Lezione 5

Operatori ed espressioni

Caricare i dati in memoria non è l'unica operazione necessaria per trattare le informazioni. Bisogna intervenire sui valori conservati, esaminandoli, combinandoli e rendendoli produttivi. Questo è il compito degli operatori, i più basilari strumenti di manipolazione dei dati presenti in Java. Tramite essi, è possibile intervenire su valori di qualsiasi tipo, svolgendo le operazioni di calcolo necessarie. Questa lezione esamina le nozioni legate agli operatori, passa in rassegna gran parte di essi e conferisce forma al fondamentale concetto di espressione.

Convenzioni per gli operatori
Le convenzioni stabilite da Sun Microsystems suggeriscono l'uso di uno spazio tra gli operatori che lavorano con più valori ed i loro operandi. Ad esempio, è meglio scrivere

```
a + b / c * d
```

piuttosto che

```
a+b/c*d
```

Nel caso degli operatori unari (che lavorano con un solo operando), le convenzioni consigliano l'approccio inverso. Ad esempio, la forma

```
a++
```

è preferibile rispetto a

```
a ++
```

5.1 - Gli operatori

Gli operatori sono sequenze di uno o più caratteri che permettono la manipolazione dei dati. Tutti i linguaggi di programmazione dispongono di una folta schiera di operatori. Nel corso delle lezioni precedenti, si è fatto ricorso a diversi operatori, soprattutto di natura aritmetica, lasciando al lettore il compito di intuirne il significato ed il comportamento:

```
int a = 4;
int b = 5;
int c = a + b;
```

In questo frammento di codice, ad esempio, si utilizzano due differenti tipi di operatori. I caratteri + e =, infatti, sono tra i più noti operatori messi a disposizione dai linguaggi di programmazione *C-like*.

5.2 - Le espressioni

Le espressioni, come già anticipato in precedenza, sono speciali combinazioni di valori ed operatori, che lavorano in concerto per calcolare un risultato:

```
5 + 3
```

Questa espressione è costituita da due dati, ossia i valori letterali *5* e *3*, e dall'operatore +. I dati coinvolti in un'espressione assumono il generico nome di operandi. Gli operandi di un'espressione possono essere specificati tanto letteralmente quanto dinamicamente:

```
5 + a
```

Il primo operando di questa espressione è un valore letterale, mentre il secondo è una variabile. Sono operandi dinamici sia i valori contenuti nelle variabili sia i dati restituiti da una funzione:

```
5 + Math.sqrt(17)
```

Un'espressione si dice *semplice* quando contiene un solo operatore. Per meglio dire, un'espressione semplice esegue un solo calcolo. Al contrario, un'espressione si dice *complessa* quando almeno uno dei suoi operandi è a sua volta un'espressione. Un'espressione complessa, quindi, contiene più di un operatore:

```
5 + 3 + 2
```

Le espressioni complesse, per essere risolte, vengono scomposte in parti semplici. L'operazione appena mostrata, ad esempio, può essere divisa in due frammenti:

```
(5 + 3) + (2)
```

Risolvendo l'espressione semplice che costituisce il primo operando della formulazione originaria, si passa alla forma:

```
8 + 2
```

Da qui, si arriva facilmente al risultato finale:

```
10
```

Come tutti i dati presenti in Java, i risultati elaborati da un'espressione sono associati ad un ben preciso tipo. In questa lezione vengono svolte delle esaurienti considerazioni sulla tipizzazione delle espressioni.

5.3 - Gli operatori aritmetici

Gli operatori aritmetici sono utilizzati per comporre delle espressioni che producono un risultato numerico. Gli operandi di un'espressione aritmetica, di conseguenza, devono essere valori dal significato numerico. In breve, gli operatori aritmetici permettono di combinare ed elaborare i dati di tipo *byte*, *short*, *int*, *long*, *float*, *double* e *char* (anche se si usano poco in corrispondenza di quest'ultimo tipo, il cui significato numerico è secondario). La **Tabella 5.1** riporta gli operatori aritmetici supportati da Java.

Operatore	*Operandi richiesti*	*Operazione aritmetica*
`+`	2	Addizione
`-`	2	Sottrazione
`*`	2	Moltiplicazione
`/`	2	Divisione
`%`	2	Modulo
`++`	1	Incremento
`--`	1	Decremento

Tabella 5.1
Gli operatori aritmetici di Java.

I primi quattro operatori mostrati in tabella simboleggiano le quattro operazioni fondamentali dell'aritmetica. Benché il loro utilizzo sia abbastanza intuitivo, è importante svolgere delle

considerazioni sui tipi degli operandi impiegati e del risultato restituito. Gli operatori di addizione, sottrazione, moltiplicazione e divisione sono *sovraccarichi*. Questo significa che sono implicitamente disponibili in più versioni, che vengono automaticamente selezionate in base al tipo degli operandi specificati. Ne esistono quattro versioni, mostrate in **Tabella 5.2**.

Tipo del primo operando	*Tipo del secondo operando*	*Tipo del risultato*
`int`	`int`	`int`
`long`	`long`	`long`
`float`	`float`	`float`
`double`	`double`	`double`

Tabella 5.2
Le differenti versioni degli operatori aritmetici fondamentali (+, -, * e /).

