Lezione 4

Variabili e tipi di dati

Tutti i programmi, senza alcuna eccezione, manipolano dei dati. Un programma di videoscrittura, ad esempio, deve tener traccia delle digitazioni dell'utente, un videogioco deve ricordare le coordinate di ciascun personaggio mostrato sullo schermo, un applicativo di rete deve scambiare informazioni con una macchina remota, un software gestionale deve manipolare i dati depositati in un archivio persistente e via discorrendo. Per questo motivo, tutti i linguaggi di programmazione offrono degli strumenti utili per la memorizzazione, la gestione e la manipolazione dei dati. La soluzione escogitata dai creatori di Java, in gran parte derivata dall'esperienza di C/C++, offre un modello ordinato, sicuro e robusto, basato su una rigida tipizzazione dei dati. Gli aspetti cardine di tale modello sono esaminati in questa lezione, che tratta le variabili, le costanti, gli ambiti di validità, i tipi predefiniti e le conversioni.

4.1 - Le variabili

All'interno di un programma, le variabili costituiscono l'elemento base per la memorizzazione dei dati ed il mantenimento delle informazioni. In Java, le variabili si definiscono combinando un tipo, un identificatore ed opzionalmente un parametro di inizializzazione:

```
tipo identificatore [ = inizializzazione];
```

Ad esempio:

```
int a = 5;
```

Non è necessario assegnare un valore alla variabile nel momento in cui la si dichiara:

```
int a;
```

Il motto delle variabili

Il motto delle variabili di Java è "prima dichiari, poi inizializzi, quindi utilizzi". Nessuna variabile può sfuggire a questa regola certosina.
Dichiarazione ed inizializzazione, dunque, sono operazioni che possono essere svolte in momenti distinti, purché la prima preceda sempre la seconda:

```
int a;
...
a = 5;
```

Java, a differenza di C/C++, richiede che una variabile venga sempre inizializzata prima di poter essere utilizzata. Il seguente codice produce un errore di compilazione:

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

Il testo dell'errore è:

```
variable a might not have been initialized
    System.out.println(a);
                       ^
```

Più variabili dello stesso tipo possono essere dichiarate nel corso di una medesima istruzione, utilizzando la virgola come separatore:

```
int a, b, c;
```

Allo stesso modo, è possibile dichiarare e contemporaneamente inizializzare più variabili dello stesso tipo:

```
int a = 1, b = 2, c = 3;
```

Sono valide anche le forme miste:

```
int a = 1, b, c = 3;
```

In questo caso, solo *a* e *c* sono state inizializzate. La variabile *b*, invece, dovrà ricevere un valore in seguito, prima di poter esser utilizzata. Le convenzioni suggerite da Sun, ad ogni modo, consigliano di utilizzare istruzioni distinte per la dichiarazione ed eventualmente l'inizializzazione di ciascuna variabile. Questo approccio rende più prolisso il codice, ma ne migliora la leggibilità.
Il valore assegnato inizialmente ad una variabile può essere determinato staticamente, attraverso l'uso di una costante letterale, oppure dinamicamente, mediante un'espressione o una funzione. Le costanti letterali sono dei valori digitati direttamente dal programmatore all'interno del codice, che non dipendono da situazioni dinamiche quali l'acquisizione di dati dall'esterno o il verificarsi di un particolare evento. Esempi di costanti letterali sono *5*, *'a'*, *"ciao"* e *2.12*. L'inizializzazione letterale di una variabile è la più semplice forma di attribuzione di un valore:

```
// Inizializzazione letterale
int a = 5;
```

Nel caso dell'inizializzazione dinamica, invece, il valore viene determinato mentre il programma è in esecuzione, poiché derivato da eventi e dati non sempre noti a priori:

```
// Inizializzazioni dinamiche
int b = a; /* A priori, non si sa quanto vale a, il cui
              valore, magari, è stato impostato dall'utente
              oppure letto da un file. */
int c = a * 2; // a moltiplicato 2 (espressione)
double d = Math.sqrt(a); // radice di a (funzione)
```

4.2 - Le costanti

Le costanti sono speciali variabili il cui contenuto, una volta assegnato, non può più essere modificato. Sono utili in molti casi, soprattutto quando si programma pensando per oggetti. Una variabile diviene costante quando, alla sua dichiarazione, è anteposta la parola *final*:

```
final int NOME_COSTANTE = 5;
```

Un codice come il seguente è scorretto, poiché il valore mantenuto da una costante non può mai essere modificato:

```
final int A = 5;
A = 3; // Errore! A è final!
```

