

# Ogni cosa è un oggetto

Sebbene sia basato sul C++, Java è più di un linguaggio “puro” orientato agli oggetti.

Sia C++ sia Java sono linguaggi ibridi, ma in Java i progettisti si sono accorti che l’ibridazione non era altrettanto importante che in C++. Un linguaggio ibrido permette molteplici stili di programmazione; la ragione per cui C++ è ibrido è quella di garantire la compatibilità a ritroso con il linguaggio C.

Poiché è un’estensione del C, il C++ comprende molte delle caratteristiche indesiderabili di quel linguaggio, il che può rendere alcuni aspetti del C++ eccessivamente complicati. Il linguaggio Java presuppone che vogliate fare solo programmazione orientata agli oggetti. Ciò significa che, prima di cominciare, dovete spostarvi mentalmente in un mondo orientato agli oggetti (a meno che non l’abbiate già fatto).

Questo sforzo iniziale vi permette di programmare in un linguaggio più semplice da imparare e da usare di molti altri linguaggi di programmazione orientata agli oggetti. In questo capitolo vedremo i componenti fondamentali di un programma Java e impareremo che in Java ogni cosa è un oggetto, perfino un programma Java.

## Gli oggetti si manipolano con i riferimenti

Ciascun linguaggio di programmazione ha i propri strumenti di manipolazione dei dati. Talvolta il programmatore deve essere costantemente consapevole di quale tipo di manipolazione si tratta. State manipolando l’oggetto direttamente o avete a che fare con qualche tipo di rappresentazione indiretta (un puntatore in C o in C++), che deve essere trattata con una sintassi speciale?

Tutto ciò è semplificato in Java. Voi trattate tutto come un oggetto, per cui c’è una sola sintassi coerente che usate dappertutto. Sebbene *trattiate* ogni cosa come un oggetto, l’identificatore che manipolate è in effetti un “riferimento” a un oggetto<sup>1</sup>. Potete imma-

---

<sup>1</sup> Questo concetto può scatenare una rissa. C’è chi dice “giusto, è un puntatore”, ma in questo modo si dà per scontata un’implementazione sottostante. Inoltre, i riferimenti Java, quanto a sintassi, sono molto più affini ai riferimenti C++ dei puntatori. Nella prima edizione di questo libro, avevo deciso di inventare un nuovo termine, “handle”, o “appiglio”, perché fra i riferimenti C++ e quelli Java vi sono differenze importanti. In quel periodo stavo giusto venendo fuori dal C++ e non volevo creare confusioni nelle menti dei programmatori C++ che, secondo me, avrebbero costituito il pubblico più ampio per Java. Nella seconda edizione decisi che “riferimento” era il termine più utilizzato e diffuso e che chiunque provenisse dal C++ avrebbe ben altro di cui occuparsi che non la terminologia dei riferimenti, quindi tanto valeva che affrontassero l’argomento buttandosi a capofitto. Vi sono, però, persone che non sono per niente

ginare questa situazione come un televisore (l'oggetto) con il telecomando (il riferimento). Finché tenete in pugno questo riferimento, avete un collegamento con il televisore, ma quando qualcuno dice "cambia canale" o "abbassa il volume", ciò che state manipolando è il riferimento, che a sua volta modifica l'oggetto. Se volete spostarvi nella stanza continuando a controllare il televisore, dovete prendere con voi il telecomando/riferimento, non il televisore.

Inoltre, il telecomando può stare da solo, senza televisore. Cioè, il solo fatto di avere un riferimento non significa che a questo sia necessariamente collegato un oggetto. Così, se volete conservare una parola o una frase, create un riferimento **String**:

```
String s;
```

Ma in questo caso avete creato *solo* il riferimento, non un oggetto. Se a questo punto decidete di inviare un messaggio a `s`, avrete un errore (al momento dell'esecuzione), perché `s` non è effettivamente collegato a niente (non c'è il televisore). Un metodo più sicuro, quindi, è quello di inizializzare sempre un riferimento quando lo create:

```
String s = "asdf";
```

Tuttavia, in questo caso utilizzate una speciale caratteristica Java, per cui le stringhe possono essere inizializzate con un testo tra virgolette. Normalmente, dovete usare un tipo di inizializzazione più generale per gli oggetti.

## Bisogna creare tutti gli oggetti

Quando create un riferimento, dovete collegarlo con un nuovo oggetto. Questo si fa, in generale, con la parola chiave **new**. **new** dice: "Fammi un nuovo oggetto". Così, nell'esempio precedente potete dire:

```
String s = new String ("asdf");
```

In questo caso non solo si dice "Fammi una nuova **String**", ma si danno anche informazioni su *come* fare la **String** fornendo una stringa di caratteri iniziale.

Naturalmente, **String** non è il solo tipo esistente. Java possiede numerosi tipi già definiti. Ancora più importante è il fatto che potete creare i vostri tipi. Infatti, questa è l'attività fondamentale della programmazione Java ed è ciò che imparerete nel resto di questo libro.

## Dove si trovano gli spazi di immagazzinamento

È utile visualizzare alcuni aspetti di come stanno le cose durante l'esecuzione del programma, in particolare come viene sistemata la memoria. I dati vengono immagazzinati in sei luoghi diversi:

1. **I registri**. Questa è l'area di immagazzinamento più veloce, perché si trova in un posto diverso da quello degli altri: all'interno del processore. Tuttavia, il numero dei re-

---

d'accordo neppure col termine "riferimento". Ho letto in un libro che era "completamente sbagliato dire che Java supporta il passaggio per riferimento", perché gli identificatori degli oggetti Java (secondo quell'autore) sono *in effetti* "riferimenti a oggetti". E (così proseguiva) tutto viene *in effetti* passato per valore. Quindi non si passa per riferimento, ma si "passa un riferimento a un oggetto mediante un valore". Ci sarebbe da discutere sulla precisione di spiegazioni così contorte, ma sono convinto che il mio approccio semplifichi la comprensione del concetto senza urtare la sensibilità di alcuno (beh, sì, qualche fondamentalista del linguaggio potrebbe sostenere che vi sto mentendo, ma mi difenderò dichiarando che sto fornendo un'appropriata astrazione).