Nel caso siano sommati due valori *int*, ad esempio, il linguaggio selezionerà automaticamente l'operazione associata a questo dominio, restituendo un risultato a sua volta di tipo *int*. In casi dissimili, la *coercion* gioca un ruolo fondamentale. Apparentemente, infatti, è possibile operare anche con i tipi numerici non riportati in tabella:

```
byte a = 5;
byte b = 2;
System.out.println(a + b);
```

Questo frammento di codice funziona correttamente e non produce alcuna anomalia. Diversamente accade quando si vuole assegnare lo stesso risultato ad una variabile di tipo *byte*:

```
byte a = 5;
byte b = 2;
byte c = a + b;
```

In questo caso, il compilatore restituisce un messaggio di errore già noto:

```
possible loss of precision
found   : int
required: byte
    byte c = a + b;
             ^
```

Poiché non esiste un operatore di somma valido per due addendi di tipo *byte*, il calcolatore si è preso la libertà di convertire implicitamente i due operandi, promuovendoli al tipo *int*. In pratica, è come se si fosse scritto:

```
byte a = 5;
byte b = 2;
int temp1 = a;
int temp2 = b;
byte c = temp1 + temp2;
```

Il tipo di somma applicata, dunque, è quella valida per due addendi *int*. Giacché il risultato restituito da questa variante è a sua volta un dato di tale tipo, non è possibile assegnarlo direttamente ad una variabile *byte*, almeno non senza ricorrere ad un *casting* esplicito:

```
byte a = 5;
byte b = 2;
byte c = (byte) (a + b);
```

Questo frammento, a differenza del precedente, viene accettato dal compilatore. Tuttavia bisogna rendersi conto che, in questa maniera, ci si accolla la responsabilità di una eventuale perdita di precisione. Attenzione, inoltre, nell'eseguire in maniera corretta le operazioni di casting in corrispondenza di un'espressione. La seguente forma, ad esempio, è inesatta:

```
byte c = (byte) a + b;
```

In questo modo, infatti, la conversione è applicata solamente al valore di *a* (che, essendo già un *byte*, non ne ha nemmeno bisogno), prima ancora che venga combinato con il secondo operando. L'impiego di una seconda coppia di parentesi tonde, utilizzate per racchiudere l'intera espressione, fa in modo che il casting si riferisca al risultato ottenuto al termine del calcolo, e non solamente ad uno degli addendi.
Le conversioni implicite sono applicate anche nel caso di operazioni di tipo misto:

```
int a = 6;
long b = 5L;
long c = a + b;
```

In questa situazione, l'intero *a* viene prima promosso al tipo *long* e poi sommato con *b*. Il risultato, di conseguenza, sarà un valore *long*. In generale, il compilatore effettua delle promozioni mirate, in modo da poter applicare la più piccola tra le varianti fruibili. Se in campo ci sono un *long* ed un valore più piccolo, il risultato sarà *long*. Tra un *float* ed un dominio inferiore, vince il tipo *float*. Con almeno un *double* in azione, il risultato sarà certamente di tipo *double*. Non è difficile prevedere quale variante sarà applicata: è sufficiente osservare attentamente il tipo degli operandi impiegati.

Gli operatori + e - dispongono di una variante ad un solo operando, secondo i modelli:

```
+valore
-valore
```

Tali forme sono utilizzate per specificare o variare il segno di un valore numerico, come nel seguente frammento:

```
int a = -5;
int b = +3; // ridondante
int c = -a; // c vale 5
int d = -c; // d vale -3
```

L'operazione *modulo* (operatore *%*) calcola il resto di una divisione intera. In pratica,

l'espressione

```
11 % 3
```

restituisce un valore numerico intero pari a 2, che è il resto della divisione intera

```
11 / 3
```

Per i programmatori C/C++
In Java, contrariamente a quanto avviene in C/C++, l'operazione modulo (operatore *%*) è applicabile anche ai tipi *float* e *double*.

L'operatore *%*, come avviene per le quattro operazioni fondamentali, ha senso solo se applicato tra valori numerici di tipo *int*, *long*, *float* o *double*. Anche in questo caso, entrano in gioco le promozioni automatiche, secondo lo stesso modello descritto in precedenza. Pertanto, è indirettamente possibile applicare l'operazione anche a dati *byte*, *short* e *char*. Il risultato restituito sarà dello stesso tipo degli operandi impiegati, tenendo conto delle eventuali promozioni automatiche:

```
int a1 = 8;
int b1 = 3;
// int % int = int
int c1 = a1 % b1;

long a2 = 8L;
long b2 = 3L;
// long % long = long
long c2 = a2 % b2;

byte a3 = 8;
byte b3 = 3;
// I due byte vengono implicitamente
// promossi al tipo int
int c3 = a3 % b3;

int a4 = 8;
long b4 = 3L;
// L'operando di tipo int viene
// promosso al tipo long
long c4 = a4 % b4;

double a5 = 8D;
long b5 = 3L;
// L'operando di tipo long viene
// promosso al tipo double
double c5 = a5 % b5;
```

Gli operatori di incremento e decremento lavorano con un solo operando numerico, che deve necessariamente essere una variabile. Non possono essere applicati ai letterali o ai risultati calcolati da espressioni e funzioni, indipendentemente dal loro tipo. Questi due speciali operatori, infatti, agiscono sulla variabile cui sono applicati, modificandone il contenuto:

```
int a = 3;
a++; // a adesso vale 4
```

```
int b = 3;
b--; // b adesso vale 3
```

