Apprendimento di ACTIONSCRIPT® 3.0

Note legali

Sommario

Capitolo 1: Introduzione ad ActionScript 3.0

Informazioni su ActionScript

ActionScript è il linguaggio di programmazione per gli ambienti runtime Adobe® Flash® Player e Adobe® AIR™. Rende possibili l'interattività, la gestione dei dati e molte altre operazioni relative al contenuto e alle applicazioni Flash, Flex e AIR.

ActionScript viene eseguito dall'AVM (ActionScript Virtual Machine), che fa parte di Flash Player e AIR. Il codice ActionScript viene solitamente trasformato in formato codice byte (una sorta di linguaggio di programmazione generato e interpretato dai computer) mediante un compilatore. Esempi di compilatori includono quello incorporato in Adobe® Flash® Professional o in Adobe® Flash® Builder™ e disponibile nell'SDK di Adobe® Flex™. Il codice byte viene incorporato in file SWF, che vengono quindi eseguiti in Flash Player e AIR.

ActionScript 3.0 offre un solido modello di programmazione che risulta familiare agli sviluppatori che hanno un'esperienza di base nella programmazione a oggetti. Alcune funzioni chiave di ActionScript 3.0 che comprendono miglioramenti rispetto alle versioni precedenti di ActionScript comprendono:

- una nuova macchina virtuale ActionScript, detta AVM2, che utilizza un nuovo set di istruzioni per il codice byte e garantisce un notevole miglioramento delle prestazioni;
- una base di codice del compilatore più moderna che garantisce ottimizzazioni più complete rispetto alle versioni precedenti del compilatore;
- un'API (Application Programming Interface) estesa e migliorata, con controllo di basso livello degli oggetti e modello realmente orientato agli oggetti;
- un'API XML basata su ECMAScript for XML (E4X), come indicato nella specifica ECMA-357 edizione 2. E4X è un'estensione di ECMAScript che aggiunge XML al linguaggio come tipo di dati nativo;
- un modello di eventi basato sulla specifica degli eventi DOM (Document Object Model) Livello 3.

Vantaggi di ActionScript 3.0

ActionScript 3.0 estende le funzionalità delle versioni precedenti del linguaggio di script ActionScript. È concepito per facilitare la creazione di applicazioni sofisticate con set di dati di grandi dimensioni e con basi di codice a oggetti e riutilizzabili. Sebbene ActionScript 3.0 non sia obbligatorio per creare contenuto da eseguire in Adobe Flash Player, la nuova versione garantisce miglioramenti delle prestazioni possibili solo con AVM2 (la macchina virtuale di ActionScript 3.0). Il codice ActionScript 3.0 può essere eseguito a una velocità fino a otto volte maggiore rispetto al codice ActionScript delle versioni precedenti.

La versione precedente della macchina virtuale ActionScript, AVM1, è in grado di eseguire codice ActionScript 1.0 e 2.0. Flash Player 9 e 10 supportano AVM1 per compatibilità con le versioni precedenti.

Nuove funzionalità di ActionScript 3.0

Sebbene ActionScript 3.0 contenga molte classi e caratteristiche simili a ActionScript 1.0 e 2.0, la nuova versione del linguaggio è concettualmente e "architettonicamente" diversa da quelle precedenti. I miglioramenti introdotti in ActionScript 3.0 riguardano varie caratteristiche aggiunte al linguaggio di base e un'API di migliorata, che offre un maggiore controllo sugli oggetti di basso livello.

Caratteristiche del linguaggio di base

Il linguaggio di base (core language) definisce i blocchi costitutivi del linguaggio di programmazione, quali le istruzioni, le espressioni, le condizioni, i cicli e i tipi. ActionScript 3.0 include molte caratteristiche che consentono di velocizzare il processo di sviluppo.

Eccezioni runtime

ActionScript 3.0 segnala più condizioni di errore rispetto alle versioni precedenti del linguaggio. Le eccezioni runtime sono utilizzate per le condizioni di errore più comuni e consentono di migliorare il lavoro di debug e di sviluppare applicazioni più efficienti nella gestione degli errori. Per gli errori runtime sono disponibili tracce di stack annotate con l'indicazione del file di origine e del numero di riga, che consentono una rapida individuazione degli errori.

Tipi runtime

In ActionScript 3.0, le informazioni sul tipo vengono mantenute in fase di runtime e utilizzate per eseguire la verifica del tipo in fase di runtime, migliorando la sicurezza dei tipi nel sistema. Le informazioni sul tipo vengono utilizzate anche per indicare le variabili nelle rappresentazioni native dei sistemi, migliorando le prestazioni e riducendo l'uso di memoria. Per fare un confronto, in ActionScript 2.0 le annotazioni di tipo sono principalmente un aiuto per lo sviluppatore e a tutti i valori viene assegnato un tipo in modo dinamico.

Classi chiuse

ActionScript 3.0 include il concetto di classe chiusa (sealed). Una classe chiusa possiede unicamente le proprietà e i metodi definiti in fase di compilazione e non consente l'aggiunta di ulteriori proprietà o metodi. L'incapacità di modificare una classe in fase di runtime consente verifiche più rigorose in fase di compilazione e permette di sviluppare programmi più solidi. Inoltre, migliora l'uso della memoria perché non è necessaria una tabella di hash interna per ogni istanza di oggetto. Le classi dinamiche possono essere specificate mediante la parola chiave `dynamic`. Tutte le classi di ActionScript 3.0 sono chiuse ("sealed") per impostazione predefinita, ma possono essere dichiarate dinamiche mediante la parola chiave `dynamic`.

Chiusure di metodo

In ActionScript 3.0 una chiusura di metodo è in grado di memorizzare la propria istanza di oggetto originale. Questa caratteristica è utile per la gestione degli eventi. In ActionScript 2.0, le chiusure di metodo non memorizzano l'istanza di oggetto dalla quale venivano estratte e si verifica pertanto un comportamento inatteso quando una chiusura di metodo viene chiamata.

ECMAScript for XML (E4X)

ActionScript 3.0 implementa ECMAScript for XML (E4X), recentemente standardizzato come ECMA-357. E4X offre una serie funzionale di costrutti di linguaggio per la gestione del codice XML. A differenza delle tradizionali API di analisi XML, in E4X XML viene elaborato come tipo di dati nativo del linguaggio. E4X sveltisce lo sviluppo di applicazioni che utilizzano il codice XML riducendo drasticamente la quantità di codice richiesta.

Per visualizzare la specifica E4X di ECMA, visitate il sito Web www.ecma-international.org.

Espressioni regolari
ActionScript 3.0 include il supporto nativo per le espressioni regolari e permette quindi di trovare e modificare rapidamente stringhe di codice. ActionScript 3.0 implementa il supporto delle espressioni regolari così come sono definite nella specifica del linguaggio ECMAScript edizione 3 (ECMA-262).

Spazi dei nomi
Gli spazi dei nomi (namespace) sono simili ai tradizionali specificatori di accesso utilizzati per controllare la visibilità delle dichiarazioni (`public`, `private`, `protected`). Funzionano come specificatori di accesso personalizzati, ai quali è possibile assegnare un nome a piacere. Gli spazi dei nomi includono un URI (Universal Resource Identifier) per evitare possibili conflitti e vengono utilizzati anche per rappresentare gli spazi dei nomi XML quando si lavora con il linguaggio E4X.