- gistri è molto limitato, per cui vengono assegnati dal compilatore secondo le sue necessità. Non potete controllarli direttamente e nei vostri programmi non c'è alcuna traccia dell'esistenza dei registri.
2. **Lo stack.** Questo risiede nell'area generale della RAM (memoria ad accesso casuale), ma ha un supporto diretto dal processore attraverso il *puntatore dello stack*. Il puntatore dello stack viene spostato verso il basso per creare nuova memoria e verso l'alto per liberarla. Questo è un modo estremamente rapido e efficace per assegnare memoria, superato solo dai registri. Il compilatore Java deve conoscere, mentre sta creando il programma, l'esatta dimensione e la durata in vita di tutti i dati che sono immagazzinati sullo stack, perché deve generare il codice che serve a spostare il puntatore dello stack in su e in giù. Questa forzatura pone limiti alla flessibilità dei programmi, per cui mentre un po' di spazio di immagazzinamento Java è presente sullo stack – in particolare, i riferimenti agli oggetti – gli oggetti Java stessi non sono disposti sullo stack.
  3. **Lo heap.** Questa è un serbatoio di immagazzinamento di uso generale (anch'esso nell'area della RAM), dove si trovano tutti gli oggetti Java. Il bello dello heap è che, a differenza dello stack, il compilatore non ha bisogno di sapere quanto spazio di immagazzinamento gli serve assegnare dallo heap o per quanto tempo quello spazio deve stare sullo heap. Così, c'è grande flessibilità nell'uso dello spazio sullo heap. Tutte le volte che dovete creare un oggetto, semplicemente scrivete il codice per crearlo usando **new** e viene assegnato uno spazio adeguato sullo heap quando il codice viene eseguito. Naturalmente c'è un prezzo da pagare per questa flessibilità: occorre più tempo per assegnare spazio dello heap che per assegnare spazio dello stack (cioè, anche se potete creare oggetti sullo stack in Java, come potete farlo in C++).
  4. **Lo spazio di immagazzinamento statico.** “Statico” si usa qui nel senso di “in un posto fisso” (sebbene anche questo sia nella RAM). Lo spazio di immagazzinamento statico contiene dati che sono disponibili per tutto il tempo di esecuzione del programma. Potete usare la parola chiave **static** per specificare che un particolare elemento di un oggetto è statico, ma gli oggetti Java veri e propri non vengono mai collocati nello spazio di immagazzinamento statico.
  5. **Lo spazio di immagazzinamento costante.** I valori costanti vengono spesso messi direttamente nel codice di programma; un posto sicuro, perché non cambiano mai. Talvolta le costanti stanno per conto loro, per cui possono essere facoltativamente messe nella memoria di sola lettura (ROM).
  6. **Spazio di immagazzinamento non RAM.** Se i dati si trovano completamente fuori da un programma, possono esistere, mentre il programma non è in esecuzione, fuori dal controllo del programma stesso. I due principali esempi di questo fatto sono gli *oggetti streamed*, in cui gli oggetti vengono convertiti in flussi (stream) di byte, generalmente per essere inviati a un'altra macchina, e *oggetti persistenti*, in cui gli oggetti vengono messi su un disco, cosicché possono mantenere il loro stato anche quando il programma è terminato. Il trucco, con questi tipi di spazi di immagazzinamento, consiste nel convertire gli oggetti in qualcosa che può stare sull'altro supporto e tuttavia può essere riconvertito, quando necessario, in un normale oggetto basato su RAM. Java supporta una *persistenza leggera* e le sue future versioni potrebbero fornire soluzioni più complete per quanto riguarda la persistenza.

### Caso speciale: tipi primitivi

C'è un gruppo di tipi cui viene riservato un trattamento speciale; potete pensarli come tipi “primitivi” che sono usati molto spesso nella programmazione. La ragione di questo trattamento speciale sta nel fatto che creare un oggetto con **new** – specialmente una pic-

cola semplice variabile – non è molto efficiente, perché **new** pone oggetti sullo heap. Per questi tipi Java ricade nell’approccio adottato da C e C++. Cioè, invece di creare la variabile usando **new**, crea una variabile “automatica” che *non è un riferimento*. La variabile conserva il valore e viene posta sullo stack, per cui è molto più efficiente. Java determina la dimensione di ciascun tipo primitivo, che non cambia passando da un’architettura di macchina all’altra come accade nella maggior parte dei linguaggi. Questa invariabilità della dimensione è una delle ragioni per le quali i programmi Java sono così portabili.

Tipo primitivo	Dimensione	Minima	Massima	Tipo wrapper
boolean	—	—	—	Boolean
char	16 bit	Unicode 0	Unicode 2 <sup>16</sup> -1	Character
byte	8 bit	-128	+127	Byte
short	16 bit	-2 <sup>15</sup>	+2 <sup>15</sup> -1	Short
int	32 bit	-2 <sup>31</sup>	+2 <sup>31</sup> -1	Integer
long	64 bit	-2 <sup>63</sup>	+2 <sup>63</sup> -1	Long
float	32 bit	IEEE754	IEEE754	Float
double	64 bit	IEEE754	IEEE754	Double
void	—	—	—	Void

Tutti i tipi numerici hanno il segno, per cui è inutile cercare tipi senza segno. La dimensione del tipo **boolean** non è esplicitamente definita; è specificato solo che può assumere i valori letterali **true** o **false**.

I tipi di dati primitivi hanno anche classi “wrapper”. Ciò significa che, se volete far sì che un oggetto non primitivo sullo heap rappresenti quel tipo primitivo, dovete usare il wrapper associato. Per esempio:

```
char c = 'x';
Character C = new Character(c);
```

Oppure potreste anche usare:

```
Character C = new Character('x');
```

La ragione di ciò sarà spiegata in un capitolo successivo.

### Numeri ad alta precisione

Java contiene due classi per eseguire aritmetica di alta precisione: **BigInteger** e **BigDecimal**. Sebbene queste rientrino all’incirca nella stessa categoria delle classi “wrapper”, né l’una né l’altra hanno un analogo primitivo.

Entrambe le classi hanno metodi che forniscono analoghi per le operazioni che si eseguono sui tipi primitivi. Cioè, con **BigInteger** o **BigDecimal** potete fare tutto ciò che potete fare con **int** o **float**; l’unica differenza è che dovete usare chiamate di metodo invece che operatori. Inoltre, poiché ci sono più implicazioni, le operazioni saranno più lente: si scambia la velocità con la precisione.

**BigInteger** supporta interi di precisione arbitraria. Ciò significa che potete rappresentare con precisione valori interi di qualsiasi dimensione senza perdere informazioni durante le operazioni.

**BigDecimal** è usato per i numeri in virgola fissa di precisione arbitraria; questi possono servire, per esempio, per calcoli monetari accurati. Per i particolari sui costruttori e i metodi che potete chiamare per queste due classi, consultate la documentazione in linea.

## Array in Java

Praticamente tutti i linguaggi di programmazione usano le matrici o array. L'uso degli array in C e in C++ è a rischio, perché quegli array sono solo blocchi di memoria. Se un programma accede all'array fuori dal suo blocco di memoria o usa la memoria prima dell'inizializzazione (comuni errori di programmazione), si possono avere risultati imprevedibili.

Uno dei principali obiettivi di Java è la sicurezza, per cui molti dei problemi che affliggono i programmatori in C e C++ non si ritrovano in Java. In Java è sicuro che un array è inizializzato e che non vi si può accedere fuori dal suo intervallo. Il controllo dell'intervallo è fatto al prezzo di avere un piccolo sovraccarico di memoria su ciascun array e di verificare l'indice in fase di esecuzione; ma il presupposto è quello di essere ripagati in termini di sicurezza e aumento di produttività.

Quando create un array di oggetti, in realtà create un array di riferimenti e ciascuno di questi riferimenti viene automaticamente inizializzato su un valore speciale con la propria parola chiave **null**. Quando Java vede **null**, riconosce che il riferimento in questione non punta a un oggetto. Dovete assegnare un oggetto a ciascun riferimento prima di usarlo e se cercate di usare un riferimento che è ancora null, il problema si manifesterà al momento dell'esecuzione. In questo modo si evitano in Java gli errori tipici che si generano con gli array.