In parole semplici, l'incremento ed il decremento sono scorciatoie per giungere brevemente allo stesso risultato che si otterrebbe con l'impiego di più operatori e più operandi:

```
int a = 3;
a = a + 1; // stesso risultato di a++

int b = 3;
b = b - 1; // stesso risultato di b--
```

L'incremento ed il decremento possono essere applicati in maniera prefissa o suffissa:

```
// Forma prefissa
++a;
--b;

// Forma suffissa
a++;
b--;
```

Apparentemente, sembra non esserci alcuna differenza tra le due formulazioni. Lo dimostra il seguente programma:

```
public class OperatoriAritmeticiUnari1 {

	public static void main(String[] args) {
		int a = 3;
		int b = 3;
		++a;
		b++;
		System.out.println("La variabile a vale " + a);
		System.out.println("La variabile b vale " + b);
	}

}
```

L'output prodotto è mostrato di seguito:

```
La variabile a vale 4
La variabile b vale 4
```

Ad ogni modo, è bene non farsi ingannare dalle apparenze. Benché il risultato sia sempre lo stesso, la differenza tra le due forme viene a galla nel momento in cui il risultato calcolato da uno dei due operatori è assegnato ad una variabile o valutato in un'espressione:

```
public class OperatoriAritmeticiUnari2 {

	public static void main(String[] args) {
		int a = 3;
		int b = 3;
```

```
            int c = ++a;
            int d = b++;
            System.out.println("La variabile a vale " + a);
            System.out.println("La variabile b vale " + b);
            System.out.println("La variabile c vale " + c);
            System.out.println("La variabile d vale " + d);
      }

}
```

L'output prodotto, questa volta, mostra un'apparente incongruenza:

```
La variabile a vale 4
La variabile b vale 4
La variabile c vale 4
La variabile d vale 3
```

Le variabili *a* e *b*, come nel caso precedente, hanno assunto valore 4. La stessa cosa è avvenuta con *c*, che ha ricevuto in fase di inizializzazione il valore della variabile *a*, subito dopo che questa è stata incrementata. La variabile *d*, al contrario, è stata inizializzata con il valore 3. In pratica, ha ricevuto il valore di *b* prima che questo fosse incrementato. Rinunciando all'impiego dell'operatore di incremento, il programma appena visto potrebbe essere tradotto nella seguente forma:

```
public class OperatoriAritmeticiUnari3 {

      public static void main(String[] args) {
            int a = 3;
            int b = 3;

            a = a + 1;
            int c = a;

            int d = b;
            b = b + 1;

            System.out.println("La variabile a vale " + a);
            System.out.println("La variabile b vale " + b);
            System.out.println("La variabile c vale " + c);
            System.out.println("La variabile d vale " + d);
      }

}
```

In generale, quindi, le forme prefisse vengono prima eseguite e poi valutate, mentre quelle suffisse sono prima valutate e poi eseguite. Tenere a mente questa differenza è utile nel momento in cui l'incremento ed il decremento devono essere impiegati all'interno delle istruzioni di controllo, che saranno esaminate nella lezione successiva.

5.4 - Concatenazione delle stringhe

La somma può essere applicata tanto ai valori numerici quanto alle stringhe. In quest'ultimo caso, anziché effettuare un'addizione, l'operatore + esegue una *concatenazione*:

```
"ci" + "ao" = "ciao"
```

La concatenazione è valida se applicata tra due stringhe, oppure tra una stringa ed un valore di tipo predefinito. Il seguente programma dimostra alcune delle concatenazioni lecite:

```
public class Concatenazioni {

	public static void main(String[] args) {
		String s = "ciao";
		int i = 5;
		double d = 1.654;
		char c = '!';
		boolean b = true;
		// stringa + int
		System.out.println(s + i);
		// double + stringa
		System.out.println(d + s);
		// stringa + char
		System.out.println(s + c);
		// boolean + stringa
		System.out.println(b + s);
	}

}
```

L'output del programma è facilmente intuibile:

```
ciao5
1.654ciao
ciao!
trueciao
```

La concatenazione, come risultato, produce sempre un dato di tipo *String*.

5.5 - Gli operatori sui bit

Java dispone di due famiglie di operatori sui bit, direttamente ereditate da C/C++, capaci di lavorare con le rappresentazioni binarie associate a ciascun valore numerico intero. Sono stati concepiti esclusivamente per i tipi *int* e *long*, ma grazie alle conversioni implicite possono indirettamente agire anche con operandi di tipo *byte*, *short* o *char*.
Gli operatori sui bit sono utili quando ci si trova nella necessità di manipolare a basso livello un flusso di dati, ad esempio per l'elaborazione delle immagini o dei campioni sonori, senza poter ricorrere ad astrazioni di più immediato impiego. Per questo motivo, gli operatori sui bit non sono indispensabili per il cammino didattico seguito in questa parte del corso. Tra l'altro, chi non conosce l'aritmetica binaria potrebbe trovarli di difficile comprensione. I lettori meno interessati all'argomento possono saltare a piè pari il paragrafo, oppure possono studiarlo in maniera blanda, per poi tornarci sopra nel momento dell'effettivo bisogno.
La **Tabella 5.3** riporta il primo gruppo dell'insieme, gli operatori logici.

Operatore	*Operandi richiesti*	*Operazione logica*
~	1	NOT

Tabella 5.3
Gli operatori logici di Java.