Ecco l'errore che si riceve tentando la compilazione di una classe che includa il frammento:

```
cannot assign a value to final variable A
    A = 3;
    ^
```

Resta comunque possibile disaccoppiare le istruzioni di dichiarazione ed inizializzazione di una costante:

```
final int A;
...
A = 5;
```

In realtà, le variabili *final* non sono delle costanti nel senso più stretto del termine, giacché possono essere inizializzate dinamicamente e non solo letteralmente. Sarebbe più corretto etichettarle come *variabili di sola lettura*. Attualmente, ad ogni modo, Java non dispone di altri meccanismi utili per creare vere e proprie costanti. Per questo motivo, tenendo sempre a mente l'ultima osservazione riportata, è possibile mischiare i concetti.

4.3 - Gli ambiti di validità delle variabili

Le variabili, così come le costanti, non sono eterne, ma possiedono un proprio ciclo vitale. Nascono nel momento in cui vengono dichiarate, divengono pronte all'uso immediatamente dopo l'inizializzazione e muoiono non appena escono dal loro *ambito di validità* (detto altrimenti *ambito di visibilità*). Gli ambiti di validità, a livello basilare, sono strettamente collegati all'impiego dei blocchi di istruzioni, descritti nella lezione precedente.

Si prenda in considerazione il seguente programma:

```
public class EsempioAmbiti1 {

	public static void main(String[] args) {
		int a = 5;
		int b = 3;
		int c = a + b;
		System.out.println(c);
	}

}
```

Le tre variabili *a*, *b* e *c* appartengono al blocco denominato *main*, poiché sono state dichiarate al suo interno. Fin quando il flusso di esecuzione percorre le istruzioni racchiuse in questa frazione del programma, i riferimenti *a*, *b* e *c* possono essere liberamente impiegati.

Come è stato rimarcato nel corso della lezione precedente, è possibile annidare più blocchi l'uno dentro l'altro:

```
public class EsempioAmbiti2 {

	public static void main(String[] args) {
		int a = 5;
		int b = 3;
		{
			int c = a + b;
			System.out.println(c);
		}
	}

}
```

In questo nuovo esempio, si è fatto ricorso ad un secondo blocco, contenuto all'interno di *main*. Compilando ed eseguendo il codice, non si riscontrerà alcuna differenza rispetto al

primo programma visto. L'equivalenza, tuttavia, è solo apparente. La variabile *c*, nel caso più recente, è stata dichiarata in un blocco annidato all'interno di *main*. Questo comporta che tale riferimento non potrà essere impiegato al suo esterno. E' possibile verificarlo con una terza variante del programma, che il compilatore si rifiuterà di processare:

```
public class EsempioAmbiti3 {

      public static void main(String[] args) {
            int a = 5;
            int b = 3;
            {
                  int c = a + b;
            }
            System.out.println(c); /* Errore! Questo non è l'ambito di
                                      validità della variabile c! */
      }

}
```

Una volta completata l'elaborazione del blocco annidato, il riferimento *c* esce dal proprio ambito di validità. In poche parole, la variabile *c* muore al termine dell'esecuzione del blocco che la contiene. Per questo motivo, non può più essere citata nelle successive porzioni di codice, giacché sconosciuta in tale sede. L'errore di compilazione che si riceve è il seguente:

```
cannot resolve symbol
symbol  : variable c
location: class EsempioAmbiti3
    System.out.println(c);
                       ^
```

Riassumendo, una variabile può essere impiegata esclusivamente nel suo blocco di appartenenza ed in tutti quelli annidati all'interno di quest'ultimo.

4.4 - Gli identificatori

Ad ogni variabile corrisponde un identificatore, vale a dire un nome, che può essere impiegato per richiamare ed utilizzare la variabile dopo che questa è stata dichiarata ed inizializzata. Gli identificatori, per essere reputati validi, devono rispettare alcune regole. Per prima cosa si ricorda che Java è un linguaggio *case-sensitive*: *nomeVariabile*, *NomeVariabile*, *NOMEVARIABILE*, *NomeVARIABILE* e tutte le possibili varianti sono nomi che identificano elementi distinti.
Gli identificatori possono essere costituiti da una sequenza qualsiasi di lettere minuscole o maiuscole, numeri, trattini di sottolineatura e simboli del dollaro. Non possono, però, cominciare con un numero. I seguenti identificatori sono validi:

```
cont   Test   VariabileProva   $aux   c3   a_prima
```

I prossimi, al contrario, sono incorretti:

```
2aux   variabile-prova   %perc
```

La sintassi di Java, inoltre, riserva 52 parole (**Tabella 4.1**), che non possono essere impiegate come identificatori.

abstract	double	int	strictfp
assert	else	interface	super
boolean	extends	long	switch
break	false	native	synchronized
byte	final	new	this
case	finally	null	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	true
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

Tabella 4.1
Le 52 parole riservate di Java 2 v1.4.

const e goto
Le parole chiave *const* e *goto* sono riservate ma attualmente non utilizzate da Java.

