

Capitolo 7

Un'occhiata più attenta a metodi e classi

- 7.1 **Sovraccaricare i metodi**
- 7.2 **Utilizzare gli oggetti come parametri**
- 7.3 **Un'occhiata più attenta al passaggio degli argomenti**
- 7.4 **Restituire gli oggetti**
- 7.5 **Ricorsione**
- 7.6 **Introduzione al controllo di accesso**
- 7.7 **Comprendere static**
- 7.8 **Introduzione a final**
- 7.9 **Nuovo esame degli array**
- 7.10 **Introduzione alle classi annidate e interne**
- 7.11 **Esplorare la classe String**
- 7.12 **Utilizzare gli argomenti della riga di comando**
- 7.13 **Varargs: argomenti a lunghezza variabile**

Questo capitolo prosegue la discussione su metodi e classi iniziata nel capitolo precedente. Esamina diversi argomenti relativi ai metodi, compreso il sovraccarico, il passaggio di parametri e la ricorsione. Il capitolo torna quindi alle classi, discutendo il controllo di accesso, l'utilizzo della parola chiave `static` e una delle più importanti classi incorporate di Java: `String`.

7.1 Sovraccaricare i metodi

In Java è possibile definire all'interno della stessa classe due o più metodi che condividono lo stesso nome, sempre che le dichiarazioni di parametri siano diverse. Quando si verifica questa situazione, si dice che i metodi sono *sovraccaricati* e il processo viene chiamato *sovraccarico dei metodi*. Il sovraccarico dei metodi è uno dei modi con cui Java implementa il polimorfismo. Se non è mai stato utilizzato un linguaggio che consente il sovraccarico dei metodi,

all'inizio il concetto potrebbe sembrare strano ma, come si vedrà, è una delle funzioni più utili e interessanti di Java.

Quando viene chiamato un metodo sovraccaricato, per determinarne la versione da chiamare Java utilizza come guida il tipo e/o il numero di argomenti. Di conseguenza, i metodi sovraccaricati devono differire nel tipo e/o nel numero di parametri. Mentre i metodi sovraccaricati possono avere tipi di ritorno diversi, il tipo di ritorno in sé non è sufficiente per distinguere due versioni di un metodo. Quando Java trova una chiamata a un metodo sovraccaricato, esegue semplicemente la versione del metodo i cui parametri corrispondono agli argomenti utilizzati nella chiamata.

Di seguito è mostrato un esempio che illustra il sovraccarico di metodi:

```
// Dimostra il sovraccarico dei metodi.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Sovraccarica test con un parametro intero.
    void test(int a) {
        System.out.println("a: " + a);
    }
    // Sovraccarica test con due parametri interi.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // sovraccarica test con un parametro double
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        // chiama tutte le versioni di test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

Questo programma genera il seguente output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Come si può vedere, `test()` viene sovraccaricato quattro volte. La prima versione non utilizza alcun parametro, la seconda prende un parametro intero, la terza due parametri interi e la quarta un parametro `double`. Il fatto che la quarta versione di `test()` restituisca anche un valore non è una conseguenza relativa al sovraccarico, poiché i tipi di ritorno non hanno alcun ruolo nella determinazione del sovraccarico.

Quando viene chiamato un metodo sovraccaricato, Java cerca una corrispondenza tra gli argomenti utilizzati per chiamare il metodo e i parametri di quest'ultimo, ma questa corrispondenza non deve necessariamente essere esatta. In alcuni casi la conversione automatica dei tipi di Java può giocare un ruolo nella determinazione del sovraccarico. Per esempio, si consideri il seguente programma:

```
// Conversione automatica del tipo applicata al sovraccarico.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Sovraccarica test con due parametri interi.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // sovraccarica test con un parametro double
    void test(double a) {
        System.out.println("Inside test(double) a: " + a);
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;
        ob.test();
        ob.test(10, 20);
        ob.test(i); // questo chiamera' test(double)
        ob.test(123.2); // questo chiamera' test(double)
    }
}
```

Questo programma genera il seguente output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

Come si può vedere, questa versione di `OverloadDemo` non definisce `test(int)`. Di conseguenza, quando all'interno di `Overload` viene chiamato `test()` con un argomento intero, non viene trovato alcun metodo corrispondente. Tuttavia, Java può convertire automaticamente un intero in un `double`, e questa conversione può essere utilizzata per risolvere la chiamata. Quindi,

quando non trova `test(int)` Java eleva `i` a `double` e poi chiama `test(double)`. Naturalmente, se fosse stato definito `test(int)`, sarebbe stato chiamato. Java utilizza la sua conversione automatica del tipo solo quando non trova una corrispondenza esatta.

Il sovraccarico dei metodi supporta il polimorfismo, perché è un modo con cui Java implementa il paradigma “un’interfaccia, più metodi”. Per comprendere come, si consideri quanto segue. Nei linguaggi che non supportano il sovraccarico dei metodi, a ogni metodo deve essere assegnato un nome univoco; tuttavia, spesso si desidera implementare essenzialmente lo stesso metodo per tipi di dati diversi. Si consideri la funzione di valore assoluto: nei linguaggi che non supportano il sovraccarico, generalmente esistono tre o più versioni di questa funzione, ciascuna con un nome leggermente diverso. Per esempio, in C la funzione `abs()` restituisce il valore assoluto di un intero, `labs()` restituisce il valore assoluto di un intero lungo e `fabs()` restituisce il valore assoluto di un valore in virgola mobile. Poiché C non supporta il sovraccarico, ciascuna funzione deve avere il proprio nome, anche se tutte e tre svolgono essenzialmente gli stessi compiti. Questo rende la situazione più complessa, dal punto di vista concettuale, di quanto non lo sia realmente. Sebbene il concetto portante di ciascuna funzione sia lo stesso, è necessario ricordare i tre nomi. Questa situazione non si verifica in Java, perché ogni metodo con valore assoluto può utilizzare lo stesso nome. Di fatto, la libreria di classi standard di Java comprende un metodo con valore assoluto, chiamato `abs()`, sovraccaricato dalla classe `Math` di Java in modo da gestire tutti i tipi numerici. Java determina la versione di `abs()` da chiamare in base al tipo di argomento.

Il merito del sovraccarico è consentire di accedere a metodi simili utilizzando un nome comune: il nome `abs` rappresenta l’azione generale da eseguire. È compito del compilatore scegliere la versione *specifica* per una determinata circostanza; il programmatore deve solo ricordare l’operazione generale da eseguire. Con l’applicazione del polimorfismo, molti nomi sono stati raggruppati in uno solo. Sebbene questo esempio sia piuttosto semplice, se si espande il concetto si può vedere come il sovraccarico possa essere utile per la gestione di complessità maggiori.

Quando si sovraccarica un metodo, ciascuna versione di quel metodo può eseguire qualsiasi attività: non esiste una regola che imponga che i metodi sovraccaricati devono essere in relazione tra loro. Tuttavia, da un punto di vista stilistico, il sovraccarico dei metodi implica una relazione. Quindi, anche se è possibile utilizzare lo stesso nome per sovraccaricare metodi non correlati, non si dovrebbe fare. Per esempio, è possibile utilizzare il nome `sqr` per creare metodi che restituiscono il quadrato di un intero e la radice quadrata di un valore in virgola mobile, ma queste due operazioni sono profondamente diverse. L’applicazione del sovraccarico dei metodi in questo modo non corrisponde al suo scopo originale: nella pratica si dovrebbero sovraccaricare solo operazioni strettamente collegate.

Sovraccaricare i costruttori

Oltre al sovraccarico di metodi normali, è anche possibile sovraccaricare i metodi dei costruttori. Di fatto, per la maggior parte delle classi reali create, i costruttori sovraccaricati saranno la regola, non l'eccezione. Per comprenderne il motivo, si tornerà alla classe `Box` sviluppata nel capitolo precedente. Di seguito è mostrata l'ultima versione di `Box`:

```
class Box {
    double width;
    double height;
    double depth;
    // Questo e' il costruttore per Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // calcola e restituisce il volume
    double volume() {
        return width * height * depth;
    }
}
```

Come si può vedere, il costruttore `Box()` richiede tre parametri. Questo significa che tutte le dichiarazioni di oggetti `Box` devono passare a `Box()` tre argomenti. Per esempio, la dichiarazione che segue non è valida:

```
Box ob = new Box();
```

Poiché `Box()` richiede tre argomenti, è un errore chiamarlo senza di essi. Questo solleva alcune domande importanti. Come fare se si desidera semplicemente una scatola e non ci si preoccupa (o non si ha conoscenza) delle sue dimensioni iniziali? Oppure, come fare se si desidera essere in grado di inizializzare un cubo specificando solo un valore che verrà utilizzato per le tre dimensioni? Per come è scritta ora la classe `Box`, queste opzioni non sono disponibili.

Fortunatamente, la soluzione a questi problemi è piuttosto semplice: sovraccaricare il costruttore di `Box` in modo che gestisca le situazioni appena descritte. Ecco il programma con una versione migliorata di `Box`, che esegue tali operazioni:

```
/* Qui, Box definisce tre costruttori per inizializzare le dimensioni
di una scatola in diversi modi.
*/
class Box {
    double width;
    double height;
    double depth;
```

```
// costruttore utilizzato quando tutte le dimensioni vengono specificate
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}
// costruttore utilizzato quando non viene specificata alcuna dimensione
Box() {
    width = -1; // utilizza -1 per indicare
    height = -1; // una scatola
    depth = -1; // non inizializzata
}
// costruttore utilizzato quando viene creato il cubo
Box(double len) {
    width = height = depth = len;
}
// calcola e restituisce il volume
double volume() {
    return width * height * depth;
}
}
class OverloadCons {
    public static void main(String args[]) {
        // crea le scatole utilizzando i vari costruttori
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;
        // ottiene il volume della prima scatola
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        // ottiene il volume della seconda scatola
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        // ottiene il volume del cubo
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
    }
}
```

L'output prodotto da questo programma è il seguente:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Come si può vedere, viene chiamato il costruttore sovraccaricato adatto basandosi sui parametri specificati quando viene eseguita `new`.

7.2 Utilizzare gli oggetti come parametri

Fino a questo punto come parametri dei metodi sono stati utilizzati solo tipi semplici; tuttavia, è corretto e consueto passare gli oggetti ai metodi. Per esempio, si consideri il breve programma che segue:

```
// Gli oggetti possono essere passati ai metodi.
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    // restituisce true se o e' uguale all'oggetto chiamante
    boolean equals(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}
class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: " + ob1.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}
```

Questo programma genera il seguente output:

```
ob1 == ob2: true
ob1 == ob3: false
```

Il metodo `equals()` all'interno di `Test` confronta due oggetti per vedere se sono uguali e restituisce il risultato, cioè confronta l'oggetto chiamante con quello passato. Se gli oggetti contengono gli stessi valori, il metodo restituisce `true`, altrimenti restituisce `false`. Si noti che il parametro `o` in `equals()` specifica `Test` come suo tipo. Sebbene `Test` sia un tipo di classe creato dal programma, è utilizzato allo stesso modo dei tipi incorporati di Java.

Uno degli utilizzi più comuni dei parametri degli oggetti implica i costruttori. Spesso si desidererà costruire un nuovo oggetto in modo che inizialmente sia uguale a qualche oggetto esistente: per eseguire questa operazione è necessario definire un costruttore che prenda un oggetto della sua classe come parametro. Per esempio, la seguente versione di `Box` consente a un oggetto di inizializzarne un altro:

```
// Qui, Box consente a un oggetto di inizializzarne un altro.
class Box {
```

```
double width;
double height;
double depth;
// costruisce il clone di un oggetto
Box(Box ob) { // passa l'oggetto al costruttore
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}
// costruttore utilizzato quando vengono specificate tutte le dimensioni
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}
// costruttore utilizzato quando non viene specificata alcuna dimensione
Box() {
    width = -1; // utilizza -1 per indicare
    height = -1; // una scatola
    depth = -1; // non inizializzata
}
// costruttore utilizzato quando viene creato il cubo
Box(double len) {
    width = height = depth = len;
}
// calcola e restituisce il volume
double volume() {
    return width * height * depth;
}
}
class OverloadCons2 {
    public static void main(String args[]) {
        // crea le scatole utilizzando i vari costruttori
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        Box myclone = new Box(mybox1);
        double vol;
        // ottiene il volume della prima scatola
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        // ottiene il volume della seconda scatola
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        // ottiene il volume del cubo
        vol = mycube.volume();
        System.out.println("Volume of cube is " + vol);
        // ottiene il volume del clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    }
}
```

Come si vedrà quando si creeranno le proprie classi, in genere è richiesta la fornitura di molte forme di costruttori per consentire agli oggetti di essere costruiti in modo utile ed efficace.

7.3 Un'occhiata più attenta al passaggio degli argomenti

In generale, un linguaggio dispone di due modi per passare un argomento a una subroutine. Il primo è la *chiamata per valore*, che copia il *valore* di un argomento nel parametro formale della subroutine. Di conseguenza, le modifiche eseguite sul parametro della subroutine non hanno effetto sull'argomento. Il secondo modo è la *chiamata per riferimento*: con questo metodo viene passato al parametro un riferimento a un argomento (non il valore dell'argomento). All'interno della subroutine, il riferimento viene utilizzato per accedere all'argomento reale specificato nella chiamata. Questo significa che le modifiche apportate al parametro avranno effetto sull'argomento utilizzato per chiamare la subroutine. Come si vedrà, Java utilizza entrambi i metodi, a seconda di ciò che viene passato.

In Java, quando si passa un tipo primitivo a un metodo, viene passato per valore, quindi ciò che accade al parametro che riceve l'argomento non ha effetto al di fuori del metodo. Per esempio, si consideri il seguente programma:

```
// I tipi primitivi vengono passati per valore.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}
class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a and b before call: " +
            a + " " + b);
        ob.meth(a, b);
        System.out.println("a and b after call: " +
            a + " " + b);
    }
}
```

L'output di questo programma è il seguente:

```
a and b before call: 15 20
a and b after call: 15 20
```

Le operazioni che avvengono all'interno di `meth()` non hanno effetto sui valori di `a` e `b` utilizzati nella chiamata; i loro valori qui non vengono cambiati in 30 e 10.

Quando si passa un oggetto a un metodo, la situazione cambia radicalmente poiché gli oggetti vengono passati per riferimento. È necessario ricordare che quando si crea una variabile di un tipo di classe, si crea solo un riferimento a un oggetto. Di conseguenza, quando si passa tale riferimento a un metodo, il parametro che lo riceve farà riferimento allo stesso oggetto al quale fa riferimento l'argomento. Questo significa che gli oggetti vengono passati ai metodi attraverso l'utilizzo di una chiamata per riferimento. Le modifiche all'oggetto all'interno del metodo hanno effetto sull'oggetto utilizzato come argomento. Per esempio, si consideri il seguente programma:

```
// Gli oggetti vengono passati per riferimento.
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    // passa un oggetto
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}
class CallByRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " +
            ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: " +
            ob.a + " " + ob.b);
    }
}
```

Questo programma genera il seguente output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

In questo caso le azioni all'interno di `meth()` hanno modificato l'oggetto utilizzato come argomento.

Quando viene passato a un metodo un riferimento a un oggetto, il riferimento stesso viene passato utilizzando una chiamata per valore. Tuttavia, poiché il valore passato si riferisce a un oggetto, la copia di tale valore fa ancora riferimento allo stesso oggetto a cui si riferisce l'argomento corrispondente.

ATTENZIONE *Quando viene passato a un metodo un tipo primitivo, l'operazione viene eseguita con una chiamata per valore. Gli oggetti vengono passati implicitamente con una chiamata per riferimento.*

7.4 Restituire gli oggetti

Un metodo può restituire qualsiasi tipo di dati, compresi i tipi di classe creati dal programmatore. Per esempio, nel programma che segue il metodo `incrByTen()` restituisce un oggetto nel quale il valore di `a` è maggiore di dieci unità rispetto a quello dell'oggetto chiamante.

```
// Restituire un oggetto.
class Test {
    int a;
    Test(int i) {
        a = i;
    }
    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}
class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
            + ob2.a);
    }
}
```

L'output generato da questo programma è il seguente:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Come si può notare, ogni volta che viene chiamato `incrByTen()` si crea un nuovo oggetto e alla routine chiamante viene restituito un riferimento a esso.

Il programma precedente evidenzia un altro punto importante: poiché utilizzando `new` tutti gli oggetti vengono allocati in modo dinamico, non è necessario preoccuparsi di oggetti che usciranno dall'ambito quando termina il metodo nel quale sono stati creati. L'oggetto continuerà a esistere fintanto che ci sarà un riferimento a esso in qualche punto del programma. Quando non esistono riferimenti a esso, l'oggetto verrà eliminato durante l'operazione di garbage collection successiva.

7.5 Ricorsione

Java supporta la *ricorsione*, cioè il processo di definizione di qualcosa in termini di se stessa. Nella programmazione Java la ricorsione è l'attributo che consente a un metodo di chiamare se stesso; un metodo che chiama se stesso si dice *ricorsivo*.

Un esempio classico di ricorsione è il calcolo del fattoriale di un numero. Il fattoriale di un numero N è il prodotto di tutti i numeri interi compresi tra 1 e N . Per esempio, 3 fattoriale è $1 \times 2 \times 3$, o 6. Di seguito è mostrato come può essere calcolato un fattoriale utilizzando un metodo ricorsivo:

```
// Un semplice esempio di ricorsione.
class Factorial {
    // questo e' un metodo ricorsivo
    int fact(int n) {
        int result;
        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}
class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

L'output di questo programma è il seguente:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Se non si ha familiarità con i metodi ricorsivi, l'operazione `fact()` potrebbe sembrare complessa. Ecco come funziona: quando viene chiamato `fact()` con argomento 1, la funzione restituisce 1; negli altri casi restituisce il prodotto di `fact(n-1)*n`. Per calcolare questa espressione, `fact()` viene chiamato con $n-1$. Questo processo viene ripetuto fino a quando n è uguale a 1 e le chiamate al metodo iniziano la restituzione.

Per comprendere meglio il funzionamento del metodo `fact()` si osserverà un breve esempio. Quando si calcola il fattoriale di 3, la prima chiamata a `fact()` causerà una seconda chiamata con argomento 2. Questa chiamata fa in modo che `fact()` venga chiamato una terza volta con argomento 1. Questa chiamata restituirà 1, che viene poi moltiplicato per 2 (il valore di n nella seconda chiamata). Il risultato (che è 2) viene quindi restituito alla chiamata originale di `fact()` e moltiplicato per 3 (il valore originale di n). Questo ge-

nera la risposta, 6. Si potrebbe trovare interessante l'inserimento in `fact()` di dichiarazioni `println()`, che mostreranno il livello in cui si trova ciascuna chiamata e le risposte intermedie.

Quando un metodo chiama se stesso, vengono allocate nello stack nuove variabili e parametri locali e il codice del metodo viene eseguito dall'inizio con queste nuove variabili. Quando ciascuna chiamata ricorsiva ritorna, le variabili e i parametri locali vecchi vengono rimossi dallo stack e l'esecuzione riprende dal punto della chiamata all'interno del metodo. Si potrebbe dire che i metodi ricorsivi si "incastrano" l'uno nell'altro.

Le versioni ricorsive di molte routine possono essere eseguite un po' più lentamente dei loro equivalenti iterativi, a causa del sovraccarico aggiunto dalle chiamate di funzioni addizionali. Molte chiamate ricorsive a un metodo possono causare un overrun dello stack. Poiché la memorizzazione dei parametri e delle variabili locali avviene nello stack e ogni nuova chiamata crea una nuova copia di tali variabili, è possibile che lo stack venga esaurito. Se si verifica tale situazione, il sistema di runtime di Java genera un'eccezione. Tuttavia, probabilmente non sarà necessario preoccuparsene, a meno di avere una routine ricorsiva che cresce a dismisura.