Potete anche creare un array di primitivi. Anche in questo caso, il compilatore garantisce l'inizializzazione azzerando la memoria per quell'array.

Gli array saranno trattati in dettaglio nei prossimi capitoli.

## Non è mai necessario distruggere un oggetto

Nella maggior parte dei linguaggi di programmazione, il concetto di durata in vita di una variabile occupa una parte significativa dell'impegno di programmazione. Quanto dura la variabile? Se è necessario distruggerla, quando dovrete farlo? Una confusione circa la durata in vita delle variabili può portare a un sacco di errori e questo paragrafo mostra come Java semplifica notevolmente il problema eseguendo il lavoro di pulizia al posto vostro.

## Ambito di visibilità

La maggior parte dei linguaggi procedurali ha il concetto di *ambito di visibilità* o *portata*. Questo determina sia la visibilità sia la durata in vita dei nomi definiti entro quell'ambito di visibilità. In C, C++ e Java, l'ambito di visibilità è determinato dal posizionamento delle parentesi graffe {}. Così per esempio:

```
{
  int x = 12;
  /* solo x disponibile */
  {
    int q = 96;
    /* x e q entrambi disponibili */
  }
  /* solo x disponibile */
  /* q "fuori portata" */
}
```

Una variabile definita entro una portata è disponibile solo per tutta quella portata.

I rientri delle righe facilitano la lettura del codice Java. Poiché Java è un linguaggio a forma libera, gli spazi in più, le tabulazioni e gli a capo non influenzano il programma risultante.

Notate che *non potete* fare quanto segue, anche se è lecito farlo in C e in C++:

```
{
    int x = 12;
    {
        int x = 96; /* non consentito */
    }
}
```

Il compilatore avvertirà che la variabile **x** è già stata definita. Così la capacità del C e del C++ di “nascondere” una variabile in un ambito più ampio non è permessa perché i progettisti di Java hanno pensato che ciò avrebbe portato a programmi poco chiari.

## Portata degli oggetti

Gli oggetti Java non hanno la stessa durata in vita dei primitivi. Quando create un oggetto Java usando **new**, questo rimane disponibile dopo la fine della sua portata. Così, se usate:

```
{
    String s = new String("a string");
} /* fine della portata */
```

il riferimento **s** scompare alla fine della portata. Tuttavia, l'oggetto **String** al quale **s** puntava sta ancora occupando memoria. In questo pezzo di codice, non c'è modo di accedere all'oggetto, perché il solo riferimento ad esso è fuori della portata. Nei prossimi capitoli vedrete come il riferimento all'oggetto possa essere trasferito e duplicato durante il corso di un programma.

Di conseguenza, poiché gli oggetti creati con **new** rimangono a disposizione per tutto il tempo che volete, un'intera serie di problemi di programmazione del C++ in Java semplicemente scompare. I problemi più ardui tendono a presentarsi in C++ perché il linguaggio non dà alcun aiuto per garantire che gli oggetti siano disponibili quando sono richiesti. E, ancora più importante, in C++ dovete assicurarvi di aver distrutto gli oggetti quando avete finito di usarli.

Questo solleva un interessante problema. Se Java lascia gli oggetti sparsi qua e là, cosa impedisce loro di riempire la memoria e bloccare il programma? Questo è esattamente il tipo di problema che si presenta in C++. Ed è qui che compare una piccola magia. Java ha un *raccoglitore di rifiuti* che esamina tutti gli oggetti creati con **new** e determina quali sono quelli cui non si fa più riferimento. Quindi libera la memoria di quegli oggetti, in modo che possa essere usata per nuovi oggetti. Ciò significa che non dovete più preoccuparvi di liberare memoria di vostra iniziativa. Non dovete fare altro che creare gli oggetti e, quando non ne avrete più bisogno, se ne andranno da soli. Ciò elimina un'intera classe di problema di programmazione: la cosiddetta “perdita di memoria”, che si verifica quando un programmatore si dimentica di liberare memoria.

## Creare nuovi tipi di dati: le classi

Se ogni cosa è un oggetto, come si determina l'aspetto e il comportamento di una particolare classe di oggetti? In altre parole, che cosa stabilisce il *tipo* di un oggetto? Potreste pensare che ci sia una parola chiave chiamata “tipo” e ciò sarebbe certamente sensato. Sto-

ricamente, tuttavia, la maggior parte dei linguaggi orientati agli oggetti ha usato la parola chiave **class** con il significato di “Sto per dirvi come si presenta un nuovo tipo di oggetto”. La parola chiave **class** (che è così comune che non sarà scritta in grassetto in tutto il resto del libro) è seguita dal nome del nuovo tipo. Per esempio:

```
class ATypeName { /* il corpo della classe si inserisce qui */ }
```

Questo introduce un nuovo tipo, per cui ora potete creare un oggetto di questo tipo usando **new**:

```
ATypeName a = new ATypeName();
```

In **ATypeName**, il corpo della classe consiste solo di un commento (gli asterischi e le barre e ciò che si trova all'interno, che vedremo più avanti in questo capitolo), per cui non è che possiate farvene un granché. Infatti, non potete dirgli di fare molto (cioè, non potete inviargli alcun messaggio interessante), finché non definite qualche metodo per farlo.

## Campi e metodi

Quando definite una classe (e tutto ciò che fate in Java è definire classi, fare oggetti di queste classi e mandare messaggi a questi oggetti), potete mettere due tipi di elementi nella classe: elementi di dati (talvolta chiamati *campi*) e funzioni membro (normalmente chiamate *metodi*). Un elemento di dati è un oggetto di qualunque tipo con il quale potete comunicare attraverso il suo riferimento. Può essere anche uno dei tipi primitivi (che non sia un riferimento). Se è un riferimento a un oggetto, dovete inizializzare quel riferimento per collegarlo a un oggetto effettivo (usando **new**, come si è visto in precedenza) in una funzione speciale chiamata *costruttore* (descritto esaurientemente nel Capitolo 4). Se è un tipo primitivo potete iniziarlo direttamente al momento della definizione nella classe. (Come vedrete più avanti, i riferimenti possono essere inizializzati anche al momento della definizione.)

Ciascun oggetto mantiene il proprio spazio di immagazzinamento per i suoi elementi di dati; gli elementi di dati non vengono divisi fra gli oggetti. Ecco un esempio di una classe con alcuni elementi di dati:

```
class DataOnly {  
    int i;  
    float f;  
    boolean b;  
}
```

Questa classe non *fa* niente, ma potete creare un oggetto:

```
DataOnly d = new DataOnly();
```

Potete assegnare valori agli elementi di dati, ma dovete prima sapere come ci si riferisce a un elemento (membro) di un oggetto. Ciò viene fatto dichiarando il nome del riferimento all'oggetto, seguito da un punto, seguito dal nome dell'elemento dentro l'oggetto:

```
RiferimentoOggetto.elemento
```

Per esempio:

```
d.i = 47;  
d.f = 1.1f;  
d.b = false;
```

Può anche darsi che l'oggetto contenga altri oggetti che contengono dati che vorreste modificare. Per fare ciò, basta continuare “a collegare i punti”. Per esempio:

```
myPlane.leftTank.capacity = 100;
```

La classe **DataOnly** non può fare altro che contenere dati, perché non ha funzioni membro (metodi). Per capire come questi funzionano, dovete prima capire che cosa sono gli *argomenti* e i *valori restituiti*, che saranno descritti fra poco.