Gli identificatori, infine, non possono essere impiegati più di una volta all'interno del medesimo ambito di visibilità, salvo in alcuni casi dove l'ambiguità è risolta da strutture sovrastanti (questi concetti saranno meglio chiariti in seguito). Due variabili che condividono lo stesso ambito, quindi, non possono assumere il medesimo nome, giacché altrimenti verrebbero confuse.

Le convenzioni stabilite da Sun Microsystems invitano gli sviluppatori ad utilizzare le lettere minuscole per i nomi delle variabili. Quando il nome di una variabile è costituito dall'unione di più parole, si suggerisce l'uso di un carattere maiuscolo per la prima lettera di ciascuna parola successiva alla prima:

```
nomeUtente    passwordDiAccesso    dataDelMioCompleanno
```

Per le costanti, invece, si raccomanda l'impiego delle sole lettere maiuscole, ricorrendo ai trattini di sottolineatura per la separazione delle singole parole che compongono l'identificatore:

```
COST    ORA_LEGALE    COEFF_ATTRITO_DINAMICO
```

Non solo le variabili si servono di identificatori. Anche i metodi, le classi, le interfacce ed i pacchetti hanno nomi che lo sviluppatore deve necessariamente definire. In generale, in questi casi, valgono in parte le medesime norme illustrate in merito alle variabili, anche se le convenzioni variano da un caso all'altro. L'argomento sarà approfondito nelle lezioni di specifico interesse.

4.5 - I tipi di dati

Java è un linguaggio tipizzato. Ad ogni tipo corrispondono un insieme di valori possibili ed una serie di funzionalità utili per manipolare gli elementi di tale insieme. I valori numerici dispongono di funzioni utili per il calcolo matematico, le stringhe godono di utilità per la manipolazione del testo, le date possono sfruttare delle procedure pensate appositamente per gli intervalli temporali e così via. Tutto ciò implica una netta ed esplicita divisione tra i valori di un tipo e quelli di un altro, che mai possono essere confusi o impiegati in maniera promiscua. I primi dati soggetti a questa norma sono le costanti letterali. *5* è un intero di

tipo *int*, *5.12* un numerico a virgola mobile di genere *double*, *'c'* un carattere della categoria *char*, *"ciao"* una stringa e così via. In secondo luogo, anche le espressioni possiedono un tipo. Le espressioni, come si vedrà meglio nella lezione successiva, sono combinazioni di valori ed operatori, che lavorano insieme per calcolare un risultato:

```
5 + 3
```

Questa espressione, ad esempio, è di tipo *int*, ossia calcola un risultato numerico intero appartenente a tale insieme di dati.

```
5 + 1.12
```

Questa, invece, è un'espressione di tipo *double*. Esistono alcune regole, che saranno discusse nel dettaglio successivamente, che consentono di prevedere il tipo associato ad una espressione, analizzando il dominio di appartenenza di ciascun suo elemento.
Le variabili e le costanti risentono delle regole di tipizzazione imposte da Java. Quando si dichiara una variabile, è necessario specificare il tipo ad essa associato. Una variabile di un certo tipo, di conseguenza, non potrà memorizzare dati appartenenti ad una categoria distinta. Tutte le variabili dichiarate *int*, ad esempio, non potranno far altro che tener traccia di valori *int*.

```
int a = 5; // Corretto, 5 è un dato int
int b = "Ciao"; // Sbagliato, "Ciao" è una stringa
```