Nuovi tipi di base
ActionScript 3.0 contiene tre tipi numerici: Number, int e uint. Number corrisponde a un numero a virgola mobile e precisione doppia. Il tipo int è un numero intero a 32 bit con segno, che consente al codice ActionScript di sfruttare la velocità di elaborazione matematica dei numeri interi della CPU. È utile nei contatori di ciclo e nelle variabili in cui sono utilizzati i numeri interi. Il tipo uint è un numero intero a 32 bit senza segno, utile per i valori di colore RGB, i conteggi di byte e altre situazioni. Per contro, ActionScript 2.0 dispone di un unico tipo numerico, Number.

Caratteristiche dell'API

Le API di in ActionScript 3.0 contengono molte classi che consentono di controllare gli oggetti a basso livello. L'architettura del linguaggio è progettata per essere più intuitiva rispetto alle versioni precedenti. Sebbene il numero delle classi sia troppo elevato per poterle descrivere tutte in dettaglio, è opportuno sottolineare alcune differenze significative.

Modello di eventi DOM3
Il modello di eventi DOM 3 (Document Object Model Level 3) fornisce una modalità standardizzata per la generazione e la gestione dei messaggi di evento. Questo modello di eventi è progettato per consentire agli oggetti delle applicazioni di interagire e comunicare, mantenendo il proprio stato e reagendo ai cambiamenti. Il modello di eventi ActionScript 3.0 è realizzato in base alla Specifica degli eventi DOM Level 3 del World Wide Web Consortium e offre un meccanismo più semplice ed efficiente rispetto agli eventi di sistema disponibili nelle versioni precedenti di ActionScript.

Gli eventi e gli errori si trovano nel pacchetto flash.events. I componenti Flash Professional e il framework Flex utilizzano lo stesso modello di eventi, quindi il sistema di eventi è uniforme in tutta la piattaforma Flash.

API dell'elenco di visualizzazione
L'API per l'accesso all'elenco di visualizzazione, l'albero contenente gli eventuali elementi visivi nell'applicazione, è composta da classi per utilizzare valori di base visivi.

La classe Sprite è un blocco strutturale leggero, progettato come classe base per elementi visivi quali interfacce e componenti. La classe Shape rappresenta le forme vettoriali raw. Per queste classi è possibile creare "naturalmente" le istanze con l'operatore `new`, nonché riassegnare dinamicamente l'elemento principale in qualunque momento.

La gestione della profondità è automatica. Sono disponibili dei metodi per specificare e gestire l'ordine di impilamento degli oggetti.

Gestione di dati e contenuti dinamici

ActionScript 3.0 include dei meccanismi intuitivi e uniformi nell'intera API per il caricamento e la gestione di risorse e dati nell'applicazione La classe Loader fornisce un sistema unico per il caricamento di file SWF e immagini e permette di accedere a informazioni dettagliate sul contenuto caricato. La classe URLLoader offre un meccanismo distinto per il caricamento di testo e dati binari nelle applicazioni basate su dati, mentre la classe Socket permette di leggere e scrivere dati binari nei socket server in qualunque formato.

Accesso di basso livello ai dati

Varie API forniscono accesso di basso livello ai dati. Per i dati in corso di scaricamento, la classe URLStream permette di accedere ai dati in formato raw binario mentre vengono scaricati. La classe ByteArray consente di ottimizzare la lettura, la scrittura e la gestione dei dati binari. L'API Sound fornisce un controllo dettagliato del suono mediante le classi SoundChannel e SoundMixer. API relative alla sicurezza forniscono informazioni sui privilegi di sicurezza di un file SWF o di un contenuto caricato, consentendo la gestione degli errori di sicurezza.

Operazioni con il testo

Il pacchetto flash.text di ActionScript 3.0 contiene tutte le API relative al testo. La classe TextLineMetrics fornisce dati metrici dettagliati per una riga di testo all'interno di un campo di testo; sostituisce il metodo `TextFormat.getTextExtent()` in ActionScript 2.0. La classe TextField contiene metodi di basso livello in grado di fornire informazioni specifiche su una riga di testo o un singolo carattere in un campo di testo. Ad esempio, il metodo `getCharBoundaries()` restituisce un rettangolo che rappresenta il riquadro di delimitazione di un carattere. Il metodo `getCharIndexAtPoint()` restituisce l'indice del carattere in un punto specifico. Il metodo `getFirstCharInParagraph()` restituisce l'indice del primo carattere di un paragrafo. I metodi a livello di riga includono `getLineLength()`, che restituisce il numero di caratteri di una riga di testo specifica, e `getLineText()`, che restituisce il testo della riga specificata. La classe Font consente di gestire i caratteri incorporati nei file SWF.

Per un controllo sul testo di livello ancora più basso, le classi nel pacchetto flash.text.engine costituiscono Flash Text Engine. Questo set di classi fornisce controllo di basso livello sul testo ed è progettato per creare framework e componenti di testo.

Capitolo 2: Guida introduttiva ad ActionScript

Nozioni fondamentali sulla programmazione

Dal momento che ActionScript è un linguaggio di programmazione, è utile avere presenti alcuni concetti fondamentali riguardanti la programmazione.

Che cosa sono i programmi

Per iniziare, è importante avere presente, da un punto di vista concettuale, che cosa si intende per programma di un computer e che cosa fa un programma. I concetti fondamentali relativi ai programmi sono due:

- Un programma è una serie di istruzioni o procedure che il computer esegue.
- Ogni procedura consiste, in definitiva, nella manipolazione di informazioni o dati.

Un programma per computer è un elenco di comandi dettagliati che l'utente impartisce al computer e che il computer esegue in sequenza. Ogni singolo comando si chiama *istruzione* e tutte le istruzioni di ActionScript terminano con un punto e virgola.

In sostanza, l'istruzione di un programma manipola dei dati che si trovano nella memoria del computer. Un esempio semplice è impartire un'istruzione che richiede al computer di sommare due numeri e di salvare il risultato in memoria. Un esempio più complesso è scrivere un programma che sposta un rettangolo disegnato sullo schermo in una posizione diversa. Il computer memorizza alcune informazioni sul rettangolo, come le coordinate x e y che ne descrivono la posizione attuale, le misure della base e dell'altezza, il colore e così via. Tutti questi frammenti di informazioni vengono trattenuti nella memoria del computer. Un programma in grado di spostare il rettangolo in un punto diverso dello schermo conterrà comandi del tipo “imposta la coordinata x su 200; imposta la coordinata y su 150”. Si tratta, in altre parole, di specificare quali nuovi valori il computer deve utilizzare come coordinate x e y. In modo completamente trasparente per l'utente, il computer interpreta e manipola i dati forniti per poter convertire questi valori nell'immagine visualizzata sullo schermo. Tuttavia, per il livello base di apprendimento, è sufficiente sapere che l'operazione “sposta il rettangolo in un punto diverso dello schermo” consiste nel cambiare alcuni dati presenti nella memoria del computer.