Operatore	*Operandi richiesti*	*Operazione logica*
`&`	2	AND
`|`	2	OR
`^`	2	XOR

Tilde su Windows e Linux
L'operatore binario NOT è rappresentato con il simbolo ~, chiamato *tilde*. Non è un carattere che si trova sulle tastiere italiane. Pertanto, per digitarlo, è necessario ricorrere ad una particolare combinazione. Gli utenti Windows possono tenere premuto il tasto ALT e digitare, in sequenza, i caratteri "1", "2" e "6". Rilasciando il tasto ALT, il carattere apparirà sullo schermo. Gli utenti Linux possono usare la combinazione ALT GR + "ì" (i accentata).

L'operatore NOT (~) inverte il valore dei bit che costituiscono l'unico operando necessario per il suo impiego:

```
byte a = 55;
System.out.println(~a);
```

L'output di questo frammento di codice è:

```
-56
```

Per comprendere meglio la natura del calcolo effettuato, bisogna partire dalla rappresentazione binaria del decimale 55:

```
55 decimale = 00110111 binario
```

L'operatore NOT inverte il valore di ciascun bit, trasformando tutti gli 0 in 1, e viceversa:

```
valore originale: 00110111
valore invertito: 11001000
```

Per giungere al risultato finale, è sufficiente effettuare il passaggio inverso dalla rappresentazione in base 2 a quella in base 10:

```
11001000 binario = -56 decimale
```

Per semplicità, nel dimostrare il calcolo, sono stati impiegati soltanto 8 bit. Il risultato restituito dall'operatore NOT, in realtà, è di tipo *int* nel caso in cui l'operando impiegato sia di genere *byte*, *short*, *int* o *char*. Se l'operando utilizzato è un valore *long*, anche il risultato elaborato sarà di tale tipo.

Gli operatori AND (&), OR (|) e XOR (^) lavorano comparando i bit dei due operandi necessari per il loro funzionamento:

```
byte a = 25;
byte b = 52;
```

```
System.out.println(a & b); // AND
System.out.println(a | b); // OR
System.out.println(a ^ b); // XOR
```

Il responso mostrato è:

```
16
61
45
```

I tre operatori paragonano uno ad uno i bit degli operandi. Il risultato ottenuto deriva dalle regole riportate nelle seguenti tabelle.

A	*B*	*A & B*
1	1	1
1	0	0
0	1	0
0	0	0

Tabella 5.4
Tattica di lavoro dell'operatore AND (*&*). L'operatore restituisce 1 quando ambo i bit confrontati valgono 1, altrimenti restituisce 0.

A	*B*	*A | B*
1	1	1
1	0	1
0	1	1
0	0	0

Tabella 5.5
Tattica di lavoro dell'operatore OR (|). L'operatore restituisce 1 quando almeno uno dei due bit confrontati vale 1, altrimenti restituisce 0.

A	*B*	*A ^ B*
1	1	0
1	0	1
0	1	1
0	0	0

Tabella 5.6
Tattica di lavoro dell'operatore XOR (^). L'operatore restituisce 1 quando solamente uno dei due bit confrontati vale 1, altrimenti restituisce 0.

Osservando le norme riportate, è ora possibile giustificare i risultati ottenuti con l'ultimo esempio sperimentato. Per prima cosa, è necessario calcolare la rappresentazione binaria dei due valori impiegati per il test:

```
25 decimale = 00011001 binario
52 decimale = 00110100 binario
```

L'operazione AND (*&*) può essere schematizzata al seguente modo:

```
00011001 &
00110100 =
----------
00010000
```

Solo le colonne che contengono due 1 immettono un bit del medesimo valore nel risultato calcolato. A questo punto, bisogna effettuare il passaggio alla rappresentazione decimale:

```
00010000 binario = 16 decimale
```

Il calcolo appena mostrato, come quelli presentati in seguito, è stato volutamente semplificato. Poiché i valori *byte*, prima di essere processati con questo genere di operatori, vengono automaticamente convertiti nelle loro controparti *int*, si sarebbero dovute mettere a confronto ben 32 colonne binarie! Per semplicità di rappresentazione, quindi, la dimostrazione è stata effettuata servendosi dei soli 8 bit effettivamente interessati dall'operazione.

L'operatore OR (|) assegna un 1 quando, nella colonna elaborata, compare almeno un bit con tale valore:

```
00011001 |
00110100 =
----------
00111101 binario = 61 decimale
```

L'operatore XOR (^, detto anche OR esclusivo) assegna un 1 quando solamente uno dei bit presenti in ciascuna colonna vale altrettanto. In corrispondenza di due 1 o di due 0, l'operatore XOR assegna uno 0:

```
00011001 ^
00110100 =
----------
00101101 binario = 45 decimale
```

All'interno dei tre calcoli effettuati, sono stati impiegati due valori numerici interi dello stesso tipo (*byte*). Gli operatori logici, ad ogni modo, possono essere utilizzati per combinare degli operandi di tipo tra loro distinto, purché appartengano sempre alla categoria dei numerici interi. Questo, oramai dovrebbe essere facile intuirlo, è possibile grazie alle conversioni implicite usate automaticamente da Java. Gli operatori sui bit, come già detto, vengono applicati esclusivamente su operandi *int* o *long*. Quando, in un'espressione logica, si utilizzato dei valori non direttamente comparabili (come i *byte* citati negli esempi precedenti), questi vengono automaticamente convertiti in *int* o in *long*, a seconda delle necessità. Il tipo del risultato restituito è un *int* nel caso in cui nessuno dei due operandi sia di tipo *long*, altrimenti anche il risultato sarà un valore *long*. Riassumendo, il codice

```
byte a = 5;
int b = 3;
int c = a & b;
```

è implicitamente equivalente al frammento

```
byte a = 5;
int b = 3;
int temp = a;
int c = temp & b;
```

La seconda famiglia di operatori sui bit disponibile in Java è costituita dagli operatori di scorrimento, elencati in **Tabella 5.7**.