Conoscere il dominio di appartenenza di un dato significa avere il totale controllo delle sue caratteristiche e delle sue possibilità. I dati numerici, ad esempio, possono andare a comporre delle espressioni algebriche. Le stringhe, invece, dispongono di caratteristiche utili per la manipolazione del testo.
Java costringe lo sviluppatore all'impiego dei tipi, senza alcuna eccezione, e lo fa a fin di bene. Altri linguaggi di programmazione, come Perl, non impongono delle regole di tipizzazione altrettanto rigide. Lo stesso avviene con la maggior parte dei linguaggi di scripting, come JavaScript (il cui nome non deve trarre in inganno, giacché è cosa assai diversa da Java). L'assenza di una forte tipizzazione, di norma, rende più spensierata la stesura di codici non eccessivamente complessi e riduce sensibilmente i tempi di apprendimento necessari per avere una buona padronanza degli strumenti. Tuttavia, a lungo andare, la mancanza di una forte tipizzazione limita le possibilità di sviluppo. C++ è un linguaggio tipizzato. Java è più fortemente tipizzato di C++. C# è tipizzato nella stessa misura di Java. Più in generale, tutti i linguaggi moderni presentano delle smaccate tendenze alla tipizzazione dei dati. Nella tipizzazione, infatti, risiede il sale della programmazione orientata agli oggetti.

4.6 - I tipi predefiniti

Java, teoricamente, dispone di un illimitato numero di tipi. La programmazione orientata agli oggetti permette la definizione di nuove categorie di dati, da sommare a quelle incorporate nel linguaggio e a quelle prodotte e distribuite da terzi. Le classi non sono altro che dei modelli per la definizione di nuovi tipi, che il programmatore può utilizzare e riutilizzare a proprio vantaggio. Nonostante questo, è bene fare distinzione tra tipi predefiniti (detti altrimenti tipi valore o tipi base) e tipi personalizzati (detti anche tipi riferimento o semplicemente classi). Java dispone di otto tipi predefiniti, destinati alla rappresentazione dei valori numerici interi, dei numerici a virgola mobile, dei caratteri e dei booleani. Le stringhe, contrariamente a quanto si può credere durante un primo approccio, non fanno parte dei tipi predefiniti. Ad esse è dedicata un'apposita classe, contenuta nei

pacchetti base della piattaforma. In Java, le stringhe sono degli oggetti.

Ai valori numerici interi sono destinati i quattro tipi riportati in **Tabella 4.2**.

Tipo	Dimensioni	Intervallo
`byte`	8 bit	Da -128 a 127
`short`	16 bit	Da -32768 a 32767
`int`	32 bit	Da -2147483648 a 2147483647
`long`	64 bit	Da -9223372036854775808 a 9223372036854775807

Tabella 4.2
Tipi predefiniti numerici interi.

Scegliere quale tipo numerico intero impiegare, è semplice questione di esigenze. Il più utilizzato è *int*, che dispone di un intervallo adatto alla maggior parte dei casi. Il tipo *byte* è impiegato soprattutto quando si devono manipolare dei dati riposti in file binari (come le immagini ed i campioni sonori), che notoriamente sono suddivisi in blocchi di 8 bit ciascuno (8 bit = 1 byte). Il tipo *long*, che consuma molta memoria ma che copre un intervallo veramente vasto, è usato soprattutto in relazione alle date e agli intervalli temporali espressi in millisecondi. Con il passare del tempo, si acquisirà l'esperienza necessaria per decretare, di volta in volta, quale tipo numerico intero scegliere.

Ai valori numerici a virgola mobile (ossia i numeri reali) sono destinati i due tipi riportati in **Tabella 4.3**.

Tipo	Dimensioni	Precisione
`float`	32 bit	Fino a 7 cifre dopo la virgola
`double`	64 bit	Fino a 16 cifre dopo la virgola

Tabella 4.3
Tipi predefiniti numerici a virgola mobile.

I due domini si differenziano per la precisione con la quale possono effettuare calcoli in virgola mobile. Il tipo *float*, benché meno dispendioso di risorse, garantisce precisione sufficiente per un numero non elevato di cifre decimali. Per questo motivo, è impiegato soprattutto in applicazioni finanziarie, per la rappresentazione delle valute. I valori in Euro o in Dollari, infatti, di rado si spingono oltre i centesimi. Il tipo *double*, invece, garantisce un maggiore margine di precisione, e per questo ben si adatta alle applicazioni matematiche e scientifiche. Non a caso, molte delle funzionalità matematiche incorporate nei pacchetti base di Java lavorano espressamente con i valori *double*.