## Valori predefiniti per membri primitivi

Quando un tipo di dati primitivo è membro di una classe, se non lo inizializzate assume sicuramente un valore di default:

Tipo primitivo	Default
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Notate bene che i valori predefiniti sono quelli che Java garantisce quando la variabile viene usata *come membro di una classe*. Ciò assicura che le variabili membro di tipi primitivi saranno sempre inizializzate (cosa che il C++ non fa), riducendo una fonte di errori. Tuttavia, questo valore iniziale può non essere corretto o addirittura lecito per il programma che state scrivendo. È meglio inizializzare sempre esplicitamente le variabili.

Questa garanzia non si applica alle variabili “locali”: quelle che non sono campi di una classe. Così, se all'interno della definizione di una funzione avete:

```
int x;
```

allora **x** riceverà un valore arbitrario (come in C e in C++); non sarà automaticamente inizializzata a zero. Avete la responsabilità di assegnare un valore appropriato prima di usare **x**. Se lo dimenticate, Java è decisamente più un passo avanti rispetto al C++: otterrete un errore in fase di compilazione che vi dirà che la variabile potrebbe non essere stata inizializzata. (Molti compilatori C++ vi avvertiranno che ci sono variabili non inizializzate, ma in Java questi sono errori.)

## Metodi, argomenti e valori restituiti

Finora, il termine *funzione* è stato usato per descrivere una subroutine identificata da nome. Il termine più comunemente usato in Java è *metodo*, inteso come “un modo per fare qualcosa”. Se volete, potete continuare a pensare in termini di funzioni. In realtà, è solo una differenza sintattica, ma d'ora in avanti in questo libro si userà la parola “metodo” anziché “funzione”.

In Java i metodi determinano i messaggi che un oggetto può ricevere. In questo paragrafo imparerete quanto sia semplice definire un metodo.

Le parti fondamentali di un metodo sono il nome, gli argomenti, il tipo restituito e il corpo. La forma base è la seguente:

```
tipo_restituito nome_metodo( /* elenco argomenti */ ) {
    /* Corpo del metodo */
}
```

Il tipo restituito è il tipo del valore che salta fuori dal metodo dopo che l'avete chiamato. La lista degli argomenti fornisce i tipi e i nomi per le informazioni che volete passare al metodo. Il nome del metodo e la lista degli argomenti insieme identificano il metodo in modo univoco.

I metodi in Java possono essere creati solo come parte di una classe. Un metodo può essere chiamato solo per un oggetto<sup>2</sup> e quell'oggetto deve essere in grado di eseguire quella chiamata di metodo. Se cercate di chiamare il metodo sbagliato per un oggetto, otterrete un messaggio di errore in fase di compilazione. Si chiama un metodo per un oggetto nominando l'oggetto seguito da un punto, seguito dal nome del metodo e dalla sua lista di argomenti, così: **objectName.methodName (arg1, arg2, arg3)**. Per esempio, supponete di avere un metodo **f()** che non prende nessun argomento e restituisce un valore di tipo **int**. Allora, se avete un oggetto chiamato **a** per il quale si può chiamare **f()**, potete scrivere:

```
int x = a.f();
```

Il tipo di valore restituito deve essere compatibile con il tipo di **x**.

Questo atto di chiamare un metodo viene comunemente detto *inviare un messaggio a un oggetto*. Nell'esempio precedente, il messaggio è **f()** e l'oggetto è **a**. La programmazione orientata agli oggetti è spesso riassunta nella semplice frase "inviare messaggi a oggetti".

## L'elenco degli argomenti

L'elenco degli argomenti del metodo specifica quali informazioni passate al metodo. Come potete indovinare, queste informazioni – come qualsiasi altra cosa in Java – prendono la forma di oggetti. Così, ciò che dovete specificare nell'elenco degli argomenti sono i tipi degli oggetti da passare e il nome da usare per ciascuno. Come in ogni situazione in Java, dove sembra che stiate distribuendo oggetti, in effetti state passando riferimenti<sup>3</sup>. In ogni caso, il tipo di riferimento deve essere corretto. Se si suppone che l'argomento sia una **String**, quello che passate deve essere una stringa.

Considerate un metodo che prenda una **String** come argomento. Ecco la definizione, che deve essere posta entro una definizione di classe per poter essere compilata:

```
int storage(String s) {
    return s.length() * 2;
}
```

Questo metodo vi dice quanti byte sono necessari per contenere l'informazione in una particolare **String**. (Ogni **char** in una **String** è lungo 16 bit o due byte per supportare i caratteri Unicode.) L'argomento è del tipo **String** e viene chiamato **s**. Una volta che **s** è passato nel metodo, potete trattarlo proprio come ogni altro oggetto. (Potete inviargli

<sup>2</sup> Si possono chiamare metodi **static**, di cui parleremo presto, *per la classe*, senza un oggetto.

<sup>3</sup> Con l'abituale eccezione dei tipi di dati "speciali" citati prima: **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** e **double**. In generale, però, si passano oggetti, il che vuol dire in realtà che si passano riferimenti ag oggetti.

messaggi.) Qui viene chiamato il metodo **length()**, che è uno dei metodi per le **String**, che restituisce il numero di caratteri di una stringa.

Potete inoltre vedere l'uso della parola chiave **return**, che fa due cose. Primo, significa “lascia il metodo, ho finito”. Secondo, se il metodo produce un valore, quel valore viene collocato subito dopo l'enunciato **return**. In questo caso, il valore restituito viene prodotto valutando l'espressione **s.length() \* 2**.

Potete restituire qualsiasi tipo volete, ma se non volete restituire niente del tutto, potete farlo indicando che il metodo restituisce **void**. Ecco qui alcuni esempi:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

Quando il tipo restituito è **void**, la parola chiave **return** viene usata solo per uscire dal metodo, per cui non è necessaria quando arrivate alla fine del metodo. Potete uscire da un metodo in qualsiasi punto, ma se avete dato un tipo restituito non **void** allora il compilatore vi forzerà (con messaggi di errore) a restituire il tipo di valore appropriato indipendentemente da dove uscite.

A questo punto, sembra che un programma non sia altro che un gruppo di oggetti con metodi che prendono altri oggetti come argomenti e inviano messaggi ad altri oggetti. Ed è proprio quello che succede davvero, tuttavia nel prossimo capitolo imparerete come si fa il lavoro minuto a basso livello prendendo decisioni entro un metodo. Per questo capitolo, ci accontenteremo di inviare messaggi.

## Costruire un programma Java

Vi sono diversi altri argomenti da capire prima di passare al primo programma Java.