Variabili e costanti

Dal momento che la programmazione consiste principalmente nella manipolazione di dati presenti nella memoria del computer, è importante disporre di un sistema per rappresentare un singolo valore all'interno di un programma. L'entità che rappresenta un valore presente nella memoria del computer si chiama *variabile*. Nel momento in cui scrivete le istruzioni del programma, inserite il nome della variabile in luogo del suo valore; è infatti compito del computer prelevare il valore che ha in memoria ogni volta che incontra quella determinata variabile. Ad esempio, supponiamo di avere due variabili chiamate `value1` e `value2` e che entrambe contengano un numero. Per sommare i due valori delle variabili potreste scrivere la seguente istruzione:

```
value1 + value2
```

Quando esegue l'istruzione, il computer individua il valore di ciascuna variabile e li somma.

In ActionScript 3.0, le variabili sono caratterizzate da tre aspetti:

- Il nome della variabile
- Il tipo di dati che la variabile può rappresentare
- L'effettivo valore memorizzato nel computer

Avete già visto come il computer usa il nome della variabile come segnaposto del valore. Ma il tipo di dati è un aspetto altrettanto importante. Quando create una variabile in ActionScript, specificate il tipo di dati che la variabile potrà contenere. Da questo momento in avanti, le istruzioni del programma potranno memorizzare solo quel tipo di dati nella variabile e qualsiasi operazione eseguita sul valore dovrà rispettare le caratteristiche associate al tipo di dati assegnato. Per creare una variabile in ActionScript, operazione comunemente definita come *dichiarare* una variabile, utilizzate l'istruzione `var`:

```
var value1:Number;
```

In questo esempio viene chiesto al computer di creare una variabile denominata `value1`, che può contenere solo dati di tipo Number. ("Number" è un tipo di dati specifico definito in ActionScript.) In alternativa, potete associare immediatamente un valore alla variabile:

```
var value2:Number = 17;
```

Adobe Flash Professional
In Flash Professional esiste un altro modo per dichiarare una variabile. Quando inserite un simbolo di clip filmato, un simbolo di pulsante o un campo di testo sullo stage, potete associarlo a un nome di istanza nella finestra di ispezione Proprietà. In modo completamente trasparente per l'utente, Flash Professional crea una variabile con lo stesso nome dell'istanza, che potete utilizzare nel codice ActionScript per chiamare quell'elemento dello stage. Supponete, ad esempio, di assegnare a un simbolo di clip filmato disponibile sullo stage il nome istanza `rocketShip`. Ogni volta che utilizzate la variabile `rocketShip` nel codice ActionScript, state in realtà modificando quel clip filmato.

Una *constante* è simile a una variabile in quanto è un nome che rappresenta un valore nella memoria del computer, con associato un tipo di dati specifico. La differenza è che a una costante è possibile assegnare un valore una sola volta nel corso di un'applicazione ActionScript. Il valore assegnato a una costante è lo stesso in tutta l'applicazione. La sintassi di dichiarazione di una costante è identica a quella di dichiarazione di una variabile, tranne che si utilizza la parola chiave `const` anziché la parola chiave `var`:

```
const SALES_TAX_RATE:Number = 0.07;
```

Una costante è utile per definire un valore utilizzato in più punti all'interno di un progetto e che, in circostanze normali, non viene modificato. L'utilizzo di una costante anziché un valore letterale semplifica la lettura del codice. Ad esempio, considerate due versioni dello stesso codice: uno che moltiplica un prezzo per `SALES_TAX_RATE` e l'altro che moltiplica il prezzo per `0,07`. La versione che utilizza la costante `SALES_TAX_RATE` è più semplice da capire. Inoltre, supponente che il valore definito dalla costante cambi. Se utilizzate una costante per rappresentare il valore nel progetto, potete modificare il valore in un unico punto (la dichiarazione di costante). Al contrario, se utilizzate valori letterali codificati nel codice, dovete modificarli in vari punti.

Tipi di dati

ActionScript mette a disposizione vari tipi di dati da utilizzare per le variabili. Alcuni possono essere considerati tipi di dati "semplici" o "fondamentali":

- String: unità di testo, come un nome o il capitolo di un libro

- Numeric: ActionScript 3.0 comprende tre sottogruppi specifici per i dati numerici:
  - Number: qualsiasi valore numerico, con o senza decimali
  - int: numero intero senza decimali
  - uint: numero intero "senza segno", ovvero un numero intero che non può essere negativo
- Boolean: tipo di dati i cui gli unici valori possibili sono vero o falso, che descrive, ad esempio, se uno switch è attivo o se due valori sono uguali

I tipi di dati semplici rappresentano un'informazione singola: ad esempio, un numero singolo o una singola sequenza di testo. Tuttavia, la maggior parte dei tipi di dati definiti in ActionScript sono complessi e rappresentano un insieme di valori in un contenitore singolo. Ad esempio, una variabile del tipo Date rappresenta un valore unico, cioè una data precisa. Ciò nonostante, il valore data è rappresentato come una serie di valori: giorno, mese, anno, ore, minuti, secondi e così via, ciascuno dei quali è un numero singolo. Sebbene la data sia in genere concepita come un valore singolo (e la potete trattare come tale impostando una variabile Date), internamente il computer considera la data come una serie di valori che, combinati, definiscono un momento preciso.

La maggior parte dei tipi di dati incorporati, e dei tipi di dati definiti dai programmatori, sono in realtà dati complessi. Alcuni dei tipi di dati complessi più conosciuti sono:

- MovieClip: un simbolo del clip filmato
- TextField: campo di testo dinamico o di input
- SimpleButton: un simbolo di pulsante
- Date: indicazione temporale precisa (data e ora)

Due termini che vengono spesso utilizzati come sinonimo di tipo di dati sono classe e oggetto. Una *classe* è semplicemente la definizione di un tipo di dati. È come un modello per tutti gli oggetti di un tipo di dati, come dire che "tutte le variabili del tipo di dati Esempio hanno queste caratteristiche: A, B e C". Un *oggetto*, per contro, è semplicemente un'istanza effettiva di una classe. Ad esempio, una variabile il cui tipo di dati è MovieClip può essere descritta come un oggetto MovieClip. Gli esempi seguenti sono modi diversi per esprimere lo stesso concetto:

- Il tipo di dati della variabile `myVariable` è Number.
- La variabile `myVariable` è un'istanza Number.
- La variabile `myVariable` è un oggetto Number.
- La variabile `myVariable` è un'istanza della classe Number.

Operazioni con gli oggetti

ActionScript è un linguaggio di programmazione orientato agli oggetti. La programmazione orientata agli oggetti è semplicemente un modello di programmazione, un modo di organizzare il codice per definire un programma che utilizza gli oggetti.

In precedenza il termine "programma per computer" è stato definito come una serie di istruzioni o procedure che il computer esegue. Concettualmente, quindi, potete immaginare che un programma è solo un lunghissimo elenco di istruzioni. In realtà, nella programmazione orientata agli oggetti, le istruzioni sono raggruppate in base all'oggetto a cui si riferiscono. Di conseguenza, il codice è organizzato in moduli funzionali dove le procedure correlate o i dati correlati sono raggruppati in un contenitore.