Alla rappresentazione dei caratteri è destinato il tipo *char*. In forma letterale, i caratteri vanno racchiusi tra una coppia di apici singoli:

```
char c = 'a';
```

Unicode
Maggiori informazioni sulla codifica Unicode possono essere reperite all'indirizzo Web:

http://www.unicode.org/

Java, per la rappresentazione dei caratteri, si attiene alla codifica *Unicode*, che indicizza i simboli di quasi tutti gli alfabeti esistenti. D'altro canto, uno degli scopi di Java è proprio lo sviluppo di software privo di barriere, siano

esse software, hardware o geografiche.
Per poter usufruire di una così vasta gamma di caratteri, i valori di tipo *char* occupano in memoria uno spazio pari a 16 bit, che garantisce 65536 combinazioni distinte.

I dati booleani sono utilizzati per esprimere condizioni di verità. Il loro dominio è costituito da due soli valori: *true* (*vero*) e *false* (*falso*). Ai booleani, Java dedica il tipo predefinito *boolean*:

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

4.7 - I valori letterali

L'utilizzo dei valori letterali è piuttosto intuitivo. Tuttavia, per farne un uso veramente proficuo, è necessario conoscere alcune norme che ne regolano il comportamento.

I letterali interi sono numeri non frazionari, digitati dal programmatore all'interno del codice. *5*, *9*, *15* e *107* sono esempi di letterali interi, espressi in base 10. In alcune situazioni, ad ogni modo, può essere comodo ricorrere ad una forma non decimale, ossia con base differente da 10. Java supporta i letterali interi ottali (con base 8) ed esadecimali (con base 16). Gli ottali possono essere espressi anteponendo uno 0 (zero) al valore digitato:

```
int a = 07;  // 7  ottale = 7  decimale
int b = 012; // 12 ottale = 10 decimale
int c = 047; // 47 ottale = 39 decimale
```

I valori esadecimali vengono espressi impiegando il suffisso *0x* (oppure *0X*):

```
int a = 0x7;  // 7  esadecimale = 7   decimale
int b = 0x12; // 12 esadecimale = 18  decimale
int c = 0xA3; // A3 esadecimale = 163 decimale
```

I letterali interi, di per sé, sono dati di tipo *int*. Tuttavia, un'istruzione come la seguente viene correttamente compilata ed eseguita:

```
byte a = 5;
```

Come mai? Se *5* è un dato *int*, come è possibile assegnarlo ad una variabile di tipo *byte*? Java fornisce, nel caso delle inizializzazioni e delle assegnazioni, un'interpretazione dinamica dei letterali interi dal tipo *int* in giù. Nell'ambito dell'istruzione appena esaminata, *5* viene inteso come un dato di tipo *byte*. L'inizializzazione della variabile, pertanto, risulta corretta. Nonostante questo, bisogna porre attenzione nel non valicare i limiti consentiti. La seguente istruzione, ad esempio, è scorretta:

```
byte a = 200;
```

Il dominio dei dati *byte* ha limite superiore pari a 127. *200*, pertanto, non può essere inteso come un *byte*, e l'inizializzazione della variabile è destinata a fallire. Un discorso analogo può essere portato avanti per il tipo *short*.
Particolare attenzione, infine, deve essere dedicata ai letterali interi che eccedono i limiti fissati per i dati *int*. La seguente istruzione è errata:

```
long a = 3000000000;
```

Benché una variabile di tipo *long* sia in grado di ospitare il valore digitato, non è possibile sfruttare delle costanti letterali che valichino il limite concesso agli *int*, per via della corrispondenza che esiste tra questi due elementi. Per motivi legati alle prestazioni, in questo caso, non interviene alcuna interpretazione dinamica. Il problema va risolto specificando esplicitamente il ricorso al tipo *long*, mediante i suffissi *L* o *l*:

```
long a = 3000000000L;
long b = 06712671267162L;
long c = 0xFF17218A5D4L;
```

In questo modo, il letterale digitato sarà direttamente valutato come un dato di tipo *long*.

I letterali a virgola mobile sono numeri decimali con una componente frazionaria. Java supporta due tipi di notazione. La più diffusa, detta notazione standard, utilizza un punto per separare la parte intera da quella frazionaria:

```
1.5   2.0   4.876
```

La notazione scientifica ingloba la notazione standard, fornendo in aggiunta una potenza di dieci per la quale deve essere moltiplicato il numero espresso. La separazione tra le due parti avviene mediante l'uso di una *E* o di una *e*. In pratica:

$1.023E5 = 1.023 \times 10^{5} = 102300$
$4.2E-4 = 4.2 \times 10^{-4} = 0.00042$

La notazione scientifica è molto comoda quando si devono esprimere valori frazionari piuttosto piccoli.
Tutti i letterali a virgola mobile vengono considerati dal compilatore come valori di tipo *double*. Per specificare un letterale di tipo *float*, è sufficiente ricorrere ai suffissi *F* o *f*:

```
float a = 4.5F;
float b = 1.2E-3F;
```

Allo stesso modo, i suffissi *D* e *d* possono essere impiegati per specificare esplicitamente dei letterali *double*:

```
double a = 4.5D;
double b = 1.2E-3D;
```

Ad ogni modo, poiché *double* è la scelta predefinita in assenza di suffissi, il ricorso a questa possibilità è, il più delle volte, ridondante.