### Visibilità del nome

Un problema in qualunque linguaggio di programmazione è il controllo dei nomi. Se usate un nome in un modulo del programma e un altro programmatore usa lo stesso nome in un altro modulo, come potete distinguere un nome dall'altro e evitare che i due nomi “siano in conflitto”? In C questo è un problema particolare, perché un programma è spesso un mare di nomi ingovernabile. Le classi C++ (su cui si basano le classi Java) annidano funzioni entro le classi in modo che non possano entrare in conflitto con i nomi di funzione annidati entro altre classi. Tuttavia, il C++ consentiva ancora dati globali e funzioni globali, per cui il conflitto continuava a essere possibile. Per risolvere questo problema, il C++ ha introdotto gli *spazi dei nomi* usando parole chiave aggiuntive.

Java è stato capace di evitare tutto ciò adottando un nuovo approccio. Per produrre un nome non equivoco per una libreria, lo specificatore usato non è diverso da un nome di dominio Internet. Infatti, i creatori di Java vogliono che usiate il vostro nome di dominio Internet a rovescio, che sarà sicuramente univoco. Poiché il mio nome di dominio è **BruceEckel.com**, la mia libreria di utility foibles si chiamerebbe **com.bruceeckel.utility.foibles**. Quando il nome di dominio è rovesciato, i punti stanno a rappresentare sottodirectory.

In Java 1.0 e Java 1.1 le estensioni di dominio **com**, **edu**, **org**, **net** ecc. venivano scritte in lettere maiuscole per convenzione, per cui la libreria si sarebbe chiamata:

**COM.bruceeckel.utility.foibles**. Durante lo sviluppo di Java 2, tuttavia, ci si accorse che questo provocava qualche problema, per cui ora l'intero nome del package è scritto in lettere minuscole.

Questo meccanismo fa sì che tutti i vostri file vivano automaticamente nei loro spazi dei nomi e che ciascuna classe entro un file deve avere un identificatore univoco. Pertanto, non dovete imparare speciali caratteristiche di linguaggio per risolvere questo problema: se ne occupa il linguaggio per conto vostro.

## Uso di altri componenti

Ogni volta che volete usare una classe predefinita nei vostri programmi, il compilatore deve sapere dove trovarla. Naturalmente, la classe potrebbe già trovarsi nello stesso file del codice sorgente dal quale viene chiamata. In questo caso, usate semplicemente la classe, perfino se viene definita solo più avanti nel file. Java elimina il problema del “riferimento in avanti”, per cui non dovete occuparvene.

Che cosa succede quando una classe si trova in un altro file? Potreste pensare che il compilatore dovrebbe essere abbastanza bravo da andare semplicemente a trovarla, ma c'è un problema. Immaginate di voler usare una classe con un nome particolare, ma che per quella classe ci sia più di una definizione (presumibilmente definizioni diverse). O peggio, immaginate di star scrivendo un programma e che mentre lo costruite aggiungete alla vostra libreria una nuova classe che è in conflitto con il nome di una classe esistente.

Per risolvere questo problema dovete eliminare tutte le ambiguità potenziali. Ciò si ottiene dicendo al compilatore Java il nome esatto della classe che volete, usando la parola chiave **import**. **import** dice al compilatore di introdurre un *package*, che è una libreria di classi. (In altri linguaggi, una libreria potrebbe comprendere funzioni e dati oltreché classi, ma ricordate che tutto il codice in Java deve essere scritto dentro una classe.)

La maggior parte delle volte userete componenti delle librerie Java standard che vengono distribuite insieme con il vostro compilatore. Con queste, non avete bisogno di preoccuparvi dei lunghi nomi di dominio scritti a rovescio; voi scrivete, per esempio:

```
import java.util.ArrayList;
```

per dire al compilatore che volete usare la classe **ArrayList** di Java. Tuttavia, **util** contiene un certo numero di classi e potreste volerne usare alcune senza dichiararle tutte esplicitamente. Ciò si ottiene facilmente usando “\*” come carattere jolly:

```
import java.util.*;
```

Di solito si importa una collezione di classi in questo modo, piuttosto che importare classi una per una.

## La parola chiave static

Generalmente, quando create una classe descrivete come si presentano gli oggetti di quella classe e come si comportano. In effetti non ottenete niente finché non create un oggetto di quella classe con **new** e a quel punto si è creato uno spazio di immagazzinamento per i dati e i metodi diventano disponibili.

Ci sono, però, due situazioni in cui questo approccio non è sufficiente. Una è quella in cui volete avere soltanto un segmento di spazio di immagazzinamento per un particolare segmento di dati, indipendentemente da quanti oggetti vengono creati o perfino se non vengono creati oggetti. L'altra è quella in cui avete bisogno di un metodo che non è associato ad alcun particolare oggetto di quella classe. Cioè, vi occorre un metodo che potete chiamare anche se non viene creato alcun oggetto. Potete raggiungere entrambi questi effetti con la parola chiave **static**. Quando dite che qualcosa è **static**, significa che il dato o il metodo non sono legati ad alcuna particolare istanza di oggetto di quella classe. Così, perfino se non avete mai creato un oggetto di quella classe, potete chiamare un metodo

**static** o accedere a un segmento di dati **static**. Con normali dati e metodi non **static**, dovete creare un oggetto e usare quell'oggetto per accedere al dato o metodo, poiché i dati e metodi non **static** devono conoscere il particolare oggetto con il quale lavorano. Naturalmente, poiché i metodi **static** non hanno bisogno che un oggetto sia creato prima del suo uso, non possono accedere direttamente a membri o metodi non **static** semplicemente chiamando gli altri membri senza fare riferimento a un oggetto con nome (poiché membri e metodi non **static** devono essere legati a un particolare oggetto.)

Alcuni linguaggi orientati agli oggetti usano i termini *dati di classe* e *metodi di classe* per dire che i dati e i metodi esistono solo per la classe come un tutto e non per qualche oggetto particolare di quella classe. Talvolta anche la letteratura Java usa questi termini.

Per rendere **static** un membro di dati o un metodo, basta semplicemente mettere la parola chiave prima della definizione. Per esempio, nel seguente esempio si genera un membro di dati **static** e lo si inizializza:

```
class StaticTest {
    static int i = 47;
}
```

Ora, perfino se fate due oggetti **StaticTest**, ci sarà ancora solo un segmento di memorizzazione per **StaticTest.i**. Entrambi gli oggetti si dividono lo stesso **i**. Considerate:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

A questo punto, sia **st1.i** sia **st2.i** hanno lo stesso valore 47 perché si riferiscono allo stesso segmento di memoria.

Ci sono due modi per fare riferimento a una variabile **static**. Come indicato in precedenza, potete dargli un nome attraverso un oggetto, dicendo, per esempio, **st2.i**. Potete fare riferimento direttamente attraverso il suo nome di classe, cosa che non potete fare con un membro non **static**. (Questo è il modo preferito per fare riferimento a una variabile **static**, perché sottolinea la natura **static** di quella variabile.)

```
StaticTest.1++;
```

L'operatore ++ incrementa la variabile. A questo punto, sia **st1.i**, sia **st2.i** assumeranno il valore 48.