Adobe Flash Professional
I simboli utilizzati in Flash Professional sono degli oggetti. Supponete di aver definito un simbolo del clip filmato, ad esempio il disegno di un rettangolo, e di aver inserito una copia sullo stage. Il simbolo del clip filmato è anche, letteralmente, un oggetto di ActionScript; oltre a essere una istanza della classe MovieClip.

Sono diverse le caratteristiche del clip filmato che possono essere modificate. Dopo averlo selezionato, potete cambiare i valori nella finestra di ispezione Proprietà, cioè modificare la coordinata x o la larghezza. Potete anche regolare i colori, modificando il valore alpha (trasparenza) o applicare un filtro ombra esterna. Con altri strumenti di Flash Professional potete apportare ulteriori modifiche come, ad esempio, applicare una rotazione usando lo strumento Trasformazione libera. Tutti questi diversi modi in cui è possibile modificare un simbolo di clip filmato in Flash Professional sono disponibili anche in ActionScript. Potete modificare il clip filmato in ActionScript cambiando i frammenti di dati che vengono combinati nel pacchetto definito oggetto MovieClip.

Nella programmazione orientata agli oggetti di ActionScript, una classe può comprendere tre tipi di caratteristiche:

- Proprietà
- Metodi
- Eventi

Questi elementi permettono di manipolare i dati utilizzati dal programma e di definire le azioni da eseguire e la relativa sequenza.

Proprietà

Una proprietà rappresenta una porzione di dato che, combinata con altre, fa parte dell'oggetto. Ad esempio, un oggetto "song" potrebbe avere le proprietà `artist` e `title`; la classe MovieClip ha proprietà del tipo `rotation`, `x`, `width` e `alpha`. Le proprietà vengono utilizzate come se fossero variabili singole; in realtà, le proprietà possono essere considerate anche come semplici variabili secondarie di un oggetto.

Ecco alcuni esempi dell'impiego di proprietà nel codice ActionScript. Questa riga di codice sposta il clip filmato denominato `square` nella coordinata x di 100 pixel:

```
square.x = 100;
```

Questa riga di codice utilizza la proprietà rotation per imprimere al clip filmato `square` la stessa rotazione del clip filmato `triangle`:

```
square.rotation = triangle.rotation;
```

Questa riga di codice modifica la scala orizzontale del clip filmato `square` su un valore di una volta e mezzo superiore rispetto all'originale:

```
square.scaleX = 1.5;
```

Notate la struttura comune: si utilizza una variabile (`square`, `triangle`) come nome dell'oggetto, seguita da un punto (`.`) e il nome della proprietà (`x`, `rotation`, `scaleX`). Il punto, conosciuto come *operatore punto*, introduce un elemento secondario di un oggetto. L'intera struttura, "nome variabile-punto-nome proprietà", viene considerata un'unica variabile che dà il nome a un singolo valore della memoria del computer.

Metodi

Un *metodo* è un'azione che un oggetto può eseguire. Ad esempio, supponete di aver creato un simbolo di clip filmato in Flash Professional contenente vari fotogrammi e animazioni nella linea temporale. Potete riprodurre o interrompere il clip filmato oppure spostare l'indicatore di riproduzione su un particolare fotogramma.

Questa riga di codice indica di avviare la riproduzione del clip filmato denominato `shortFilm`:

```
shortFilm.play();
```

Questa riga di codice indica di interrompere la riproduzione del clip filmato denominato `shortFilm` (l'indicatore di riproduzione si ferma nel punto in cui si trova, come succede quando si preme il tasto pausa di un video):

```
shortFilm.stop();
```

Questa riga di codice indica di spostare l'indicatore di riproduzione al fotogramma 1 e di interrompere la riproduzione del clip filmato denominato `shortFilm` (come succede quando si riavvolge un video):

```
shortFilm.gotoAndStop(1);
```

I metodi si invocano proprio come le proprietà: si scrive il nome dell'oggetto (una variabile) seguito da un punto, poi si aggiunge il nome del metodo seguito da due parentesi. Le parentesi indicano la *chiamata* al metodo, vale a dire che si sta richiedendo all'oggetto di eseguire quella determinata azione. A volte le parentesi possono contenere dei valori o delle variabili che forniscono informazioni aggiuntive necessarie per eseguire l'azione. Questi valori aggiuntivi vengono definiti *parametri* del metodo. Ad esempio, per il metodo `gotoAndStop()` è necessario specificare il fotogramma dove l'indicatore di riproduzione si deve fermare, pertanto richiede la specifica di un parametro tra parentesi. Altri metodi, come `play()` e `stop()`, sono sufficientemente chiari e non richiedono informazioni supplementari, anche se le due parentesi del nome vengono mantenute.

Contrariamente alle proprietà e alle variabili, i metodi non sono utilizzati come segnaposto di valori. Tuttavia alcuni di essi possono eseguire dei calcoli e restituire un risultato che può essere utilizzato come una variabile. Per esempio, il metodo `toString()` della classe Number converte il valore numerico in una stringa di testo:

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

Ad esempio, potete utilizzare il metodo `toString()` per visualizzare il valore della variabile Number in un campo di testo dello schermo. La proprietà `text` della classe TextField è definita come un tipo String, pertanto può contenere solo valori di testo. (La `proprietà` text rappresenta il testo visualizzato sullo schermo). Questa riga di codice converte il valore numerico della variabile `numericData` in testo e lo visualizza sullo schermo nell'oggetto TextField denominato `calculatorDisplay`:

```
calculatorDisplay.text = numericData.toString();
```

Eventi

Un programma per computer è una serie di istruzioni che il computer esegue in sequenza. I programmi più semplici non sono nient'altro che una serie di procedure che il computer esegue e terminate le quali il programma finisce. I programmi creati con ActionScript, tuttavia, sono progettati per non interrompersi e attendere l'input da parte dell'utente o il verificarsi di altre condizioni. Gli eventi sono il meccanismo che definisce le istruzioni che il computer esegue e la relativa sequenza.

In sostanza, gli *eventi* sono condizioni che si verificano di cui ActionScript è consapevole e a cui è in grado di reagire. Molti eventi sono collegati all'interazione dell'utente, ad esempio la selezione di un pulsante o la pressione di un tasto della tastiera, ma ne esistono anche di altri tipi. Per esempio, se usate ActionScript per caricare un'immagine esterna, esiste un evento che può avvertire quando l'immagine è stata caricata completamente. Quando un programma ActionScript è in esecuzione, concettualmente resta in attesa del verificarsi di determinati eventi. Quando questi eventi si realizzano, esegue il codice ActionScript specificato per quegli eventi.

Principi di gestione degli eventi

Per *gestione degli eventi* si intende la definizione delle azioni da eseguire in risposta a particolari eventi. La scrittura di codice ActionScript per la gestione degli eventi presuppone l'identificazione di tre elementi importanti:

- L'origine dell'evento: su che oggetto si verificherà l'evento? Ad esempio, quale pulsante è stato selezionato o quale oggetto Loader carica l'immagine? L'origine dell'evento corrisponde anche al *target dell'evento*, in quanto si tratta dell'oggetto in cui il computer esegue l'azione (in cui l'evento si verifica).
- L'evento: qual è l'evento che si attende, l'evento a cui bisogna rispondere? È particolarmente importante identificare l'evento specifico, perché molti oggetti attivano eventi diversi.
- La risposta: che cosa deve succedere quando l'evento si realizza?