Il principale vantaggio dei metodi ricorsivi è che possono essere utilizzati per creare versioni più chiare e più semplici di numerosi algoritmi rispetto alle loro versioni iterative. Per esempio, l'algoritmo di ordinamento `QuickSort` è piuttosto difficile da implementare in modo iterativo. Alcuni problemi, in particolare quelli relativi all'intelligenza artificiale, sembrano portare a soluzioni ricorsive. Infine, sembra che alcune persone pensino più facilmente in modo ricorsivo piuttosto che in modo iterativo.

Quando si scrivono i metodi ricorsivi è necessario avere una dichiarazione `if` in qualche punto, in modo da costringere il metodo a restituire il controllo senza che sia eseguita la chiamata ricorsiva. Se non si esegue questa operazione, una volta chiamato il metodo questo non ritornerà mai, un errore molto comune quando si lavora con la ricorsione. Si consiglia di utilizzare numerose dichiarazioni `println()` durante lo sviluppo, in modo da poter osservare ciò che accade e annullare l'esecuzione se si nota che è stato commesso un errore.

Di seguito è mostrato un altro esempio di ricorsione. Il metodo ricorsivo `printArray()` stampa i primi elementi `i` dell'array `values`.

```
// Un altro esempio che utilizza la ricorsione.
class RecTest {
    int values[];
    RecTest(int i) {
        values = new int[i];
    }
    // visualizza l'array -- in modo ricorsivo
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println "[" + (i-1) + "]" + values[i-1];
    }
}
```

```
}  
class Recursion2 {  
    public static void main(String args[]) {  
        RecTest ob = new RecTest(10);  
        int i;  
        for(i=0; i<10; i++) ob.values[i] = i;  
        ob.printArray(10);  
    }  
}
```

Il programma genera il seguente output:

```
[0] 0  
[1] 1  
[2] 2  
[3] 3  
[4] 4  
[5] 5  
[6] 6  
[7] 7  
[8] 8  
[9] 9
```

7.6 Introduzione al controllo di accesso

È noto che l'incapsulamento collega i dati con il codice che li manipola. Tuttavia, l'incapsulamento fornisce un altro importante attributo: il *controllo di accesso*. Attraverso l'incapsulamento è possibile controllare quali parti di un programma possono accedere ai membri di una classe. Controllando l'accesso è possibile evitare usi impropri. Per esempio, consentendo l'accesso ai dati solo attraverso un set di metodi ben definito, è possibile evitare un uso improprio di tali dati. Di conseguenza, se implementata correttamente, una classe crea una "scatola nera" che può essere utilizzata, ma i cui funzionamenti interni non possono essere manipolati. Tuttavia, le classi presentate in precedenza non raggiungono completamente questo obiettivo. Per esempio, si consideri la classe Stack mostrata alla fine del Capitolo 6. Anche se i metodi `push()` e `pop()` forniscono allo stack un'interfaccia controllata, questa interfaccia non è solida; vale a dire che per un'altra parte del programma è possibile raggiungere i metodi e accedere direttamente allo stack. Naturalmente, nelle mani sbagliate questo potrebbe essere un problema. In questo paragrafo verrà presentato il meccanismo con il quale è possibile controllare l'accesso ai vari membri di una classe.

Il modo in cui è possibile accedere a un membro viene determinato dallo *specificatore di accesso*, che modifica la sua dichiarazione. Java fornisce una vasta gamma di specificatori di accesso. Alcuni aspetti del controllo di accesso sono legati prevalentemente all'ereditarietà o ai package (un *package* è, in sostanza, un raggruppamento di classi). Queste parti del meccanismo di controllo di accesso di Java saranno descritte più avanti; per il momento si

inizierà a esaminare il controllo di accesso applicato a una singola classe. Una volta comprese le basi del controllo di accesso, il resto sarà semplice.

Gli specificatori di accesso di Java sono `public`, `private` e `protected`; Java definisce inoltre un livello di accesso predefinito. `protected` si applica solo quando è implicata l'ereditarietà, mentre gli altri specificatori di accesso sono descritti di seguito.

Si inizierà definendo `public` e `private`. Quando un membro di una classe viene modificato dallo specificatore `public`, quel membro è accessibile a qualsiasi altro codice. Quando un membro di una classe è specificato come `private`, quel membro è accessibile solo da altri membri della sua classe. Ora è possibile comprendere perché `main()` è sempre stato preceduto dallo specificatore `public`: viene chiamato dal codice esterno al programma, cioè dal sistema di runtime di Java. Quando non viene utilizzato alcuno specificatore di accesso, per impostazione predefinita il membro di una classe è pubblico all'interno del suo package, ma non è accessibile al di fuori di esso (i package sono descritti nel prossimo capitolo).

Nelle classi sviluppate finora, tutti i membri di una classe hanno utilizzato il modo di accesso predefinito, cioè pubblico. Tuttavia, in generale non si desidera che questa situazione si verifichi: solitamente si preferisce limitare l'accesso ai dati membri di una classe, consentendo l'accesso solo attraverso metodi. Inoltre, si incontreranno situazioni in cui si desidererà definire metodi privati per una classe.

Uno specificatore di accesso precede la parte restante della specifica di un tipo di membro, cioè deve iniziare la dichiarazione di un membro. Ecco un esempio:

```
public int i;
private double j;
private int myMethod(int a, char b) { // ...
```

Per comprendere gli effetti dell'accesso pubblico e privato, si consideri il seguente programma:

```
/* Questo programma illustra la differenza tra public e private.
*/
class Test {
    int a; // accesso predefinito
    public int b; // accesso pubblico
    private int c; // accesso privato
    // metodi per accedere a c
    void setc(int i) { // imposta il valore di c
        c = i;
    }
    int getc() { // ottiene il valore di c
        return c;
    }
}
class AccessTest {
```

```

public static void main(String args[]) {
    Test ob = new Test();
    // Questi vanno bene, e' possibile accedere direttamente ad a e b
    ob.a = 10;
    ob.b = 20;
    // Questo non va bene e causera' un errore
    // ob.c = 100; // Errore!
    // Occorre accedere a c attraverso i suoi metodi
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
        ob.b + " " + ob.getc());
    }
}

```

All'interno della classe `Test`, `a` utilizza l'accesso predefinito, che per questo esempio equivale a specificare `public`; `b` è specificata esplicitamente come `public`; al membro `c` viene fornito l'accesso privato. Questo significa che non è accessibile da parte di codice esterno alla sua classe. Quindi, all'interno della classe `AccessTest`, `c` non può essere utilizzata direttamente: occorre accedere a essa attraverso i suoi metodi pubblici, `setc()` e `getc()`. Se si eliminasse il segno di commento posto all'inizio della riga seguente,

```
// ob.c = 100; // Errore!
```

non sarebbe possibile compilare il programma a causa della violazione di accesso.