Operatore	*Operandi richiesti*	*Operazione*
<<	2	Scorrimento a sinistra
>>	2	Scorrimento a destra
>>>	2	Scorrimento a destra senza segno

Tabella 5.7
Gli operatori di scorrimento di Java.

Gli operatori di scorrimento si applicano a valori *int* e *long* e restituiscono un risultato dello stesso tipo degli operandi sfruttati. Anche in questo caso, le conversioni implicite sono automaticamente impiegate per l'adattamento dei dati e l'estensione delle funzionalità ai domini *byte*, *short* e *char*. Gli operatori di scorrimento si servono di due operandi:

```
valore operatore_di_scorrimento numero_cifre
```

Il secondo operando, etichettato *numero_cifre*, indica la larghezza della traslazione che deve essere effettuata. Benché questo valore possa essere un intero qualsiasi, l'operazione ha senso solo per valori compresi tra 1 e 31, nel caso di operandi *int*, o tra 1 e 63, per i *long*.

Lo scorrimento a sinistra consiste in una traslazione verso tale direzione dei bit che costituiscono la rappresentazione binaria di un intero. Ad esempio:

```
byte a = 4;
System.out.println(a << 3);
```

L'output prodotto da questo frammento è:

```
32
```

Il calcolo è riproducibile osservando alcuni passi. Per prima cosa, bisogna ricavare la rappresentazione binaria del valore 4:

```
4 decimale = 00000100 binario
```

A questo punto, l'operazione è effettuata traslando verso sinistra tutti i bit che compongono la rappresentazione. Vanno irrimediabilmente persi i bit che, per effetto dello spostamento, escono dal margine sinistro. Le posizioni vuote che si formano sulla destra vengono riempite con degli 0:

```
valore originale: 00000100
risultato:        00100000 binario = 32 decimale
```

Lo scorrimento a destra funziona in maniera parzialmente inversa. L'affermazione può essere dimostrata invertendo l'ultimo esempio visto:

```
byte a = 32;
System.out.println(a >> 3);
```

L'output mostrato è:

```
4
```

L'operazione di scorrimento a destra, rispetto al precedente caso, impiega una regola differente per il riempimento dei bit rimasti vuoti a seguito della traslazione. Mentre nel caso dello spostamento a sinistra le posizioni scoperte vengono completate con degli 0, lo spostamento a destra riempie i vuoti con delle copie del precedente contenuto del primo bit più a sinistra. Lo scopo di questo accorgimento è il mantenimento del segno. Gli *int* e i *long* di Java, infatti, rappresentano le cifre negative usufruendo della notazione binaria comunemente nota come complemento a 2. Secondo questa notazione, che può essere approfondita servendosi di testi più pertinenti, il primo bit a sinistra di ogni valore è impiegato per la rappresentazione del segno. Quando vale 0, il numero è positivo. Se è 1, il numero è negativo, ed il suo valore assoluto è mantenuto nei rimanenti bit in forma invertita (gli 1 sono cambiati in 0, e viceversa) ed incrementato di una unità. Questa speciale notazione rende molto celeri le operazioni fondamentali come l'addizione e la sottrazione. In pratica, semplificando ad 8 bit la rappresentazione degli interi:

```
13 decimale = 00001101 binario
```

Per ottenere la rappresentazione di -13 bisogna anzitutto invertire tutti i bit di 13:

```
11110010
```

Per concludere, si deve incrementare di una unità il valore ottenuto:

```
11110010 +
       1 =
----------
11110011 binario = -13 decimale
```

Effettuando su -13 uno spostamento verso destra di tre posizioni, si ottiene:

```
11110011 >> 3 =
11111110
```

I primi tre bit a sinistra sono stati riempiti con degli 1, giacché tale è il valore che simboleggia il segno dell'operando originario. Il complemento a 2 è una tecnica ciclica, che può essere nuovamente utilizzata per passare da un numero negativo al suo opposto positivo. Applicandola al risultato ottenuto, quindi, sarà possibile convertire in decimale il risultato binario appena ottenuto.

Prendo il valore di partenza:

```
  11111110
```

Inverto i bit e mi ricordo del segno:

```
 -00000001
```

Sommo 1:

```
 -00000010 binario = -2 decimale
```

E' possibile riscontrare la bontà del calcolo con una sola istruzione Java:

```
System.out.println(-13 >> 3);
```

Lo scorrimento a destra senza segno, al contrario dello scorrimento a destra di tipo semplice e analogamente allo scorrimento a sinistra, completa con degli 0 gli spazi lasciati vuoti dalla traslazione dei bit. In questa maniera, il segno dell'operando originale non sempre può essere mantenuto. L'operazione è utile quando le rappresentazioni binarie dei valori numerici vengono in qualche modo riadattate per rappresentare degli interi senza segno. Si tratta di una tattica frequentemente usata nella codifica binaria delle immagini, dove le componenti di colore e trasparenza vengono espresse tramite degli indici il cui limite inferiore è lo zero.