I letterali booleani sono costituiti dalle due parole *true* e *false*. Il loro utilizzo, come già dimostrato in precedenza, è immediato:

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

Chi proviene da C o da C++ deve sapere che Java, a differenza di questi linguaggi, non assegna alcun significato numerico ai valori booleani. Di conseguenza, *0* e *1* non sono letterali booleani validi. I valori booleani, inoltre, non possono essere impiegati in espressioni algebriche. In definitiva, i termini *true* e *false* non sono degli alias per *1* e *0*, ma sono dei valori a sé stanti, non rappresentabili in altro modo.

I caratteri hanno rappresentazione letterale racchiusa tra una coppia di apici singoli. La maggior parte dei caratteri disponibili a video può essere espressa in questo modo. *'a'*, *'U'* e *'#'*, ad esempio, sono rappresentazioni valide. Altri caratteri, invece, non possono ricorrere a questa forma. E' il caso dei ritorni a capo, degli stessi apici singoli e di molti altri. Per risolvere il problema, Java mette a disposizione delle particolari sequenze di escape, mostrate in **Tabella 4.4**.

Sequenza	Significato
`\'`	Apice singolo
`\"`	Apice doppio
`\\`	Barra retroversa
`\r`	Ritorno a capo
`\n`	Nuova riga
`\f`	Avanzamento modulo
`\t`	Tabulazione
`\b`	Backspace

Tabella 4.4
Sequenze di escape per i caratteri letterali.

Le sequenze di escape si usano al seguente modo:

```
char a = '\n'; // Avanzamento a nuova riga
char b = '\''; // Apice singolo '
char c = '\\'; // Barra retroversa \
```

Più in generale, è possibile esprimere un qualsiasi carattere Unicode indicando il suo indice con una notazione ottale in forma *'\xxx'*, oppure con una notazione esadecimale in forma *'\uxxxx'*. Ad esempio:

```
char a = '\141';   // 141 è l'indice ottale di 'a'
char b = '\u0061'; // 61 è l'indice esadecimale di 'a'
```

I valori *char*, infine, vengono trattati alla stregua di numerici interi senza segno (l'intervallo del dominio *char* varia da 0 a 65536). Questo significa che è possibile assegnare un intero ad un carattere e viceversa. Benché questa caratteristica, essendo disorientante, venga poco usata, rimane possibile specificare l'indice di un carattere Unicode senza ricorrere a particolari delimitatori. Ad esempio, sapendo che l'indice in base 10 del carattere *'a'* è 97, è possibile scrivere:

```
char c = 97; // 97 è l'indice decimale di 'a'
```

Ad ogni modo, si sconsiglia l'uso di questa forma, mentre si raccomanda l'impiego delle sequenze di escape per i caratteri di natura problematica.

Le stringhe, benché non facciano parte dei tipi predefiniti di Java, godono di una particolare rappresentazione letterale. Una stringa può essere espressa in forma letterale come una sequenza di caratteri racchiusa tra una coppia di apici doppi:

```
String a = "Sono una stringa";
```

All'interno delle stringhe, le spaziature assumono significato letterale, e pertanto non è più

possibile usarle liberamente come si può fare nel resto del codice. Una stringa letterale, inoltre, deve sempre essere contenuta in una sola riga. La seguente istruzione, pertanto, non risulta valida:

```
// Errore!
String a = "Sono
        una stringa";
```

Questa istruzione genera una catena di errori:

```
unclosed string literal
String a = "Sono
           ^
unclosed string literal
        una stringa";
                   ^
';' expected
        una stringa";
                    ^
```