Per osservare come può essere applicato il controllo di accesso a un esempio più pratico, si consideri la seguente versione migliorata della classe `Stack` mostrata alla fine del Capitolo 6.

```

// Questa classe definisce uno stack di interi che puo' contenere 10 valori.
class Stack {
    /* Ora, sia stck che tos sono privati. Questo significa che non
       possono essere modificati accidentalmente o intenzionalmente in maniera
       da danneggiare lo stack.
    */
    private int stck[] = new int[10];
    private int tos;
    // Inizializza la cima dello stack
    Stack() {
        tos = -1;
    }
    // Inserisce un elemento nello stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
    // Prende un elemento dallo stack
    int pop() {

```

```

    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
}

```

Ora sia `stck`, che contiene lo stack, sia `tos`, l'indice della cima dello stack, sono specificati come private. Questo significa che non è possibile accedervi o alterarli se non con `push()` e `pop()`. Rendendo `tos` privato, per esempio, si impedisce ad altre parti del programma di impostarlo inavvertitamente su un valore che vada oltre la fine dell'array `stck`.

Il programma che segue mostra la classe `Stack` migliorata. Provando a eliminare le righe commentate si noterà che i membri `stck` e `tos` sono effettivamente inaccessibili.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();
        // inserisce alcuni numeri nello stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);
        // prende quei numeri dallo stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
        // queste dichiarazioni non sono valide
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}

```

Sebbene generalmente i metodi forniscano accesso ai dati definiti da una classe, non deve sempre essere così. È assolutamente ammesso consentire a una variabile di istanza di essere pubblica quando ci sia un buon motivo per cui lo sia. Per esempio, per facilitare le cose la maggior parte delle classi semplici di questo libro è stata creata senza badare troppo a controllare l'accesso alle variabili di istanza; tuttavia nella maggior parte delle classi reali sarà necessario consentire operazioni sui dati solo attraverso metodi. Il prossimo capitolo tornerà sull'argomento del controllo di accesso che, come si vedrà, è di particolare importanza quando è coinvolta l'ereditarietà.

7.7 Comprendere static

Esistono situazioni in cui si desidera definire un membro di classe che sarà utilizzato indipendentemente da qualsiasi oggetto di tale classe. Normalmente, si accede a un membro di classe solo insieme agli oggetti della sua classe; tuttavia, è possibile creare un membro che può essere utilizzato in modo indipendente, senza riferimento a un'istanza specifica. Per creare un simile membro, occorre anteporre alla sua dichiarazione la parola chiave `static`. Quando un membro viene dichiarato come `static`, si può accedere a esso prima che venga creato qualsiasi oggetto della sua classe e senza riferimento ad alcun oggetto. È possibile dichiarare come `static` sia i metodi che le variabili. L'esempio più comune di un membro `static` è `main()`, che viene dichiarato `static` perché deve essere chiamato prima della creazione di qualsiasi oggetto.

Le variabili di istanza dichiarate come `static` sono, in sostanza, variabili globali. Quando vengono dichiarati gli oggetti della sua classe, non viene creata alcuna copia di una variabile `static`, ma tutte le istanze della classe condividono la stessa variabile `static`.

I metodi dichiarati come `static` hanno numerose limitazioni:

- possono chiamare solo altri metodi `static`;
- devono accedere solo a dati `static`;
- non possono in alcun modo fare riferimento a `this` o a `super` (la parola chiave `super` è relativa all'ereditarietà ed è descritta nel prossimo capitolo).

Se è necessario eseguire calcoli per inizializzare le variabili `static`, è possibile dichiarare un blocco `static` che viene eseguito una sola volta, quando la classe viene caricata. L'esempio che segue mostra una classe con un metodo `static`, alcune variabili `static`, e un blocco di inizializzazione `static`:

```
// Dimostra le variabili, i metodi e i blocchi static.
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

Non appena viene caricata la classe `UseStatic`, vengono eseguite tutte le dichiarazioni `static`. In primo luogo, a viene impostata su 3, quindi viene

eseguito il blocco static (stampando un messaggio); infine b viene inizializzata su $a * 4$ o 12. Successivamente, viene chiamato `main()`, che chiama `meth()` passando 42 a x. Le tre dichiarazioni `println()` si riferiscono alle due variabili static `a` e `b` e alla variabile locale `x`.

Di seguito è mostrato l'output del programma:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Esternamente alla classe in cui sono definiti, i metodi e le variabili static possono essere utilizzati indipendentemente da qualsiasi oggetto: basta specificare il nome della loro classe seguito dall'operatore punto. Per esempio, se si desidera chiamare un metodo static esternamente alla sua classe, è possibile utilizzare la seguente forma generale:

```
classname.method( )
```

Qui, *classname* è il nome della classe nella quale viene dichiarato il metodo static. Come si può osservare, questo formato è simile a quello utilizzato per chiamare metodi non static attraverso variabili che fanno riferimento a oggetti. Si può accedere a una variabile static allo stesso modo: con l'utilizzo dell'operatore punto sul nome della classe. Java implementa una versione controllata di metodi e di variabili globali in questo modo.

Di seguito è mostrato un esempio: all'interno di `main()` si accede al metodo static `callme()` e alla variabile static `b` esternamente alla loro classe.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;
    static void callme() {
        System.out.println("a = " + a);
    }
}
class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Ecco l'output di questo programma:

```
a = 42
b = 99
```

7.8 Introduzione a final

Una variabile può essere dichiarata come `final`, impedendo così la modifica del suo contenuto. Questo significa che è necessario inizializzare una variabile `final` quando viene dichiarata. Per esempio:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Le parti successive del programma possono ora utilizzare `FILE_OPEN` e le altre variabili come se fossero costanti, senza timore che un valore sia stato modificato.

La scelta di identificatori scritti tutti in lettere maiuscole per le variabili `final` è una convenzione diffusa. Le variabili dichiarate come `final` non occupano memoria su una base per istanza; di conseguenza, una variabile `final` è una costante.

La parola chiave `final` può anche essere applicata ai metodi, ma il suo significato è molto diverso da quello che ha quando viene applicata alle variabili. Il secondo utilizzo di `final` sarà descritto nel prossimo capitolo, quando si parlerà di ereditarietà.

7.9 Nuovo esame degli array

Gli array sono stati presentati in precedenza in questo libro, prima della descrizione sulle classi. Ora che si conoscono le classi, si può dire una cosa importante sugli array: sono implementati come oggetti. Per questo motivo esiste un particolare attributo degli array molto comodo: le dimensioni, cioè il numero di elementi che può contenere un array, si trovano nella sua variabile di istanza `length`. Tutti gli array dispongono di questa variabile che conterrà sempre le dimensioni dell'array. Di seguito è mostrato un programma che mostra tale proprietà:

```
// Questo programma illustra il membro di array length.
class Length {
    public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};
        System.out.println("length of a1 is " + a1.length);
        System.out.println("length of a2 is " + a2.length);
        System.out.println("length of a3 is " + a3.length);
    }
}
```

Il programma visualizza il seguente output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

Come si può notare, vengono visualizzate le dimensioni di ciascun array. Occorre ricordare che il valore di `length` non ha niente a che vedere con il numero di elementi effettivamente utilizzati, ma riflette solo il numero di elementi che l'array può contenere.