Trucchi binari
Le operazioni binarie permettono dei simpatici trucchi, capaci di migliorare lievemente le prestazioni di un programma. Uno dei più celebri si basa sull'operatore XOR, e permette lo scambio del contenuto di due variabili intere senza ricorrere all'ausilio di una terza variabile temporanea:

```
a = a ^ b;
b = b ^ a;
a = a ^ b;
```

Al termine dell'esecuzione del frammento, i valori contenuti in *a* e *b* saranno scambiati.

Un secondo trucco riguarda invece gli operatori di scorrimento, e permette la moltiplicazione e la divisione per 2 con migliori prestazioni rispetto i normali operatori aritmetici tra interi:

```
a = a << 1; // è come a * 2
a = a >> 1; // è come a / 2
```

Comunque sia, questi stratagemmi sono riportati solo in veste di curiosità. Non è molto conveniente adottarli, perché inficiano la leggibilità del codice. D'altro canto, i miglioramenti che garantiscono in fatto di prestazioni sono veramente minimi, e Java non è certo un linguaggio che viene scelto in situazioni in cui le prestazioni sono un fattore determinante. Non a caso, questi trucchi sono molto più usati dai programmatori C.

5.6 - Gli operatori di confronto

Gli *operatori di confronto*, detti anche *operatori di relazione*, paragonano i due operandi cui sono applicati, restituendo un risultato di natura booleana (vale a dire un valore di tipo

boolean). In **Tabella 5.8** sono riassunti gli operatori di confronto supportati da Java.

Operatore	*Operandi richiesti*	*Tipo di paragone*
==	2	Uguale
!=	2	Diverso
>	2	Maggiore
<	2	Minore
>=	2	Maggiore o uguale
<=	2	Minore o uguale

Tabella 5.8
Gli operatori di confronto di Java.

Gli operatori **==** e *!=* possono essere applicati a qualsiasi tipo di dato, purché la comparazione abbia un senso logico (ad esempio, è inutile confrontare un valore booleano con un qualsiasi dato numerico). Il loro scopo è verificare l'uguaglianza dei due operandi paragonati, restituendo *true* o *false* secondo l'esito dell'operazione:

```
int a = 5;
int b = 3;
System.out.println(a == b); // Scrive false, perché sono diversi
System.out.println(a != b); // Scrive true, perché non sono uguali
```

I rimanenti operatori di confronto possono essere applicati solo a quei tipi di dati per i quali vige una regola di ordinamento predefinita, ossia ai domini *byte*, *short*, *int*, *long*, *float*, *double* e *char*:

```
int a = 5;
byte b = 5;
double c = 12.54;
System.out.println(a > b);  // false
System.out.println(a >= b); // true
System.out.println(a < c);  // true
System.out.println(b <= c); // true
System.out.println(a > c);  // false
```

I caratteri, secondo lo standard Unicode, sono associati a degli indici numerici, la cui successione rende valide le norme di ordinamento alfabetico e lessicografico:

```
char a = 'a';
char b = 'b';
char c = 'c';
System.out.println(a > b); // false
System.out.println(a < b); // true
System.out.println(b < c); // true
```

Le espressioni costruite intorno agli operatori di confronto sono comunemente chiamate *espressioni booleane*, per via del tipo del valore restituito. Le espressioni booleane sono ampiamente utilizzate per controllare il flusso di esecuzione del codice, come si vedrà nella lezione successiva.

5.7 - Gli operatori logici booleani

Gli operatori logici booleani lavorano su degli operandi *boolean* e producono un risultato del medesimo tipo. Per questo motivo, sono spesso utilizzati insieme agli operatori di confronto per la costruzione di espressioni booleane complesse. Gli operatori logici booleani supportati da Java sono elencati in **Tabella 5.9**.

Operatore	*Operandi richiesti*	*Operazione logica*
`!`	1	NOT booleano
`&`	2	AND booleano
`|`	2	OR booleano
`^`	2	XOR booleano
`&&`	2	AND booleano short-circuit
`||`	2	OR booleano short-circuit

Tabella 5.9
Gli operatori logici booleani di Java.

Come è possibile notare, gli operatori *&* (AND), | (OR) e ^ (XOR) sono i medesimi che si impiegano per le comparazioni bit a bit. In questo caso, però, il loro impiego è molto più immediato e semplice.

L'operatore NOT (*!*) inverte la condizione booleana espressa dal proprio operando:

```
boolean a = true;
boolean b = !a; // b è false
```

L'operatore AND (*&* e *&&*) combina due operandi booleani, restituendo *true* quando entrambi sono veri:

```
boolean a = true;
boolean b = false;
boolean c = a & b; // c è false
boolean d = a & true; // d è true
```

L'operatore OR (| e ||) restituisce *true* quando almeno uno dei suoi operandi è vero:

```
boolean a = true;
boolean b = false;
boolean c = a | b; // c è true
boolean d = b | false; // d è false
```

L'operatore XOR (^, detto anche OR esclusivo) restituisce *true* quando soltanto uno dei suoi operandi è vero:

```
boolean a = true;
boolean b = false;
```

```
boolean c = a ^ b; // c è true
boolean d = a ^ true; // d è false
```

Le seguenti tabelle riportano degli specchietti riassuntivi sulle funzionalità degli ultimi tre operatori descritti.