Il primo errore è dovuto al fatto che il compilatore, prima del termine della riga, si attende il carattere di chiusura della stringa. Il secondo è del tutto analogo, visto che i doppi apici impiegati nella seconda linea del frammento vengono ora intesi come l'apertura di una nuova stringa letterale, anch'essa non correttamente terminata. Infine, il punto e virgola è interpretato come parte integrate della nuova stringa accidentalmente aperta, e pertanto il compilatore reclama la chiusura tanto del valore letterale quanto dell'intera istruzione. Insomma, mai distribuire una stringa su più righe a questo modo! Il seguente trucco, basato sull'unione di più valori letterali immessi separatamente, è invece lecito e consigliato nel caso di testi particolarmente lunghi:

```
String a = "";
a += "Ciao, ";
a += "sono una ";
a += "stringa distribuita ";
a += "su più righe grazie ";
a += "ad uno stratagemma...";
```

Il reale significato tecnico di questa forma sarà trattato nella lezione successiva.
Per la rappresentazione dei caratteri problematici, come i ritorni a capo, è possibile impiegare le medesime sequenze di escape valide per i caratteri. Si verifichi l'affermazione compilando e mandando in esecuzione il seguente programma:

```
public class StringheLetterali {

      public static void main(String[] args) {
            String a = "Prima riga.\nSeconda riga.";
            System.out.println(a);
      }

}
```

I programmatori C/C++ ricordino sempre che, in Java, le stringhe sono rese come oggetti, e non come array di caratteri.

4.8 - Compatibilità tra tipi e conversioni

La forte tipizzazione dei dati adottata Java è un gran vantaggio. Un approccio di questo

genere, ad ogni modo, deve pur sempre ammettere un certo margine di elasticità, altrimenti l'eccessivo rigore finirebbe per complicare il codice in maniera spropositata. Ci sono dei casi in cui un valore di un certo tipo può essere cambiato, automaticamente o manualmente, in un dato di differente categoria. Alcuni tipi di valori, infatti, godono di un certo grado di compatibilità con altri domini. Java dispone di due meccanismi di conversione: la *coercion* (in italiano *coercizione*, che è grossomodo un sinonimo di *costrizione*) ed il *casting* (termine che può essere inteso nel significato di adattamento).

4.9 - La coercion, ossia la conversione implicita

La *coercion* (o *promozione*) è una conversione che il linguaggio, in determinati casi, può eseguire automaticamente. In **Tabella 4.5** sono riportate le conversioni implicite valide per i tipi predefiniti.

Tipo	Conversione
`byte`	`short, int, long, float, double`
`short`	`int, long, float, double`
`int`	`long, float, double`
`long`	`float, double`
`float`	`double`
`double`	nessuna coercion ammessa
`char`	`int, long, float, double`
`boolean`	nessuna coercion ammessa

Tabella 4.5
Le conversioni implicite ammesse sui tipi predefiniti di Java.

Un estratto di codice come il seguente, dunque, è corretto e funzionante:

```
int a = 50;
long b = a;
```

In effetti, è facile rendersi conto di come la conversione tentata sia effettivamente semplice, implicita e possibile. Il tipo *long* include al suo interno l'intero intervallo che caratterizza i dati *int*. Qualsiasi *int*, pertanto, può essere mutato senza problemi in un *long*. Il compilatore non crea problemi al riguardo.

4.10 - Il casting, ossia la conversione esplicita

Il seguente programma tenta due conversioni che non possono essere effettuate implicitamente:

```
public class ConversioniErrate {

	public static void main(String[] args) {
		int a = 1;
		// Da int a boolean
		boolean b = a;
		// Da int a byte
		byte c = a;
	}

}
```

Il compilatore, come prevedibile, restituisce due messaggi di errore. Il primo riguarda la conversione da *int* a *boolean*:

```
incompatible types
found   : int
required: boolean
    boolean b = a;
                ^
```

Il testo del messaggio è "incompatible types", ossia "tipi non compatibili". In casi come questo, non c'è nulla da fare: la conversione tentata prevede un'operazione illogica, che in alcun modo può essere effettuata. Non è un problema: ciò che è illogico non potrà mai tornare utile.
Il secondo errore restituito, al contrario, suona diversamente:

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

L'avviso, questa volta, è "possible loss of precision", ossia "possibile perdita di precisione". Tanto *int* quanto *byte* sono domini che fanno riferimento a valori numerici. Il passaggio dall'uno all'altro, pertanto, è logico e ammissibile. La conversione da *byte* a *int*, addirittura, può essere effettuata implicitamente. Perché, dunque, non avviene lo stesso con la trasformazione inversa? Il tipo *int* abbraccia un intervallo più vasto del tipo *byte*. Qualsiasi valore *byte*, pertanto, ha sicuramente un equivalente nello spazio degli *int*. Non avviene, però, il contrario. Il dominio *int* ammette dei valori che non esistono nello spazio dei *byte*. Un esempio è l'intero *200*, che può essere memorizzato in una variabile *int*, ma non in una variabile *byte* (il limite superiore di questo tipo, si ricorda ancora una volta, è 127):

```
int a = 200;  // Corretto
byte b = 200; // Errato, 200 è troppo elevato per il tipo byte
```