Il membro `length` può essere utile in molte situazioni. Per esempio, di seguito è mostrata una versione migliorata della classe `Stack`. Come si può ricordare, le versioni precedenti di questa classe hanno sempre creato uno stack di dieci elementi; la versione che segue consente di creare stack di qualsiasi dimensione: il valore di `stck.length` viene utilizzato per impedire allo stack di eccedere nelle dimensioni.

```
// Classe Stack migliorata che utilizza il membro di array length.
class Stack {
    private int stck[];
    private int tos;
    // alloca e inizializza lo stack
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }
    // Inserisce un elemento nello stack
    void push(int item) {
        if(tos==stck.length-1) // utilizza il membro length
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
    // Prende un elemento dallo stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);
        // aggiunge alcuni numeri allo stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);
        // prende quei numeri dallo stack
```

```

        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}

```

Si noti che il programma crea due stack: uno con cinque elementi e uno con otto. Il fatto che gli array mantengano le informazioni sulla loro lunghezza facilita la creazione di stack di qualsiasi dimensione.

7.10 Introduzione alle classi annidate e interne

È possibile definire una classe all'interno di un'altra classe; tali classi sono note come *classi annidate*. L'ambito di una classe annidata è limitato dall'ambito della classe che la contiene. Di conseguenza, se la classe B è definita all'interno della classe A, B è nota ad A, ma non al di fuori di A. Una classe annidata ha accesso ai membri della classe in cui si trova, compresi i membri privati. Tuttavia, la classe che la racchiude non ha accesso ai membri della classe annidata.

Esistono due tipi di classi annidate: *statiche* e *non statiche*. Una classe annidata statica è una classe alla quale viene applicato il modificatore `static`. Poiché è statica, deve accedere ai membri della classe che la contiene attraverso un oggetto; in altre parole non può fare direttamente riferimento ai membri della classe in cui è contenuta. A causa di questa limitazione, le classi annidate statiche vengono utilizzate raramente.

Il tipo più importante di classe annidata è la classe *interna*, una classe annidata non statica. Ha accesso a tutte le variabili e ai metodi della classe che la contiene e può fare riferimento direttamente a essi, nello stesso modo in cui lo fanno gli altri membri non statici della classe esterna. Di conseguenza, una classe interna si trova completamente all'interno dell'ambito della classe in cui è racchiusa.

Il programma che segue mostra come definire e utilizzare una classe interna. La classe chiamata `Outer` ha una variabile di istanza chiamata `outer_x`, un metodo di istanza chiamato `test()`, e definisce una classe interna chiamata `Inner`.

```

// Dimostra una classe interna.
class Outer {
    int outer_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }
}

```

```

// questa e' una classe interna
class Inner {
    void display() {
        System.out.println("display: outer_x = " + outer_x);
    }
}
}
class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}

```

L'output di questa applicazione è il seguente:

```
display: outer_x = 100
```

Nel programma, una classe interna chiamata `Inner` viene definita all'interno dell'ambito della classe `Outer`. Di conseguenza, tutto il codice della classe `Inner` può accedere direttamente alla variabile `outer_x`. Un metodo di istanza chiamato `display()` viene definito all'interno di `Inner`. Questo metodo visualizza `outer_x` nel flusso di output standard. Il metodo `main()` di `InnerClassDemo` crea un'istanza della classe `Outer` e chiama il suo metodo `test()`, il quale crea un'istanza della classe `Inner`; successivamente, viene chiamato il metodo `display()`.

È importante comprendere che la classe `Inner` è nota solo all'interno dell'ambito della classe `Outer`. Il compilatore Java genera un messaggio di errore se qualsiasi codice esterno alla classe `Outer` cerca di istanziare `Inner`. Generalizzando, una classe annidata non è diversa da qualsiasi altro elemento di programma: è nota solo all'interno dell'ambito che la contiene.

Come già spiegato, una classe interna ha accesso a tutti i membri della classe che la contiene, ma non accade il contrario. I membri della classe interna sono noti solo all'interno dell'ambito della classe interna e non possono essere utilizzati dalla classe esterna. Per esempio:

```

// Questo programma non verra' compilato.
class Outer {
    int outer_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }
}
// questa e' una classe interna
class Inner {
    int y = 10; // y e' locale per Inner
    void display() {
        System.out.println("display: outer_x = " + outer_x);
    }
}

```



```
display: outer_x = 100
display: outer_x = 100
```

Anche se le classi annidate non vengono utilizzate spesso, sono particolarmente utili nella gestione di eventi in un'applet. Si tornerà sull'argomento delle classi annidate nel Capitolo 22, dove si esaminerà come si possono utilizzare le classi interne per semplificare il codice necessario per la gestione di determinati tipi di eventi. Si conosceranno anche le *classi interne anonime*, cioè classi interne senza nome.

Una considerazione finale: le classi annidate non erano ammesse dalla specifica originale 1.0 di Java, ma sono state aggiunte a partire da Java 1.1.

7.11 Esplorare la classe String

Sebbene la classe `String` verrà esaminata in dettaglio nella Parte seconda di questo libro, qui sarà trattata brevemente perché alcuni programmi di esempio riportati verso la fine della Parte prima utilizzano le stringhe. Probabilmente `String` è la classe più utilizzata nella libreria di classi di Java. Il motivo ovvio è che le stringhe sono una parte molto importante della programmazione.

La prima cosa da comprendere sulle stringhe è che ogni stringa creata in realtà è un oggetto di tipo `String`. Anche le costanti di stringa sono oggetti `String`. Per esempio, nella dichiarazione:

```
System.out.println("This is a String, too");
```

la stringa `"This is a String, too"` è una costante `String`. Fortunatamente, Java gestisce le costanti `String` allo stesso modo in cui gli altri linguaggi gestiscono le "normali" stringhe, quindi non occorre preoccuparsi.

La seconda cosa da comprendere è che gli oggetti di tipo `String` sono immutabili: una volta creato un oggetto `String`, il suo contenuto non può essere alterato. Anche se questa potrebbe sembrare una limitazione seria, non lo è per due motivi:

- se è necessario modificare una stringa, se ne può sempre creare una nuova che contiene le modifiche;
- Java definisce una classe peer di `String`, chiamata `StringBuffer`, che consente di alterare le stringhe, quindi in Java sono ancora disponibili tutte le normali manipolazioni di stringhe (`StringBuffer` è descritta nella Parte seconda di questo libro).