Il codice ActionScript per la gestione degli eventi deve sempre comprendere questi tre elementi e riprodurre questa struttura di base (gli elementi in grassetto sono segnaposti modificabili a seconda del caso specifico):

```
function eventResponse(eventObject:EventType):void
{
    // Actions performed in response to the event go here.
}

eventSource.addEventListener(EventType.EVENT_NAME, eventResponse);
```

Questo esempio di codice esegue due compiti. Per prima cosa, definisce una funzione, vale a dire specifica le azioni che devono avvenire in risposta all'evento. In secondo luogo, chiama il metodo `addEventListener()` dell'oggetto source, in pratica associando la funzione all'evento. Quando si verifica l'evento vengono attivate le azioni descritte nella funzione. Consideriamo ogni compito in maggiore dettaglio.

Una *funzione* rappresenta un modo per raggruppare varie azioni creando un'unica entità dotata di un nome proprio. Le funzioni sono quasi identiche ai metodi, con l'unica differenza che non sono necessariamente associate a una classe specifica; infatti, si potrebbe definire il termine "metodo" come una funzione associata a una classe specifica. Una funzione per la gestione di eventi deve avere obbligatoriamente un nome (in questo caso `eventResponse`) e un parametro (nel nostro esempio: `eventObject`). La specifica di un parametro per una funzione equivale alla dichiarazione di una variabile, di conseguenza, per il parametro è necessario definire il tipo di dati. In questo esempio, il tipo di dati del parametro è `EventType`.

A ogni tipo di evento che si desidera ascoltare è associata una classe ActionScript. Il tipo di dati specificato per il parametro funzione è sempre la classe associata dell'evento specifico a cui si desidera rispondere. Ad esempio, un evento `click` (attivato quando l'utente fa clic su un elemento con il mouse) è associato alla classe MouseEvent. Per scrivere una funzione di listener per un evento `click`, dovete definire la funzione di listener con un parametro con associato il tipo di dati MouseEvent. Infine, tra le parentesi graffe (`{ ... }`) scrivete le procedure che il computer deve eseguire quando l'evento si verifica.

Una volta scritta la funzione di gestione degli eventi, dovete comunicare all'oggetto di origine dell'evento (l'oggetto su cui si verifica l'evento, ad esempio, il pulsante) che la funzione deve essere chiamata nel momento in cui l'evento si verifica. Per registrare la funzione con l'oggetto di origine dell'evento, chiamate il metodo `addEventListener()` dell'oggetto, infatti, tutti gli oggetti associati ad eventi sono associati anche al metodo `addEventListener()`. Il metodo `addEventListener()` accetta due parametri:

- Prima di tutto, il nome dell'evento specifico a cui rispondere. Ogni evento è collegato a una classe specifica. Ogni classe event ha un valore speciale, che è come un nome univoco, definito per ciascuno dei suoi eventi e che utilizzate come primo parametro.
- Secondariamente, il nome della funzione di risposta all'evento. Si noti che i nomi di funzione passati come parametri vanno scritti senza parentesi.

Processo di gestione degli eventi

Segue una descrizione dettagliata del processo che si mette in atto quando si crea un listener di eventi. L'esempio illustra la creazione di una funzione di listener da chiamare quando l'utente fa clic sull'oggetto `myButton`.

Il codice scritto dal programmatore è il seguente:

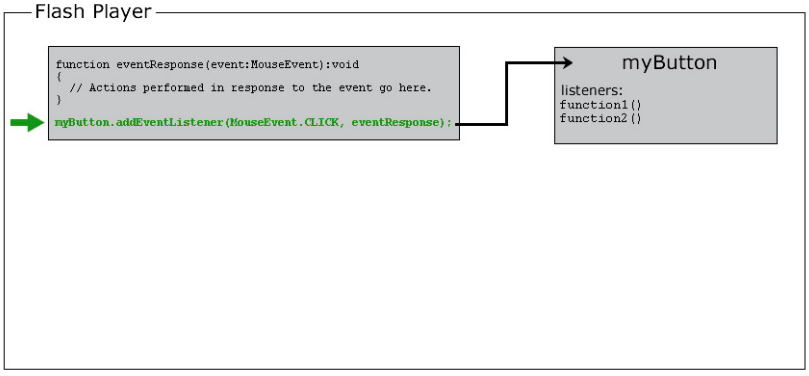
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

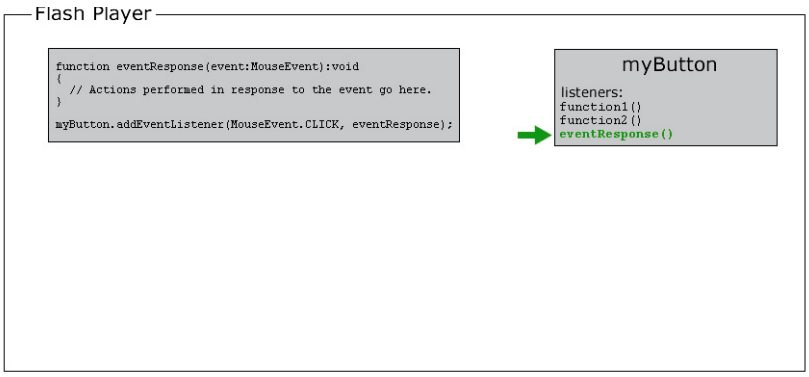
Ecco cosa succede quando il codice viene eseguito:

1 Nel momento in cui carica il file SWF, il computer registra l'esistenza della funzione `eventResponse()`.

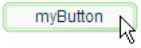
2 Il computer, allora, esegue il codice (e in particolare le linee del codice che non sono presenti in una funzione). In questo caso è presente una sola riga di codice: chiamata al metodo `addEventListener()` nell'oggetto di origine dell'evento (denominato `myButton`) e passaggio della funzione `eventResponse` come parametro.



Internamente, `myButton` mantiene un elenco di funzioni che intercettano ciascuno di questi eventi. Quando viene chiamato il metodo `addEventListener()`, `myButton` registra la funzione `eventResponse()` tra i suoi listener di eventi.

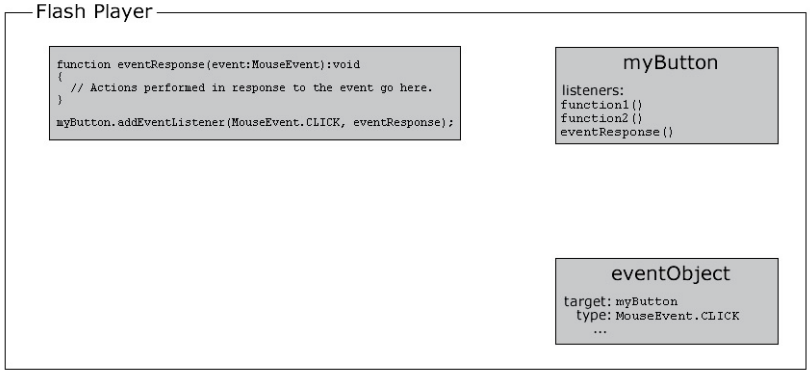


3 Quando l'utente fa clic sull'oggetto `myButton`, viene attivato l'evento `click` (definito nel codice come `MouseEvent.CLICK`).