Una logica simile vale per i metodi statico. Potete fare riferimento a un metodo statico o tramite un oggetto, come potete fare con qualsiasi metodo, o con la speciale sintassi addizionale **ClassName.method()**. Si definisce un metodo statico in maniera analoga:

```
class StaticFun {
    static void incr() { StaticTest.i++; }
}
```

Potete vedere che il metodo **StaticFun incr()** incrementa il dato **static i**. Potete chiamare **incr()** nel solito modo, attraverso un oggetto:

```
StaticFun sf = new StaticFun();
sf.incr();
```

Oppure, poiché **incr()** è un metodo statico, potete chiamarlo direttamente attraverso la sua classe:

```
StaticFun.incr();
```

Sebbene, quando è applicato a un elemento di dati, **static** cambi decisamente il modo in cui il dato viene creato (uno per ciascuna classe contro uno per ciascun oggetto del non

**static**), quando è applicato a un metodo non è così categorico. Un uso importante di **static** per i metodi è quello che vi permette di chiamare il metodo senza creare un oggetto. Questo è essenziale, come vedremo, per definire il metodo **main()** che è il punto di ingresso per eseguire un'applicazione.

Come qualsiasi altro metodo, un metodo **static** può creare o usare oggetti con nome del suo tipo, per cui un metodo **static** viene spesso usato come “pastore” di un gregge di istanze del proprio tipo.

## Il primo programma Java

Infine, ecco qui il programma<sup>4</sup>. Comincia stampando una stringa e quindi la data, usando la classe **Date** della libreria standard di Java. Notate che qui viene introdotto un altro stile di commento: il segno “//”, che è un commento fino alla fine della riga:

```
// HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```

All'inizio di ciascun file di programma, dovete porre l'enunciato **import** per richiamare eventuali classi aggiuntive che vi occorreranno per il codice di quel file. Notate che ho detto “aggiuntive”; questo perché c'è una certa libreria di classi che vengono automaticamente introdotte in ogni file Java: **java.lang**. Avviate il vostro browser Web ed esaminate la documentazione di Sun. (Se non l'avete scaricata da *java.sun.com* o se non avete altrimenti installato la documentazione Java, fatelo ora.) Selezionate **java.lang**. In questo modo richiamerete un elenco di tutte le classi che fanno parte di quella libreria. Poiché **java.lang** è implicitamente compreso in ogni file di codice Java, queste classi sono disponibili automaticamente. Non c'è una classe **Date** elencata in **java.lang**, il che significa che dovete importare un'altra libreria per usarla. Se non sapete in quale libreria si trova una particolare classe, o se volete vedere tutte le classi, potete selezionare “Tree” nella documentazione Java. Ora potete trovare ogni singola classe che accompagna Java. Quindi potete usare la funzione “trova” del browser per trovare **Date**. Quando lo fate, lo vedrete elencato come **java.util.Date**; ciò vi fa sapere che è nella libreria **util** e che dovete indicare **import.java.util.\*** per usare **Date**.

Se ritornate all'inizio, selezionate **java.lang** e poi **System**, vedrete che la classe **System** ha diversi campi e se selezionate **out** scoprirete che è un oggetto **static PrintStream**. Poiché

---

<sup>4</sup> Alcuni ambienti di programmazione fanno scorrere fulmineamente i programmi sullo schermo e li chiudono prima che abbiate avuto la possibilità di vedere i risultati. Potete inserire questo frammento di codice alla fine di una **main()** per ottenere una pausa dell'output:

```
try {
    System.in.read();
} catch (Exception e) {}
```

In questo modo l'output verrà sospeso fino a quando non premerete Invio (o qualunque altro tasto). Questo codice richiama concetti che verranno presentati soltanto molto più avanti nel libro, quindi per il momento non siete in grado di capirlo, però funziona.

è **static**, non dovete creare nulla. L'oggetto **out** è sempre lì e non avete che da usarlo. Ciò che potete fare con questo oggetto **out** è determinato dal tipo che è: un **PrintStream**. Giustamente, **PrintStream** è indicato nella descrizione come un ipercollegamento, per cui se vi fate sopra un clic vedrete una lista di tutti i metodi che potete chiamare per **PrintStream**. Ce ne sono parecchi e saranno descritti più avanti in questo libro. Per ora, tutto ciò che ci interessa è **println()**, che in effetti significa “stampa sulla console quello che invio e finisci con una nuova riga”. Così, in ogni programma Java che scrivete potete dire **System.out.println(“things”)** tutte le volte che volete stampare qualcosa sulla console.

Il nome della classe è uguale al nome del file. Quando create un programma autonomo come questo, una delle classi del file deve avere lo stesso nome del file. (Il compilatore si lamenta se non lo fate.) Quella classe deve contenere un metodo chiamato **main()** con la firma indicata:

```
public static void main(String[] args) {
```

La parola chiave **public** significa che il metodo è disponibile al mondo esterno (descritto nei particolari nel Capitolo 5). L'argomento di **main()** è un array di oggetti **String**. Gli **args** non verranno usati in questo programma, ma il compilatore Java insiste che siano lì perché conservano gli argomenti chiamati sulla riga di comando.

La riga che stampa la data è piuttosto interessante:

```
System.out.println(new Date());
```

Si consideri l'argomento: un oggetto **Date** viene creato solo per inviare il suo valore a **println()**. Non appena questo enunciato è finito, quella **Date** non è più necessaria e il raccoglitore di spazzatura può farsi avanti e prendersela in ogni momento. Non dobbiamo preoccuparci della sua eliminazione.

## Compilazione ed esecuzione

Per compilare ed eseguire questo programma e tutti gli altri programmi di questo libro, dovete prima avere un ambiente di programmazione Java. Vi sono diversi ambienti di sviluppo realizzati da terzi, ma in questo libro supponiamo che stiate usando il JDK della Sun, che è gratuito. Se utilizzate un altro sistema di sviluppo, dovete consultare la documentazione di quel sistema per stabilire come compilare e eseguire i programmi.

Andate su Internet al sito [java.sun.com](http://java.sun.com). Lì troverete informazioni e collegamenti che vi guideranno attraverso la procedura per scaricare e installare il JDK per la vostra particolare piattaforma.

Una volta che il JDK è installato e avete impostato le informazioni sul percorso in modo che possa trovare **javac** e **java**, scaricate e scompattate il codice sorgente di questo libro (disponibile nel Booksite abbinato a questo volume, [www.apogeonline.com/libri/00998/](http://www.apogeonline.com/libri/00998/) allegati o sul sito [www.bruceeckel.com](http://www.bruceeckel.com)). In questo modo otterrete una sottodirectory per ciascun capitolo. Andate nella sottodirectory **c02** e digitate:

```
javac HelloDate.java
```

Questo comando non dovrebbe produrre alcuna risposta. Se ricevete un qualche tipo di messaggio d'errore, significa che non avete installato il JDK correttamente e quindi dovrete analizzare il problema.

Se, invece ricevete in risposta solo il prompt di comando, potete scrivere:

```
java HelloDate
```

e quindi otterrete come output il messaggio e la data.

Questo è il procedimento che potete seguire per compilare e eseguire ciascuno dei programmi di questo libro. Tuttavia, vedrete che il codice sorgente di questo libro ha anche un file chiamato **makefile** in ciascun capitolo e questo contiene comandi “make” per costruire automaticamente i file di quel capitolo. Guardate la pagina Web di questo libro all’indirizzo *www.BruceEckel.com* per i particolari su come usare i makefile.

## Commenti e documentazione incorporati

Ci sono due tipi di commenti in Java. Il primo è il tradizionale commento stile C che è stato ereditato dal C++. Questi commenti cominciano con i caratteri `/*` e continuano, possibilmente su molte righe, fino alla coppia di caratteri `*/`. Notate che molti programmatori iniziano ciascuna riga di un commento continuo con un `*`, così vedrete spesso:

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

Ricordate, tuttavia che tutto ciò che si trova fra `/*` e `*/` viene ignorato, per cui non c’è alcuna differenza se scrivete:

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

La seconda forma di commento viene dal C++. È il commento in unica riga, che comincia con `//` e continua fino alla fine della riga. Questo tipo di commento è conveniente e comunemente usato per la sua facilità. Non occorre cercare sulla tastiera per trovare `/` e poi `*` (basta premere lo stesso tasto due volte); inoltre, non è necessario chiudere il commento. Pertanto spesso vedrete:

```
// questo è un commento su una sola riga
```

## Documentazione basata sui commenti

Uno dei pregi del linguaggio Java è che i suoi progettisti non hanno ritenuto che scrivere codice fosse la sola attività importante, ma hanno pensato anche alla sua documentazione. Forse il problema principale della documentazione del codice è il suo aggiornamento. Se la documentazione e il codice sono separati, diventa difficile cambiare la documentazione ogni volta che si cambia il codice. La soluzione sembra semplice: collegare il codice alla documentazione. Il modo più semplice per farlo è mettere tutto nello stesso file. Per completare il quadro, tuttavia, occorrono una sintassi speciale per i commenti, per marcare la documentazione, e uno strumento per estrarre quei commenti e metterli in una forma utilizzabile. Questo è ciò che ha fatto Java.

Lo strumento per estrarre i commenti si chiama *javadoc*. Utilizza un po’ della tecnologia del compilatore Java per individuare le tag speciali di commento che mettete nei programmi e non solo estrae le informazioni contrassegnate con queste tag, ma tira fuori anche il nome della classe o il nome del metodo vicini al commento. In questo modo potete cavarvela con una minima quantità di lavoro per generare un’adeguata documentazione dei vostri programmi.

L’output di *javadoc* è un file HTML che potete visualizzare con il vostro browser Web. Questo strumento vi permette di creare e gestire un unico file sorgente, generando automaticamente documentazione utilizzabile. Grazie a *javadoc* abbiamo uno standard per creare la documentazione ed è sufficientemente facile da autorizzarci a esigere che tutte le librerie Java siano documentate.

## Sintassi

Tutti i comandi javadoc si trovano solo entro commenti `/**`. I commenti finiscono con `*/` come al solito. Vi sono due modi principali di usare javadoc: incorporare HTML o usare “doc tag”. I doc tag sono comandi che cominciano con un “@” e sono collocati all’inizio di una riga di commento. (Un “\*” iniziale viene ignorato, però.)

Vi sono tre “tipi” di documentazione di commento, che corrispondono all’elemento che il commento precede: classe, variabile o metodo. Cioè, un commento di classe compare esattamente prima della definizione di una classe; un commento di variabile appare appena prima della definizione di una variabile e un commento di metodo si trova subito prima della definizione di un metodo. Ecco un semplice esempio:

```
/** Commento di una classe */
public class docTest {
    /** Commento di una variabile */
    public int i;
    /** Commento di un metodo */
    public void f() {}
}
```

Si noti che javadoc elaborerà la documentazione di commento soltanto per membri **public** e **protected**. I commenti per membri **private** e “amichevoli” (vedere il Capitolo 5) vengono ignorati, per cui non avrete alcun output. (Tuttavia, potete usare il flag **-private** per comprendere anche i membri **private**.) Questo ha senso, poiché soltanto i membri **public** e **protected** sono disponibili fuori dal file, che è ciò che il programmatore client si aspetta. Tuttavia, tutti i commenti di **class** sono compresi nell’output. L’output del codice precedente è un file HTML che ha lo stesso formato standard di tutto il resto della documentazione Java; così gli utilizzatori saranno a proprio agio con il formato e potranno navigare facilmente nelle vostre classi. Vale la pena caricare il codice precedente, farlo passare sotto javadoc ed esaminare il file HTML ottenuto per vederne i risultati.

## HTML incorporato

Javadoc passa comandi HTML al documento HTML generato. Ciò vi consente di utilizzare appieno l’HTML. Tuttavia, il motivo principale è quello di consentirvi di formattare il codice, come per esempio:

```
/**
 * <pre>
 * System.out.println(new Date());
 * </pre>
 */
```

Potete inoltre usare HTML proprio come fareste in qualsiasi altro documento Web per formattare il normale testo nelle vostre descrizioni:

```
/**
 * Potete <em>persino</em> inserire un elenco:
 * <ol>
 * <li> Elemento uno
 * <li> Elemento due
 * <li> Elemento tre
 * </ol>
 */
```

Si noti che entro il commento di documentazione, gli asterischi all'inizio di una riga vengono eliminati da javadoc, insieme agli spazi iniziali. Javadoc riformatta tutto in modo da renderlo conforme all'aspetto della documentazione standard. Non usate titoli come <H1> o <Hr> come HTML incorporato, perché javadoc inserisce i propri titoli e i vostri possono interferire con quelli.

Tutti i tipi di documentazione di commento – classe, variabile e metodo – possono supportare l'HTML incorporato.

### **@see: riferimento alle altre classi**

Tutti e tre i tipi di commenti di documentazione (classe, variabile e metodo) possono contenere tag **@see**, che vi lasciano fare riferimento alla documentazione di altre classi. Javadoc genererà HTML con le tag **@see** ipercollegate all'altra documentazione. Le forme sono:

```
@see nomeclasse
@see nomeclasse_pienamente_qualificato
@see nomeclasse_pienamente_qualificato#nome_metodo
```

### **Tag di documentazione delle classi**

Insieme con l'HTML incorporato e i riferimenti **@see**, la documentazione delle classi può comprendere tag per informazioni sulla versione e il nome dell'autore. La documentazione delle classi può inoltre essere usata per le *interfacce* (vedere il Capitolo 8).

#### **@version**

Questa ha la forma:

```
@version informazioni_sulla_versione
```

in cui **informazioni\_sulla\_versione** è qualsiasi informazione significativa che ritenete conveniente inserire. Quando il flag **-version** è posto sulla riga di comando di javadoc, le informazioni sulla versione saranno richiamate specialmente nella documentazione HTML generata.

#### **@author**

Questa ha la forma:

```
@author informazioni_sull'autore
```

in cui **informazioni\_sull'autore** è, presumibilmente, il vostro nome, ma potrebbe anche comprendere il vostro indirizzo e-mail o qualsiasi altra informazione appropriata. Quanto il flag **-author** è posto sulla riga di comando di javadoc, le informazioni sull'autore saranno richiamate specialmente nella documentazione HTML generata. Si possono avere più tag di autore per un elenco di autori, ma devono essere poste una di seguito all'altra. Tutte le informazioni sull'autore saranno raggruppate in un singolo paragrafo dell'HTML generato.

#### **@since**

Questa tag vi permette di indicare la versione di quel codice che cominciò a usare una particolare caratteristica. Lo vedrete apparire nella documentazione Java HTML per indicare quale versione di JDK è stata usata.

## Tag di documentazione delle variabili

La documentazione delle variabili può comprendere solo HTML incorporati e riferimenti `@see`.

## Tag di documentazione dei metodi

Come la documentazione incorporata e i riferimenti `@see`, i metodi prevedono tag per parametri, valori restituiti e eccezioni.

### **@param**

Questa ha la forma:

```
@param nome_parametro descrizione
```

in cui **nome\_parametro** è l'identificatore nell'elenco dei parametri e **descrizione** è un testo che può continuare su righe successive. La descrizione si considera finita quando si incontra una nuova tag di documentazione. Potete averne un qualsiasi numero, presumibilmente una per ciascun parametro.

### **@return**

Questa ha la forma:

```
@return descrizione
```

in cui **descrizione** vi dà il significato del valore restituito e può continuare su righe successive.

### **@throws**

Le eccezioni saranno trattate nel Capitolo 10, ma in breve possiamo dire che sono oggetti che possono essere “gettati” fuori da un metodo, se il metodo fallisce. Sebbene quando chiamate un metodo possa emergere una sola eccezione, un particolare metodo potrebbe produrre un qualsiasi numero di tipi diversi di eccezioni che richiedono tutte una descrizione. Così, la forma del tag delle eccezioni è:

```
@throws nome_classe_pienamente_qualificato descrizione
```

in cui **nome\_classe\_pienamente\_qualificato** fornisce un nome non equivoco di una classe di eccezioni che è definita da qualche parte e **descrizione** (che può continuare su righe successive) vi dice perché questo particolare tipo di eccezione può emergere dalla chiamata di metodo.

### **@deprecated**

Questo tag viene usata per indicare caratteristiche che sono state superate da una caratteristica migliore. Il tag “deprecated” suggerisce di non usare più una particolare caratteristica, perché è probabile che in futuro sarà eliminata. Se viene usato un metodo contrassegnato con il tag `@deprecated`, il compilatore emette un avvertimento.

## Esempio di documentazione

Ecco qui ancora il primo programma Java, questa volta con l'aggiunta dei commenti di documentazione:

```
//: c02:HelloDate.java
import java.util.*;
```

```

/** Il primo esempio di programma di questo libro.
 * Visualizza una stringa e la data di oggi.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 2.0
 */
public class HelloDate {
    /** Unico punto di ingresso per la classe e l'applicazione
     * @param args array di argomenti stringa
     * @return Nessun valore restituito
     * @exception exceptions Nessuna eccezione sollevata
     */
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
} ///:-

```

La prima riga del file usa la mia tecnica di mettere un “:” come contrassegno speciale della riga di commento contenente il nome del file sorgente. Quella riga contiene le informazioni di percorso del file (in questo caso, **c02** indica il Capitolo 2) seguite dal nome del file<sup>5</sup>. Anche l’ultima riga finisce con un commento indicante la fine del listato del codice sorgente, che ne permette l’estrazione automatica dal testo di questo libro e la sua verifica con il compilatore.

## Stile di codifica

Uno standard non ufficiale in Java è quello di scrivere in carattere maiuscolo la prima lettera del nome di una classe. Se il nome della classe consiste in diverse parole, queste vengono scritte tutte di seguito (cioè, non si usano trattini bassi per separare i nomi) e la prima lettera di ciascuna parola incorporata viene scritta in carattere maiuscolo, come per esempio:

```
class AllTheColorsOfTheRainbow { // ...
```

Per quasi tutto il resto: metodi, campi (variabili membro) e nomi di riferimento agli oggetti, lo stile accettato è esattamente quello adottato per le classi *eccetto* che la prima lettera dell’identificatore è minuscola.

Per esempio:

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}

```

Naturalmente, ricordate che anche l’utente deve scrivere tutti questi nomi lunghi, quindi siate clementi.

Anche il codice Java che vedrete nelle librerie Sun segue l’uso delle parentesi graffe aperte e chiuse che viene fatto in questo libro.

---

<sup>5</sup> Uno strumento che ho creato utilizzando Python (si veda [www.python.org](http://www.python.org)) utilizza queste informazioni per estrarre i file del codice, metterli in opportune sottodirectory e creare makefile.

## Riepilogo

In questo capitolo avete visto abbastanza programmazione Java per capire come si scrive un semplice programma; inoltre avete potuto passare in rassegna il linguaggio e alcuni suoi concetti fondamentali. Tuttavia, gli esempi fin qui visti sono tutti del tipo “fa questo, poi fa quello, poi fa qualcosa d’altro”. E se volete che il programma faccia scelte come “se il risultato di fare questo è rosso, fa quello; altrimenti, fa qualcosa d’altro”? Vedremo nel prossimo capitolo il supporto in Java per questa basilare attività di programmazione.

## Esercizi

1. Seguendo l’esempio **HelloDate.java** in questo capitolo, create un programma “hello, world” che stampi semplicemente questo enunciato. Vi occorre solamente un metodo singolo nella vostra classe (quello “main” che viene eseguito quando si avvia il programma). Ricordatevi di farlo **static** e di inserire l’elenco degli argomenti, anche se non lo usate. Compilate il programma con **javac** e eseguitelo usando **java**. Se usate un ambiente di sviluppo diverso da JDK, imparate come si fa a compilare e eseguire programmi in quell’ambiente.
2. Trovate i frammenti di codice riguardanti **ATypeName** e convertiteli in un programma che possa essere compilato e eseguito.
3. Convertite i frammenti di codice **DataOnly** in un programma che possa essere compilato e eseguito.
4. Modificate l’Esercizio 3 in modo che i valori dei dati in **DataOnly** siano assegnati e stampati in **main()**.
5. Scrivete un programma che comprenda e chiami il metodo **storage()** definito come frammento di codice in questo capitolo.
6. Convertite i frammenti di codice **StaticFun** in un programma funzionante.
7. Scrivete un programma che stampi tre argomenti presi dalla riga di comando. Per fare ciò dovrete indicizzare l’array di **String** delle righe di comando.
8. Convertite l’esempio **AllTheColorsOfTheRainbow** in un programma che possa essere compilato e eseguito.
9. Trovate il codice della seconda versione di **HelloDate.java**, che è un semplice esempio di documentazione tramite commenti. Eseguite **javadoc** sul file ed esaminate i risultati con il vostro browser Web.
10. Convertite **docTest** in un file che possa essere compilato e quindi fatelo passare sotto attraverso **javadoc**. Verificate la documentazione risultante con il vostro browser Web.
11. Aggiungete un elenco di articoli HTML alla documentazione nell’Esercizio 10.
12. Prendete il programma dell’Esercizio 1 e aggiungetegli la documentazione di commento. Estraiete questa documentazione in un file HTML usando **javadoc** ed esaminatelo con il vostro browser Web.