Le stringhe possono essere costruite in diversi modi. Il più semplice è l'utilizzo di una dichiarazione come la seguente:

```
String myString = "this is a test";
```

Una volta creato un oggetto `String`, è possibile utilizzarlo dovunque sia consentita una stringa. Per esempio, questa dichiarazione visualizza `myString`:

```
System.out.println(myString);
```

Java definisce un operatore per gli oggetti `String`: `+`, utilizzato per concatenare due stringhe. Per esempio, questa dichiarazione:

```
String myString = "I" + " like " + "Java.";
```

ha come risultato `myString` che contiene "I like Java."

Il programma che segue mostra i concetti appena descritti:

```
// Dimostrazione di String.
class StringDemo {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1 + " and " + strOb2;
        System.out.println(strOb1);
        System.out.println(strOb2);
        System.out.println(strOb3);
    }
}
```

L'output prodotto da questo programma è mostrato di seguito:

```
First String
Second String
First String and Second String
```

La classe `String` contiene diversi metodi che possono essere utilizzati. È possibile confrontare due stringhe per verificarne l'uguaglianza utilizzando `equals()`; si può ottenere la lunghezza di una stringa chiamando il metodo `length()`, mentre si può ricavare il carattere in un indice specificato di una stringa chiamando `charAt()`. Le forme generali di questi tre metodi sono mostrate di seguito:

```
boolean equals(String object)
int length( )
char charAt(int index)
```

Ecco il programma che illustra questi metodi:

```
// Dimostrazione di alcuni metodi String.
class StringDemo2 {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;
        System.out.println("Length of strOb1: " +
```

```

        strObj1.length());
    System.out.println("Char at index 3 in strObj1: " +
        strObj1.charAt(3));
    if(strObj1.equals(strObj2))
    System.out.println("strObj1 == strObj2");
    else
    System.out.println("strObj1 != strObj2");
    if(strObj1.equals(strObj3))
    System.out.println("strObj1 == strObj3");
    else
    System.out.println("strObj1 != strObj3");
    }
}

```

Questo programma genera il seguente output:

```

Length of strObj1: 12
Char at index 3 in strObj1: s
strObj1 != strObj2
strObj1 == strObj3

```

Naturalmente, è possibile avere array di stringhe, come è possibile avere array di qualsiasi altro tipo di oggetto. Per esempio:

```

// Dimostra gli array String.
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "one", "two", "three" };
        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " +
                str[i]);
    }
}

```

Ecco l'output di questo programma:

```

str[0]: one
str[1]: two
str[2]: three

```

Come si vedrà nel prossimo paragrafo, gli array di stringhe giocano un ruolo importante in molti programmi Java.

7.12 Utilizzare gli argomenti della riga di comando

A volte si desidera passare informazioni a un programma quando lo si esegue. Questa operazione può essere eseguita passando a `main()` gli *argomenti della riga di comando*. Un argomento della riga di comando è l'informazione che segue direttamente il nome del programma sulla riga di comando quando

viene eseguito. Accedere agli argomenti della riga di comando all'interno di un programma Java è facile: sono memorizzati come stringhe nell'array `String` passato a `main()`. Per esempio, il programma che segue mostra tutti gli argomenti della riga di comando con cui viene chiamato:

```
// Visualizza tutti gli argomenti della riga di comando.
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "]: " +
                args[i]);
    }
}
```

Provare a eseguire il programma, come mostrato di seguito:

```
java CommandLine this is a test 100 -1
```

Quando eseguito, apparirà il seguente output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

ATTENZIONE *Tutti gli argomenti della riga di comando vengono passati come stringhe. È necessario convertire manualmente i valori numerici nelle loro forme interne, come spiegato nel Capitolo 16.*

7.13 Varargs: argomenti a lunghezza variabile

J2SE 5 ha inserito una nuova funzione che semplifica la creazione di metodi che devono utilizzare un numero variabile di argomenti. Questa funzione viene chiamata `varargs`, che è l'abbreviazione di *argomenti a lunghezza variabile*. Un metodo che utilizza un numero variabile di argomenti viene chiamato *metodo varargs*.

Le situazioni in cui si richiede che a un metodo venga passato un numero variabile di argomenti non sono insolite. Per esempio, un metodo che apre una connessione Internet può utilizzare un nome utente, una password, un nome di file, un protocollo e così via, ma inserire impostazioni predefinite se alcune informazioni non vengono fornite. In questa situazione, sarebbe pratico passare solo gli argomenti ai quali non vengono applicate le impostazioni predefinite. Un altro esempio è il nuovo metodo `printf()` che appartiene alla libreria di I/O di Java. Come si vedrà nel Capitolo 19, esso utilizza un numero variabile di argomenti, che formatta e poi mostra come risultato.

Prima di J2SE 5, gli argomenti a lunghezza variabile potevano essere gestiti in due modi, nessuno dei quali era particolarmente piacevole. Innanzitutto, se il numero massimo di argomenti era piccolo e noto, si potevano creare versioni sovraccaricate del metodo, una per ciascun modo in cui era possibile chiamare il metodo. Sebbene questa operazione funzioni ed è adatta in alcune situazioni, si applica solo a una classe di circostanze molto limitata.

Nei casi in cui il numero massimo di argomenti potenziali era più grande, o sconosciuto, veniva utilizzato un secondo approccio in cui gli argomenti venivano inseriti in un array e poi l'array veniva passato al metodo. Questo approccio è illustrato nel seguente programma:

```
// Utilizza un array per passare un numero variabile di
// argomenti a un metodo. Questo e' l'approccio
// vecchio stile agli argomenti a lunghezza variabile.
class PassArray {
    static void vaTest(int v[]) {
        System.out.print("Number of args: " + v.length +
            " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        // Notare come deve essere creato un array
        // per contenere gli argomenti.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };
        vaTest(n1); // 1 argomento
        vaTest(n2); // 3 argomenti
        vaTest(n3); // nessun argomento
    }
}
```

L'output del programma è mostrato di seguito:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Nel programma, al metodo `vaTest()` vengono passati i suoi argomenti attraverso l'array `v`. Questo approccio vecchio stile agli argomenti a lunghezza variabile consente a `vaTest()` di utilizzare un numero arbitrario di argomenti. Tuttavia, richiede che questi argomenti vengano inseriti manualmente in un array prima di chiamare `vaTest()`. Non solo è noioso costruire un array ogni volta che viene chiamato `vaTest()`, ma è potenzialmente causa di errori. La nuova funzione offre un'opzione migliore e più semplice.