Il *casting*, ossia la conversione esplicita (che personalmente chiamo anche *conversione consapevole*), è un meccanismo che consente di eseguire le trasformazioni a rischio di una perdita di precisione. Il modello sintattico per le operazioni di casting è mostrato di seguito:

```
(nuovo_tipo) valore
```

Ad esempio:

```
int a = 5;
byte b = (byte) a;
```

Questo frammento di codice richiede esplicitamente la conversione del valore *int* contenuto nella variabile *a* in un dato *byte*, in modo che questo possa essere memorizzato in una variabile di quest'ultimo tipo. Il compilatore, di fronte alla dichiarazione esplicita, non solleva obiezioni ed accetta il codice.
L'impiego del casting è una maniera per dichiarare la consapevolezza di una perdita di precisione. Un po' come dire: "ok, caro compilatore, so che la conversione che ti richiedo è potenzialmente problematica, ma accettala ugualmente ed io me ne assumerò la piena responsabilità". Il seguente programma dimostra quali siano le conseguenze di un tale approccio:

```
public class Casting {

	public static void main(String[] args) {
		// Primo test, che si risolve senza problemi.
		int a1 = 5;
		byte b1 = (byte)a1;
		System.out.println("Dato originale (int): " + a1);
		System.out.println("Dato convertito (byte): " + b1);
		// Secondo test, problematico (872 è oltre il range dei byte).
		int a2 = 872;
		byte b2 = (byte)a2;
		System.out.println("Dato originale (int): " + a2);
		System.out.println("Dato convertito (byte): " + b2);
	}

}
```

L'output prodotto è mostrato di seguito:

```
Dato originale (int): 5
Dato convertito (byte): 5
Dato originale (int): 872
Dato convertito (byte): 104
```

Nel primo caso, tutto è andato nel migliore dei modi. Giacché 5 è un valore condiviso da *int* e *byte*, non si riscontra nessun problema. Diversamente avviene con *872*, un valore che in alcun modo può essere convertito in un *byte*. In questo caso, si è verificata la famigerata perdita di precisione citata sopra. Il valore ottenuto (104) è apparentemente calcolato in maniera illogica. In realtà, deriva da una perdita di bit nella rappresentazione binaria di 872 (i calcolatori, internamente, gestiscono i dati in forma binaria). Giusto per chiarire meglio le idee:

```
872 decimale = 1101101000 binario
```

Per la memorizzazione del valore occorrono 10 bit, tanti quanti i numeri che compongono la sua rappresentazione binaria. Il tipo *int*, che dispone di 32 bit, può tranquillamente memorizzare la cifra. Il tipo *byte*, al contrario, impiega solamente 8 bit. Nel momento in cui avviene il casting, i due bit più a sinistra vengono scartati, per fare in modo che il dato possa rientrare nello spazio concesso. La perdita, ovviamente, influisce sul valore rappresentato:

```
1101101000 binario = 872 decimale
  01101000 binario = 104 decimale
```

Tralasciando le spiegazioni di carattere più tecnico, che nella maggior parte dei contesti sono irrilevanti, l'esperienza insegna come le conversioni esplicite vadano usate con cautela e consapevolezza, pena il riscontro di valori a prima vista inspiegabili.
In altri casi, ad ogni modo, la perdita di precisione è una fonte di preoccupazione di minore entità. Ad esempio, nel passaggio da un *double* di ridotte dimensioni ad un *int*, tutto quello che si perde è la parte frazionaria del valore rappresentato:

```
double a = 4.561;
int b = (int) a; // b vale 4
```

In un caso come questo, la perdita potrebbe anche essere esplicitamente ricercata dal programmatore, per simulare un'operazione di troncamento.

Il casting può essere applicato a qualsiasi tipo di valore, sia esso espresso letteralmente, contenuto in una variabile, calcolato da un'espressione o restituito da una chiamata a funzione:

```
int a = (int) 5.12; // Da valore letterale
int b = (int) nomeVariabile; // Da variabile
int c = (int) (nomeVariabile - 2.189); // Da espressione
int d = (int) Math.sqrt(3); // Da chiamata a funzione
```