A	*B*	*A & B* *A && B*
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*false*

Tabella 5.10
Tattica di lavoro dell'operatore AND (*&*), valida anche per l'operatore AND short-circuit (*&&*). L'operatore restituisce *true* quando ambo gli operandi sono veri, altrimenti restituisce *false*.

A	*B*	*A | B* *A || B*
true	*true*	*true*
true	*false*	*true*
false	*true*	*true*
false	*false*	*false*

Tabella 5.11
Tattica di lavoro dell'operatore OR (|), valida anche per l'operatore OR short-circuit (||). L'operatore restituisce *true* quando almeno uno dei due operandi è vero, altrimenti restituisce *false*.

A	*B*	*A ^ B*
true	*true*	*false*
true	*false*	*true*
false	*true*	*true*
false	*false*	*false*

Tabella 5.12
Tattica di lavoro dell'operatore XOR (^). L'operatore restituisce *true* quando solamente uno dei due operandi è vero, altrimenti restituisce *false*.

Le varianti *short-circuit* (letteralmente, *corto-circuito*) disponibili per gli operatori AND e OR sono spesso più utilizzate delle loro controparti originarie. La motivazione è presto spiegata. Si consideri il seguente frammento di codice:

```
boolean a = false;
boolean b = true;
boolean c = a & b; // AND semplice
```

Nel momento in cui la terza riga viene eseguita, la macchina virtuale di Java esamina prima il contenuto di *a* e poi quello di *b*, per trarre il giusto risultato. In questo particolare caso, però, sarebbe possibile fornire una risposta dopo la sola analisi del valore conservato in *a*. Giacché questo è *false*, tenendo a mente le regole di funzionamento dell'operatore AND, è già possibile decretare il risultato, che sarà ovviamente *false*. Tutto ciò avviene

indipendentemente dal valore di *b*. Mentre la formulazione classica dell'operatore AND esegue comunque ambo le valutazioni, la variante short-circuit osserva il ragionamento appena presentato:

```
boolean a = false;
boolean b = true;
boolean c = a && b; // AND short-circuit
```

Il primo valore esaminato è falso? Bene, allora a cosa serve esaminare anche il secondo? Il risultato può già essere decretato una volta per tutte: *false*! Un'operazione di troppo è stata così evitata. Le scorciatoie short-circuit favoriscono la robustezza del codice in diverse situazioni. La lezione successiva presenterà un esempio in merito.
Discorso analogo con la variante short-circuit dell'operatore OR. In questo caso, però, la seconda valutazione sarà saltata soltanto quando il primo operando è vero. Solo in tale situazione, infatti, è possibile ottenere immediatamente un risultato certo.

5.8 - Gli operatori di assegnazione

L'assegnazione è un'operazione necessaria per memorizzare un valore all'interno di una variabile. La forma generica di un'assegnazione è la seguente:

```
variabile operatore_di_assegnazione valore
```

Ad esempio:

```
a = 5
```

Il principale operatore di assegnazione è l'uguale (=), che è stato sinora impiegato diverse volte. La peculiarità di questo operatore è che può essere utilizzato anche per le inizializzazioni, e non solo per le assegnazioni generiche. Tale caratteristica non è condivisa dagli altri operatori di assegnazione, che possono lavorare esclusivamente su variabili già inizializzate.
L'assegnazione, oltre a causare la variazione del valore conservato in una variabile, costituisce anche un'espressione, poiché restituisce un valore. Lo dimostra il seguente frammento:

```
int a;
System.out.println(a = 5);
```

Il valore *5* non solo è stato memorizzato all'interno della variabile *a*, ma è anche stato propagato all'indietro, vero la porzione di codice che precede l'operazione. Questo consente, ad esempio, di effettuare delle assegnazioni a catena:

```
int a, b, c, d;
a = b = c = d = 5;
```

L'impiego dell'assegnazione all'interno di un'espressione complessa, ad ogni modo, è un'operazione che in rare occasioni torna effettivamente utile. Il più delle volte, infatti, non fa altro che diminuire la leggibilità del codice.

I rimanenti operatori di assegnazione messi a disposizione da Java sono delle scorciatoie per abbinare l'assegnazione ad una delle operazioni a due operandi attuabili tramite le altre categorie di operatori già esaminati:

```
int a = 5;
int a += 2; // è come a = a + 2
```

Gli operatori di assegnazione di questo tipo sono riportati in Tabella 5.13.

Operatore	*Corrispondenza*
`+=`	`+` (sia sui numerici sia sulle stringhe)
`-=`	`-`
`*=`	`*`
`/=`	`/`
`%=`	`%`
`&=`	`&` (sia binario sia booleano)
`|=`	`|` (sia binario sia booleano)
`^=`	`^` (sia binario sia booleano)
`<<=`	`<<`
`>>=`	`>>`
`>>>=`	`>>>`

Tabella 5.13
Operatori di assegnazione con calcolo di Java.