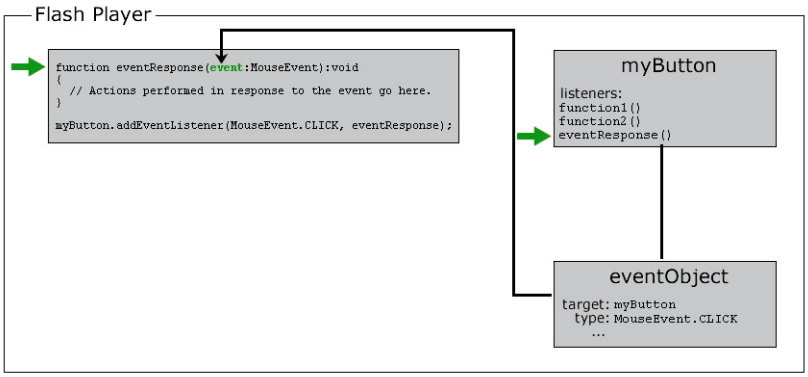


A questo punto, si verifica quanto segue:

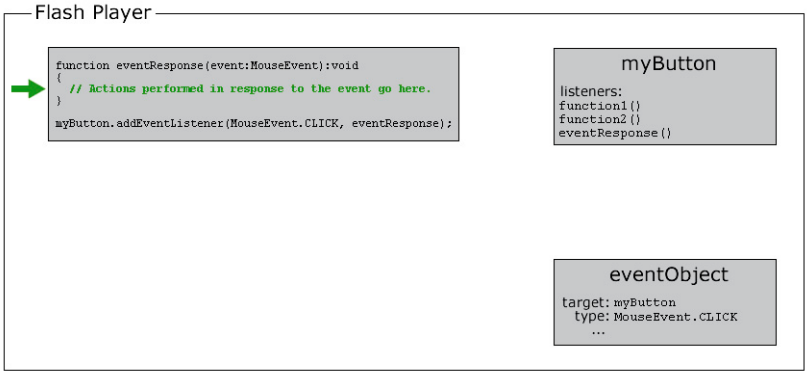
a Viene creato un oggetto che è un'istanza della classe associata all'evento in questione (MouseEvent in questo esempio). Per molti eventi, questo oggetto è un'istanza della classe Event. Per gli eventi del mouse è un'istanza MouseEvent. Per altri eventi, è un'istanza della classe associata a tale evento. L'oggetto creato è noto come *oggetto evento* e contiene informazioni specifiche sull'evento: tipo di evento, posizione in cui si è verificato e altre informazioni specifiche, se disponibili.



b Il computer esamina l'elenco di listener di eventi registrati in `myButton` e chiama ogni singola funzione passando l'oggetto evento come parametro. Dal momento che la funzione `eventResponse()` è uno dei listener di `myButton`, come parte del processo il computer chiama la funzione `eventResponse()`.



c Nel momento in cui viene chiamata la funzione `eventResponse()`, il suo codice viene eseguito e le azioni specificate vengono attivate.



Esempi di gestione degli eventi

Seguono alcuni esempi concreti di codice di gestione degli eventi per dare un'idea degli elementi evento più comuni e delle possibili variazioni disponibili per scrivere codice per la gestione di eventi:

- Selezionare un pulsante per avviare la riproduzione del clip filmato corrente. Nell'esempio che segue, `playButton` è il nome istanza del pulsante e `this` è un nome speciale che significa "l'oggetto corrente":

```
this.stop();

function playMovie(event:MouseEvent):void
{
    this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- Rilevare l'inserimento di testo in un campo di testo. Nell'esempio che segue, `entryText` è un campo di testo di input e `outputText` è un campo di testo dinamico:

```
function updateOutput(event:TextEvent):void
{
    var pressedKey:String = event.text;
    outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- Selezionare un pulsante per accedere a un URL. Nell'esempio che segue, `linkButton` è il nome di istanza del pulsante:

```
function gotoAdobeSite(event:MouseEvent):void
{
    var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
    navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

Creazione di istanze di oggetti

Per poter utilizzare un oggetto in ActionScript, l'oggetto deve già essere stato creato. Il primo passo nella creazione di un oggetto è la dichiarazione di una variabile, vale a dire, l'identificazione di una porzione di memoria del computer destinata a contenere dei dati. In seguito, questo spazio vuoto va riempito assegnando un valore alla variabile, in pratica creando un oggetto e memorizzandolo nella variabile in modo che possa diventare utilizzabile. Il processo di creazione di un oggetto corrisponde alla *creazione di un'istanza* dell'oggetto. In altre parole, viene creata un'istanza di una particolare classe.

Esiste un modo semplice per creare un'istanza di un oggetto che non prevede l'utilizzo di ActionScript. In Flash Professional, inserite un simbolo di clip filmato, un simbolo di pulsante o un campo di testo sullo stage e ad esso assegnate un nome di istanza. Flash Professional dichiara automaticamente una variabile usando quel nome di istanza, crea un'istanza dell'oggetto e memorizza l'oggetto nella variabile. Analogamente, in Flex create un componente in MXML codificando un tag MXML o inserendo il componente nell'editor in modalità Flash Builder Design. Quando assegnate un ID a questo componente, tale ID diventa il nome di una variabile ActionScript contenente l'istanza del componente.

Tuttavia, non sempre vorrete creare un oggetto graficamente, inoltre per gli oggetti non visuali questo non è possibile. Esistono altri modi per creare istanze dell'oggetto utilizzando solo ActionScript.

Molti tipi di dati di ActionScript consentono di creare istanze usando un'*espressione letterale*, vale a dire, un valore scritto direttamente in linguaggio ActionScript. Di seguito sono forniti alcuni esempi.

- Valore numerico letterale (inserire il numero direttamente):

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- Valore String letterale (racchiudere il testo tra virgolette):

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- Valore Boolean letterale (usare i valori letterali `true` o `false`):

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- Valore Array letterale (racchiudere un elenco di valori separati da virgola tra parentesi quadre):

```
var seasons:Array = ["spring", "summer", "autumn", "winter"];
```

- Valore XML letterale (inserire XML direttamente):