Un argomento a lunghezza variabile viene specificato da tre punti (...). Per esempio, ecco come viene scritto `vaTest()` utilizzando un `vararg`:

```
static void vaTest(int ... v) {
```

Questa sintassi indica al compilatore che `vaTest()` può essere chiamato con zero o più argomenti. Di conseguenza, `v` viene implicitamente dichiarato come array di tipo `int[]`. Per questo motivo, all'interno di `vaTest()` si accede a `v` attraverso la normale sintassi di array. Di seguito è riportato il programma precedente riscritto utilizzando un `vararg`:

```
// Dimostra gli argomenti a lunghezza variabile.
class VarArgs {
    // ora vaTest() utilizza un vararg.
    static void vaTest(int ... v) {
        System.out.print("Number of args: " + v.length +
            " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        // Si noti come vaTest() puo' essere chiamato con un
        // numero variabile di argomenti.
        vaTest(10); // 1 argomento
        vaTest(1, 2, 3); // 3 argomenti
        vaTest(); // nessun argomento
    }
}
```

L'output del programma è lo stesso di quello della versione originale.

Sono due le cose importanti da notare in questo programma: innanzitutto, come già spiegato, all'interno di `vaTest()` `v` viene trattato come un array. Questo perché `v` è un array. La sintassi `...` indica semplicemente al compilatore che verrà utilizzato un numero variabile di argomenti, e che tali argomenti verranno memorizzati nell'array a cui fa riferimento `v`. Inoltre, in `main()` `vaTest()` viene chiamato con numeri diversi di argomenti, compreso nessun argomento. Gli argomenti vengono automaticamente inseriti in un array e passati a `v`. Nel caso di nessun argomento, la lunghezza dell'array è zero.

Un metodo può disporre di parametri "normali" e di un parametro a lunghezza variabile. Tuttavia, il parametro a lunghezza variabile deve essere l'ultimo dichiarato dal metodo. Per esempio, questa dichiarazione di metodo è perfettamente valida:

```
int doIt(int a, int b, double c, int ... vals) {
```

In questo caso, i primi tre argomenti utilizzati in una chiamata a `doIt()` vengono accoppiati ai primi tre parametri. Quindi, si suppone che qualsiasi argomento rimanente appartenga a `vals`.

Occorre ricordare che il parametro `varargs` deve essere l'ultimo. Per esempio, la seguente dichiarazione non è corretta:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Errore!
```

Qui, si è tentato di dichiarare un parametro regolare dopo il parametro `varargs`, che non è valido.

Occorre essere consapevoli di un'ulteriore limitazione: ci deve essere un solo parametro `varargs`.

Per esempio, anche questa dichiarazione non è valida:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Errore!
```

Il tentativo di dichiarare il secondo parametro `varargs` non è valido.

Di seguito è riportata una versione rielaborata del metodo `vaTest()` che utilizza un argomento regolare e uno a lunghezza variabile:

```
// Utilizza varargs con argomenti standard.
class VarArgs2 {
    // Qui, msg e' un parametro normale e v e' un
    // parametro varargs.
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v.length +
            " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        vaTest("One vararg: ", 10);
        vaTest("Three varargs: ", 1, 2, 3);
        vaTest("No varargs: ");
    }
}
```

L'output di questo programma è il seguente:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

Sovraccaricare i metodi `vararg`

È possibile sovraccaricare un metodo che utilizza un argomento a lunghezza variabile. Per esempio, il programma che segue sovraccarica `vaTest()` tre volte:

```
// Varargs e sovraccarico.
class VarArgs3 {
    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
            "Number of args: " + v.length +
```

```
        " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...) " +
            "Number of args: " + v.length +
            " Contents: ");
        for(boolean x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    static void vaTest(String msg, int ... v) {
        System.out.print("vaTest(String, int ...): " +
            msg + v.length +
            " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        vaTest(1, 2, 3);
        vaTest("Testing: ", 10, 20);
        vaTest(true, false, false);
    }
}
```

L'output prodotto da questo programma è mostrato di seguito:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false
```

Questo programma illustra entrambi i modi in cui può essere sovraccaricato un metodo varargs. Innanzitutto, i tipi del suo parametro vararg possono variare. Questo è il caso di `vaTest(int ...)` e `vaTest(boolean ...)`. Occorre ricordare che `...` fa in modo che il parametro venga trattato come un array del tipo specificato.

Di conseguenza, proprio come si possono sovraccaricare i metodi utilizzando diversi tipi di parametri di array, è possibile sovraccaricare i metodi vararg impiegando tipi diversi di varargs. In questo caso, Java utilizza la differenza di tipo per stabilire quale metodo sovraccaricato chiamare.

Il secondo modo per sovraccaricare un metodo varargs consiste nell'aggiungere un parametro normale. Questo è ciò che viene realizzato con `vaTest(String, int ...)`. In questo caso, per stabilire quale metodo chiamare Java utilizza sia il numero di argomenti che il loro tipo.

Varargs e ambiguità

Quando si sovraccarica un metodo che utilizza un argomento a lunghezza variabile si possono verificare errori inaspettati. Tali errori riguardano l'ambiguità perché è possibile creare una chiamata ambigua a un metodo varargs sovraccaricato. Per esempio, si consideri il seguente programma:

```
// Varargs, sovraccarico e ambiguità'.
//
// Questo programma contiene un errore e non
// verrà compilato!
class VarArgs4 {
    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
            "Number of args: " + v.length +
            " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...) " +
            "Number of args: " + v.length +
            " Contents: ");
        for(boolean x : v)
            System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        vaTest(1, 2, 3); // OK
        vaTest(true, false, false); // OK
        vaTest(); // Errore: Ambiguo!
    }
}
```

In questo programma il sovraccarico di `vaTest()` è assolutamente corretto. Tuttavia, il programma non verrà compilato a causa della seguente chiamata:

```
vaTest(); // Errore: Ambiguo!
```

Poiché il parametro `vararg` può essere vuoto, questa chiamata potrebbe essere trasformata in una chiamata a `vaTest(int ...)` o a `vaTest(boolean ...)`. Entrambe sono ugualmente valide. Di conseguenza, la chiamata è sostanzialmente ambigua.

Di seguito è riportato un altro esempio di ambiguità: le seguenti versioni sovraccaricate di `vaTest()` sono sostanzialmente ambigue anche se una utilizza un parametro normale:

```
static void vaTest(int ... v) { // ...  
static void vaTest(int n, int ... v) { // ...
```

Sebbene gli elenchi dei parametri di `vaTest()` siano diversi, non esiste un modo attraverso il quale il compilatore possa eseguire la seguente chiamata:

```
vaTest(1)
```

Questa operazione si trasforma in una chiamata a `vaTest(int ...)` con un argomento `varargs`, o in una chiamata a `vaTest(int, int ...)` senza argomenti `varargs`? Non esiste un modo con cui il compilatore possa rispondere a questa domanda. Di conseguenza, la situazione è ambigua.

A causa di errori di ambiguità come quelli appena mostrati, sarà necessario evitare il sovraccarico e utilizzare semplicemente due nomi diversi per i metodi. Inoltre, in alcuni casi gli errori di ambiguità mostrano una falla concettuale nel codice, alla quale si può rimediare studiando una soluzione ponderata.