Ecco, infine, un frammento dimostrativo:

```
int a = 3;
a += 2; // a vale ora 5
a -= 3; // a vale ora 2
a *= 4; // a vale ora 8
a /= 2; // a vale ora 4
a %= 3; // a vale ora 1

boolean a = true;
a &= false; // a vale ora false
a |= true;  // a vale ora true
a ^= true;  // a vale ora false
```

5.9 - L'operatore di valutazione condizionata

Java dispone inoltre di uno speciale operatore ternario, capace di lavorare con ben tre operandi. Si tratta dell'operatore di valutazione condizionata:

```
valore_booleano ? valore_1 : valore_2
```

Se *valore_booleano* è vero, allora viene valutato *valore_1*, altrimenti è valutato *valore_2*. In termini più pratici:

```
boolean a = true;
int b = a ? 3 : 5;
```

La variabile *b*, al termine dell'operazione, contiene il valore 3. Questo perché la condizione impiegata è vera. Il seguente frammento effettua la valutazione inversa, assegnando a *b* il valore 5:

```
boolean a = false;
int b = a ? 3 : 5;
```

Ci sono casi in cui la valutazione condizionata è molto utile. Si prenda in considerazione la seguente istruzione, supponendo che a sia una variabile numerica:

```
int b = (a == 0) ? 0 : (5 / a);
```

Se si fosse scritto direttamente

```
b = 5 / a;
```

si sarebbe incappati nel rischio di una divisione per zero, operazione illecita che causa l'arresto del programma a seguito della propagazione di un errore di runtime. Basta fare una prova per verificarlo:

```
int a = 0;
b = 5 / a;
```

Il programma viene correttamente compilato. Da un punto di vista sintattico, infatti, non contiene alcuna imprecisione. Tuttavia, tentando l'esecuzione, si incappa nel problema sopra descritto, rappresentato dal messaggio:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Al contrario, usando la valutazione condizionata, tutto procede per il meglio. Nessuna divisione per zero è mai tentata. Tutto ciò rende il codice robusto e solleva lo sviluppatore dal compito di dover scrivere delle più artefatte procedure di controllo. Quello presentato, ad ogni modo, non è l'unico caso in cui la valutazione condizionata costituisce un passaggio conveniente. Più in generale, la tecnica è utile ogni volta che una valutazione dipende dall'esito di un confronto booleano.

5.10 - Altri operatori

La panoramica presentata in questa lezione non esaurisce le operazioni ammissibili in Java. Alcune procedure di calcolo, con i corrispondenti operatori, sono descritte altrove. Le operazioni di casting, ad esempio, sono già state esaminate nella lezione precedente. Gli operatori indispensabili per l'uso degli oggetti e degli array saranno invece descritti più avanti. In generale, mancano all'appello cinque operatori: *new*, *instanceof*, il punto, le parentesi tonde (che hanno triplice valenza) e le parentesi quadre.

5.11 - Regole di precedenza tra gli operatori

Le espressioni complesse non sempre sono risolte linearmente, da sinistra verso destra. L'ordine di valutazione, il più delle volte, dipende dalla priorità associata ai singoli operatori impiegati. Si consideri la seguente espressione:

```
5 + 2 * 3
```

L'ordine di esecuzione di questo calcolo non è lineare, giacché il prodotto è eseguito prima della somma:

```
5 + (2 * 3)
5 + 6
11
```

A ciascun operatore è associato un grado di priorità. I calcoli con priorità maggiore sono eseguiti prima degli altri. La **Tabella 5.14** schematizza l'ordine di precedenza degli operatori di Java.

[]	.	(parametri)		
var++	var--	++var	--var	
+val	-val			
~	!			
new	(casting)			
*	/	%		
+	-			
<<	>>	>>>		
<	>	<=	>=	instanceof
&	^	|		
&&	||			
? :				
=				
+=	-=			
*=	/=	%=		
&=	^=	|=		
<<=	>>=	>>>=		

Tabella 5.14
Regole di precedenza tra gli operatori di Java. Gli operatori riportati nelle righe più alte hanno priorità maggiore rispetto quelli presentati più in basso. All'interno di ciascuna riga, la priorità è maggiore verso sinistra e minore verso destra.

5.12 - Impiego delle parentesi tonde

Le parentesi tonde, in Java, ricoprono tre diversi significati, secondo il contesto in cui sono applicate. Nella precedente lezione, è stato discusso il loro ruolo nelle operazioni di casting:

```
(nuovo_tipo) valore
```

Benché l'argomento non sia ancora stato trattato, alcuni esempi hanno già dimostrato come le parentesi tonde possano essere impiegate per fornire dei valori ad una specifica funzionalità:

```
Math.sqrt(17)
```

Il terzo possibile utilizzo delle parentesi tonde riguarda le espressioni complesse. Proprio come avviene in aritmetica, le parentesi possono essere utilizzate per variare le precedenze altrimenti in vigore:

```
(5 + 2) * 3
7 * 3
21
```

Nell'espressione appena riportata, la somma assume priorità maggiore rispetto al prodotto. Il normale ordine di valutazione, dunque, è stato invertito. Più coppie di parentesi tonde possono essere annidate le une dentro le altre:

```
((5 + 7) / (5 - 3)) * 4
(12 / 2) * 4
6 * 4
24
```

Le parentesi, infine, possono essere impiegate per migliorare la leggibilità del codice e per facilitare la comprensione di un'espressione. Nel seguente calcolo, ad esempio, l'uso delle parentesi è ridondante, poiché ripropone le normali priorità già stabilite dal linguaggio:

```
((a * b) > (a / d)) && ((a + b) != (c - d))
```

La ridondanza, ad ogni modo, è utile, giacché semplifica la comprensione di un'espressione altrimenti criptica:

```
a * b > a / d && a + b != c - d
```

L'uso di parentesi ridondanti non influisce negativamente sulle prestazioni del codice. Di conseguenza, non c'è alcuna controindicazione al loro utilizzo.