [throws TipoEccezione1, TipoEccezione2, ...]
{
   corpo
}
```

Per creare e costruire un nuovo oggetto, si invoca un costruttore con un'espressione `new`

```
new NomeClasse(valoreParametro1, valoreParametro2, ...)
```

Un costruttore può eseguire il corpo di un altro costruttore della stessa classe, usando la sintassi

```
this(valoreParametro1, valoreParametro2, ...)
```

Ad esempio

```
public Employee()
{
   this("", 0);
}
```

Un costruttore può anche invocare un costruttore della sua superclasse, usando la sintassi

```
super(valoreParametro1, valoreParametro2, ...)
```

L'invocazione di `this` o `super` deve essere il primo enunciato all'interno del costruttore.

Gli array si costruiscono con questa sintassi

```
new TipoArray [ = {valoreIniziale1, valoreIniziale2, ...}]
```

Ad esempio

```
new int[] = { 1, 4, 9, 16, 25}
```

Quando viene costruito un oggetto, vengono compiute le seguenti azioni:

- Tutte le variabili di esemplare vengono inizializzate a 0, `false` o `null`.
- Le assegnazioni di valori iniziali a variabili di esemplare e i blocchi di inizializzazione vengono eseguiti nell'ordine in cui sono dichiarati.
- Viene invocato il corpo del costruttore.

Quando viene caricata una classe, vengono compiute le seguenti azioni:

- Tutte le variabili statiche vengono inizializzate a 0, `false` o `null`.
- Le assegnazioni di valori iniziali a variabili statiche e i blocchi statici di inizializzazione vengono eseguiti nell'ordine in cui sono dichiarati.

## Enunciati

Un *enunciato* può essere uno dei seguenti:

- un'espressione seguita da un punto e virgola
- un enunciato di diramazione o di ciclo
- un enunciato `return`
- un enunciato `throw`
- un *blocco*, cioè un gruppo di dichiarazioni di variabili e di enunciati racchiusi fra parentesi graffe
- un blocco `try`

Java ha due enunciati di diramazione (`if` e `switch`), tre enunciati di ciclo (`while`, `for` e `do`) e due meccanismi di controllo di flusso non lineari (`break` e `continue`).

L'enunciato `if` ha la forma

```
if (condizione) enunciato1 [else enunciato2]
```

Se la *condizione* è vera, allora viene eseguito *enunciato1*, altrimenti viene eseguito *enunciato2*.

L'enunciato `switch` ha la forma

```
switch (espressione)
{
```

```
   gruppo1
   gruppo2
   ...
   [default:
      enunciato1
      enunciato2
      ...]
}
```

dove ciascun *gruppo* ha la seguente forma

```
case costante1:
case costante2:
...
   enunciato1
   enunciato2
   ...
```

L'*espressione* deve essere di tipo intero, di tipo stringa o di un tipo enumerativo. In relazione al suo valore, il controllo viene trasferito al primo enunciato che segue l'etichetta `case` corrispondente, oppure al primo enunciato che segue l'etichetta `default` se nessuna delle etichette `case` ha il valore richiesto. L'esecuzione prosegue con l'enunciato successivo fino a trovare un enunciato `break` o `return`, oppure viene lanciata un'eccezione, oppure si raggiunge la fine del blocco `switch`; eventuali altre etichette `case` vengono ignorate.

Il ciclo `while` ha la forma seguente

```
while (condizione) enunciato
```

L'*enunciato* viene eseguito finché la *condizione* è vera.

Il ciclo `for` ha la forma seguente

```
for (espressioneDiInizializzazione|dichiarazioneDiVariabile;
      condizione;
      espressioneDiAggiornamento1,
      espressioneDiAggiornamento2, ...)
   enunciato
```

Si esegue una sola volta l'espressione di inizializzazione o la dichiarazione di variabile, poi, finché la *condizione* rimane vera, si eseguono l'*enunciato* del ciclo e, successivamente, le espressioni di aggiornamento. Vediamo alcuni esempi

```
for (i = 0; i < 10; i++)
{
   sum = sum + i;
}
for (int i = 0, j = 9; i < 10; i++, j--)
{
   a[j] = b[i];
}
```

Il ciclo `for` esteso (detto anche "for each") ha la forma seguente

```
for (Tipo variabile : array|oggettoCheImplementaIterable)
   enunciato
```

Quando questo ciclo scandisce un array, è equivalente a

```
for (int i = 0; i < array.length; i++)
{
   Tipo variabile = array[i];
   enunciato
}
```

Altrimenti, l'*oggettoCheImplementaIterable* deve essere un esemplare di una classe che realizza l'interfaccia `Iterable`; in tal caso, il ciclo è equivalente a

```
Iterator i = oggettoCheImplementaIterable.iterator();
while (i.hasNext())
{
   Tipo variabile = i.next();
   enunciato
}
```

Il ciclo `do` ha la forma

```
do enunciato while (condizione);
```

L'*enunciato* viene eseguito ripetutamente finché la *condizione* è vera. Diversamente dal ciclo `while`, l'enunciato di un ciclo `do` viene eseguito almeno una volta.

L'enunciato `break` provoca l'uscita dall'enunciato `while`, `do`, `for` o `switch` più interno che lo racchiude (senza contare gli enunciati `if` o i blocchi di enunciati).

Qualsiasi enunciato (compresi gli enunciati `if` e i blocchi di enunciati) può essere contrassegnato da un'etichetta:

```
etichetta: enunciato
```

L'enunciato break con etichetta

```
break etichetta;
```

provoca la terminazione dell'enunciato etichettato.

L'enunciato `continue` porta l'esecuzione al termine della porzione *enunciato* di un ciclo `while`, `do` o `for`. Nel caso di un ciclo `while` o `do`, viene poi verificata la condizione del ciclo, mentre nel caso di un ciclo `for` vengono eseguite le espressioni di aggiornamento.

L'enunciato `continue` con etichetta

```
continue etichetta;
```

porta l'esecuzione al termine della porzione *enunciato* di un ciclo `while`, `do` o `for` avente l'etichetta corrispondente.

## Eccezioni

L'enunciato `throw`

```
throw espressione;
```

termina bruscamente il metodo in cui si trova e porta il controllo di flusso all'interno della più annidata corrispondente clausola `catch` di un blocco `try` circostante. L'*espressione* deve assumere il valore di un riferimento a un oggetto di una sottoclasse di `Throwable`.

L'enunciato `try` ha la forma seguente

```
try [dichiarazioneDiRisorse] bloccoTry
[catch (TipoEccezione1 variabileEccezione1) bloccoCatch1
 catch (TipoEccezione2 variabileEccezione2) bloccoCatch2
 ...]
[finally bloccoFinally]
```

- Tutti i blocchi sono normali blocchi di enunciati, cioè sequenze di enunciati delimitate da parentesi graffe.
- La dichiarazione di risorse, facoltativa, dichiara e inizializza una o più variabili con esemplari di classi che implementano l'interfaccia `AutoCloseable`.

Vengono eseguiti gli enunciati presenti nel *bloccoTry*: se uno di essi lancia un oggetto di tipo eccezione il cui tipo sia una sottoclasse di uno dei tipi indicati in una clausola `catch`, viene eseguito il *bloccoCatch* corrispondente e, appena si entra in tale blocco, si dice che l'eccezione è stata gestita.

Quando il *bloccoTry* è stato eseguito completamente (perché tutti i suoi enunciati sono stati eseguiti completamente; oppure perché uno dei suoi enunciati era `break`, `continue` o `return`; oppure, ancora, perché è stata lanciata un'eccezione al suo interno), viene eseguito il *bloccoFinally*.

Se si entra nel *bloccoFinally* perché è stata lanciata un'eccezione e in tale blocco viene lanciata una nuova eccezione, quest'ultima maschera l'eccezione precedente.

Se è presente la dichiarazione di risorse, quando il blocco `try` termina in qualunque modo, viene invocato il metodo `close` su tutte le variabili ivi inizializzate.

## Pacchetti

Una classe può essere inserita in un pacchetto, scrivendo nel file sorgente la dichiarazione di pacchetto

```
package nomePacchetto;
```

come prima dichiarazione che non sia una dichiarazione `import`.

Un nome di pacchetto ha la forma

```
identificatore1.identificatore2....
```

Ad esempio

```
java.util
com.horstmann.bigjava
```

Il nome completo ("pienamente qualificato", *fully qualified*) di una classe è

```
nomePacchetto.NomeClasse
```

Ci si può sempre riferire alle classi usando i loro nomi pienamente qualificati, ma è scomodo. Per tale motivo, ci si può riferire alle classi importate scrivendo semplicemente `NomeClasse`. Tutte le classi che si trovano nel pacchetto `java.lang` e nel pacchetto a cui appartiene il file sorgente sono importate automaticamente. Per importare ulteriori classi, si usa una direttiva `import`

```
import nomePacchetto.NomeClasse;
```

oppure

```
import nomePacchetto.*;
```

La seconda versione importa tutte le classi del pacchetto.

## Tipi e metodi generici

Un tipo generico viene dichiarato con uno o più *tipi parametrici*, indicati dopo il nome del tipo

```
modificatori class|interface NomeTipo<tipoParametrico1, tipoParametrico2, ...>
```

Un metodo generico, analogamente, viene dichiarato con uno o più tipi parametrici, indicati *prima* del tipo del valore restituito dal metodo

```
modificatori <tipoParametrico1, ...> tipoRestituito nomeMetodo
```

Ciascun tipo parametrico ha la forma

```
nomeTipoParametrico [extends vincolo1 & vincolo2 & ...]
```

Ad esempio

```
public class BinarySearchTree<T extends Comparable>
public interface Comparator<T>
public <T extends Comparable & Cloneable> T cloneMin(T[] values)
```

I tipi parametrici possono essere usati nella definizione del tipo o del metodo generico come se fossero normali tipi; possono essere sostituiti da qualsiasi tipo effettivo che soddisfi i relativi *vincoli*. Ad esempio, nel tipo `BinarySearchTree<String>`, il tipo effettivo `String` ha sostituito il tipo parametrico `T`.

I tipi parametrici possono anche essere costituiti da *tipi con carattere jolly* ("wild-card types"), che hanno la forma

```
? [super|extends Tipo]
```

Si indica in questo modo un tipo specifico che è sconosciuto nel momento in cui viene definito. Ad esempio, `Comparable<? super Rectangle>` equivale al tipo `Comparable<S>` per uno specifico tipo `S`, che può essere il tipo `Rectangle` oppure un suo supertipo, come `RectangularShape` oppure `Shape`.

## Commenti

Ci sono tre tipi di commenti

```
/* commento */
// commento di una riga
/** commento di documentazione */
```

Il *commento di una riga* si estende fino alla fine della riga; gli altri commenti possono estendersi su più righe, fino al marcatore `*/`.

I commenti di documentazione sono utilizzati dal programma di utilità `javadoc` per generare documentazione in modo automatico.