```
var employee:XML = <employee>
        <firstName>Harold</firstName>
        <lastName>Webster</lastName>
    </employee>;
```

ActionScript definisce espressioni letterali anche per i tipi di dati Array, RegExp, Object e Function.

Il modo più comune per creare un'istanza di un tipo dati è utilizzare l'operatore `new` con il nome della classe, come mostrato di seguito:

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

La creazione di un oggetto mediante l'operatore `new` è comunemente descritta con l'espressione "chiamare la funzione di costruzione della classe". La *funzione di costruzione* è uno speciale metodo chiamato come parte del processo di creazione di un'istanza di una classe. Notate che per creare un'istanza in questo modo, il nome della classe deve essere seguito dalle parentesi e può essere necessario specificare dei parametri, proprio come accade quando si chiama un metodo.

Utilizzate l'operatore `new` per creare un'istanza dell'oggetto anche per i tipi di dati che consentono di creare istanze tramite espressioni letterali. Queste due righe di codice, ad esempio, generano lo stesso risultato:

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

È importante avere dimestichezza con il metodo `new`*ClassName*`()` per la creazione degli oggetti Molti tipi di dati di ActionScript non hanno una rappresentazione visiva, per cui non possono essere creati inserendo un elemento sullo stage di Flash Professional né utilizzando la modalità Design dell'editor MXML di Flash Builder. L'unica opzione di cui disponete è creare un'istanza di questi tipi di dati in ActionScript usando l'operatore `new`.

Adobe Flash Professional
In particolare, Flash Professional permette di usare l'operatore `new` per creare un'istanza di un simbolo di clip filmato definito nella Libreria ma non inserito sullo stage.

Altri argomenti presenti nell'Aiuto

Operazioni con gli array

Uso delle espressioni regolari

Creazione di oggetti MovieClip mediante ActionScript

Elementi comuni dei programmi

Esistono altri elementi fondamentali per la creazione dei programmi ActionScript.

Operatori

Gli *operatori* sono simboli speciali (o, più raramente, parole) che permettono di eseguire dei calcoli. Vengono utilizzati prevalentemente per operazioni aritmetiche e per comparare fra loro dei valori. In genere, un operatore utilizza uno o più valori per estrapolare un risultato. Ad esempio:

- L'operatore di addizione (`+`) somma tra loro più valori e restituisce un numero come risultato:

  ```
  var sum:Number = 23 + 32;
  ```

- L'operatore di moltiplicazione (`*`) moltiplica un valore per un altro e restituisce un numero come risultato:

  ```
  var energy:Number = mass * speedOfLight * speedOfLight;
  ```

- L'operatore di uguaglianza (`==`) confronta due valori per stabilire se sono uguali e restituisce un valore booleano (true o false):

  ```
  if (dayOfWeek == "Wednesday")
  {
      takeOutTrash();
  }
  ```

  Come si può evincere da questi esempi, l'operatore di uguaglianza e in generale gli operatori di "confronto", vengono utilizzati prevalentemente con l'istruzione `if` per determinare se le procedure che seguono devono essere eseguite o meno.

Commenti

Durante la scrittura del codice ActionScript può essere utile aggiungere delle note per spiegare il funzionamento di determinate righe o annotare il motivo di una particolare scelta. I *commenti del codice* sono uno strumento che permette di aggiungere testo che viene ignorato dal computer. Nel codice ActionScript si possono aggiungere due tipi di commenti:

- Commento a riga singola: per indicare che una riga è un commento, inserire due barre in qualsiasi punto della riga. Quanto segue le due barre viene ignorato dal computer:

  ```
  // This is a comment; it's ignored by the computer.
  var age:Number = 10; // Set the age to 10 by default.
  ```

- Commenti su più righe: un commento che occupa più righe comprende un indicatore di inizio (`/*`), il commento vero e proprio seguito da un indicatore di fine (`*/`). Tutto il testo scritto tra i due indicatori viene ignorato dal computer, indipendentemente dal numero di righe occupate:

  ```
  /*
  This is a long description explaining what a particular
  function is used for or explaining a section of code.

  In any case, the computer ignores these lines.
  */
  ```

I commenti vengono spesso utilizzati anche per un altro scopo: disabilitare temporaneamente una o più righe di codice per testare, ad esempio, soluzioni alternative o individuare il motivo per cui il codice ActionScript non funziona come previsto.

Controllo del flusso

Spesso, all'interno di un programma, si presenta la necessità di ripetere determinate azioni, di eseguire solo certe azioni e non altre o di eseguire azioni alternative a seconda delle condizioni. Il *controllo del flusso* è il meccanismo che gestisce quali azioni devono essere eseguite. In ActionScript sono disponibili vari tipi di elementi di controllo del flusso.

- Funzioni: le funzioni agiscono come scorciatoie: raggruppano una serie di azioni in un'unica entità e consentono di eseguire dei calcoli. Sono importanti per la gestione di eventi ma anche come strumento per raggruppare serie di istruzioni.
- Cicli: i cicli sono costrutti che permettono di eseguire ciclicamente un gruppo di istruzioni per un determinato numero di volte o fino al verificarsi di una determinata condizione. I cicli consentono di manipolare vari elementi collegati tra loro tramite l'uso di una variabile il cui valore cambia a ogni iterazione.
- Istruzioni condizionali: le istruzioni condizionali consentono di definire procedure da eseguire solo in alcune circostanze o di presentare pezzi di codice alternativi da eseguire a seconda delle condizioni che si verificano. L'istruzione condizionale più usata è la struttura `if`. `if` permette di verificare un valore o un'espressione tra parentesi. Se il valore è `true` vengono eseguite le righe di codice racchiuse tra parentesi graffe. In caso contrario, vengono ignorate. Ad esempio:

```
if (age < 20)
{
    // show special teenager-targeted content
}
```

L'istruzione generalmente associata a `if` è `else`, la quale indica le procedure alternative eseguite dal computer se la condizione non è `true`:

```
if (username == "admin")
{
    // do some administrator-only things, like showing extra options
}
else
{
    // do some non-administrator things
}
```

Esempio: Animation portfolio piece (Flash Professional)

Questo esempio ha lo scopo di offrirvi una prima opportunità di scoprire come è possibile unire parti di ActionScript in un'applicazione ActionScript completa. La porzione del portfolio animazione è un esempio di come sia possibile modificare un'animazione lineare esistente, aggiungendovi alcuni elementi interattivi minori. Ad esempio, potete incorporare un'animazione creata per un cliente in un portfolio in linea. Il comportamento interattivo che verrà aggiunto all'animazione include due pulsanti selezionabili dall'utente: uno per avviare l'animazione e uno per spostarsi su un URL separato (ad esempio il menu portfolio o la home page dell'autore).

Il processo di creazione di questa porzione può essere suddiviso nelle seguenti fasi principali:

1. Preparazione del file FLA per l'aggiunta di elementi ActionScript e interattivi
2. Creazione e aggiunta di pulsanti
3. Scrittura del codice ActionScript
4. Provare l'applicazione.

Preparazione per l'aggiunta di elementi interattivi

Prima di aggiungere elementi interattivi all'animazione, è utile configurare il file FLA creando alcuni spazi in cui aggiungere il nuovo contenuto. Questa operazione include la creazione di spazio effettivo sullo stage in cui poter posizionare i pulsanti e anche la creazione di "spazio" nel file FLA per mantenere separati gli elementi diversi.

Per configurare il file FLA per l'aggiunta di elementi interattivi:

1. Create un file FLA con un'animazione semplice, ad esempio un'unica interpolazione di movimento o di forma. Se disponete già di un file FLA contenente l'animazione da utilizzare nel progetto, apritelo e salvatelo con un nuovo nome.
2. Decidete la posizione in cui devono essere visualizzati i due pulsanti: uno per avviare l'animazione e l'altro per collegarsi al portfolio o alla home page dell'autore. Se necessario, liberate o aggiungete spazio sullo stage per il nuovo contenuto. Se l'animazione non è dotata di una finestra di avvio, potete crearne una nel primo fotogramma. In tal caso, potete anche spostare l'animazione in modo che venga avviata nel fotogramma 2 o in uno successivo.
3. Aggiungete un nuovo livello sopra gli altri livelli nella linea temporale e denominatelo **buttons**. Questo è il livello in cui verranno aggiunti i pulsanti.
4. Aggiungete un nuovo livello sopra il livello buttons e denominarlo **actions**. Questo è il livello in cui verrà aggiunto all'applicazione il codice ActionScript.

Creazione e aggiunta dei pulsanti

A questo punto, è necessario creare e posizionare i pulsanti che costituiscono il fulcro dell'applicazione interattiva.

Per creare e aggiungere pulsanti al file FLA:

1. Mediante gli strumenti di disegno, create l'aspetto visivo del primo pulsante (il pulsante "play") nel livello buttons. Ad esempio, disegnate un'ovale orizzontale contenente del testo.
2. Mediante lo strumento Selezione, selezionate tutte le parti grafiche del pulsante.
3. Nel menu principale, scegliete Elabora > Converti in simbolo.
4. Nella finestra di dialogo, scegliete Pulsante come tipo di simbolo, assegnate un nome al simbolo e fare clic su OK.
5. Con il pulsante selezionato, nella finestra di ispezione Proprietà assegnate al pulsante il nome di istanza **playButton**.
6. Ripetete i passaggi da 1 a 5 per creare il pulsante che collega alla home page dell'autore. Denominate questo pulsante **homeButton**.

Scrittura del codice

Il codice ActionScript per questa applicazione può essere suddiviso in tre set di funzionalità, sebbene vengano inserite tutte nella stessa posizione. Le tre azioni che il codice esegue sono le seguenti:

- Interrompere l'indicatore di riproduzione non appena il file SWF viene caricato (quando l'indicatore di riproduzione raggiunge il fotogramma 1).
- Attendere un evento per avviare la riproduzione del file SWF quando l'utente fa clic sul pulsante play.
- Attendere un evento per indirizzare il browser all'URL appropriato quando l'utente fa clic sul pulsante della home page dell'autore.

Per creare il codice che interrompe l'indicatore di riproduzione quando raggiunge il fotogramma 1:

1. Selezionate il fotogramma chiave nel fotogramma 1 del livello actions.

2 Per aprire il pannello Azioni, dal menu principale scegliere Finestra > Azioni.

3 Nel riquadro dello script, immettete il codice seguente:

```
stop();
```

Per scrivere il codice che avvia l'animazione quando si fa clic sul pulsante play:

1 Al termine del codice inserito nei punti precedenti, aggiungete due righe vuote.

2 Inserite il codice seguente alla fine dello script:

```
function startMovie(event:MouseEvent):void
{
    this.play();
}
```

Questo codice definisce una funzione di nome `startMovie()`. Quando la funzione `startMovie()` viene chiamata, viene avviata la riproduzione della linea temporale principale.

3 Nella riga successiva al codice aggiunto al punto precedente, inserite la riga di codice seguente:

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

Questa riga di codice registra la funzione `startMovie()` come listener per l'evento `click` di `playButton`. In altre parole, fa in modo che quando si fa clic sul pulsante di nome `playButton`, venga chiamata la funzione `startMovie()`.

Per scrivere codice per indirizzare il browser a un URL quando si fa clic sul pulsante home page:

1 Al termine del codice inserito nei punti precedenti, aggiungete due righe vuote.

2 Inserite il codice seguente alla fine dello script:

```
function gotoAuthorPage(event:MouseEvent):void
{
    var targetURL:URLRequest = new URLRequest("http://example.com/");
    navigateToURL(targetURL);
}
```

Questo codice definisce una funzione di nome `gotoAuthorPage()`. Questa funzione prima crea un'istanza URLRequest che rappresenta l'URL http://example.com/, quindi passa questo URL alla funzione `navigateToURL()`, che determina l'apertura dell'URL nel browser.

3 Nella riga successiva al codice aggiunto al punto precedente, inserire la riga di codice seguente:

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

Questa riga di codice registra la funzione `gotoAuthorPage()` come listener per l'evento `click` di `homeButton`. In altre parole, fa in modo che quando si fa clic sul pulsante di nome `homeButton`, venga chiamata la funzione `gotoAuthorPage()`.

Prova dell'applicazione

L'applicazione è ora completamente funzionante. Proviamo il codice per essere certi che sia proprio così.

Per provare l'applicazione:

1 Nel menu principale, scegliete Controllo > Prova filmato. Flash Professional crea il file SWF e lo apre in una finestra di Flash Player.

2 Provate entrambi i pulsanti per verificare che eseguano le funzioni impostate.

3 Se i pulsanti non funzionano, controllate le seguenti condizioni:

- I pulsanti hanno nomi di istanze distinti?
- Per le chiamate al metodo `addEventListener()` vengono utilizzati gli stessi nomi delle istanze dei pulsanti?
- I nomi evento utilizzati nelle chiamate al metodo `addEventListener()` sono corretti?
- Il parametro specificato per ciascuna funzione è corretto? (Entrambi i metodi richiedono un singolo parametro con tipo di dati MouseEvent.)

Tutti questi errori, e molti degli altri errori possibili, generano un messaggio di errore. Il messaggio di errore può essere visualizzato quando scegliete il comando Prova filmato o quando fate clic sul pulsante durante la prova del progetto. Cercate nel pannello Errori del compilatore eventuali errori del compilatore (quelli che si verificano quando scegliete Prova filmato per la prima volta) e nel pannello Output eventuali errori di runtime (errori che si verificano mentre la riproduzione del file SWF è in corso, ad esempio quando fate clic su un pulsante).

Creazione di applicazioni con ActionScript

Per il processo di creazione di applicazioni tramite il codice ActionScript non è sufficiente conoscere la sintassi e i nomi delle classi. La maggior parte della documentazione della piattaforma Flash copre questi due argomenti (sintassi e utilizzo delle classi ActionScript). Tuttavia, per creare un'applicazione ActionScript è anche opportuno conoscere le informazioni seguenti:

- Quali programmi usare per scrivere codice ActionScript
- Come organizzare il codice ActionScript
- Come inserire il codice ActionScript in un'applicazione
- Fasi che compongono lo sviluppo di un'applicazione ActionScript

Opzioni per l'organizzazione del codice

Il codice ActionScript 3.0 può essere utilizzato come "motore" per ogni tipo di applicazione, da semplici animazioni grafiche a sofisticati sistemi di elaborazione delle transazioni client/server. A seconda del tipo di applicazione che intendete creare, adottate uno o più di questi metodi per inserire il codice ActionScript nel vostro progetto.

Memorizzazione del codice nei fotogrammi di una linea temporale di Flash Professional

In Flash Professional, potete aggiungere codice ActionScript a qualunque fotogramma della linea temporale. Il codice viene eseguito durante la riproduzione del filmato, nel momento in cui l'indicatore di riproduzione arriva al fotogramma.

L'inserimento di codice ActionScript nei fotogrammi rappresenta un modo semplice per aggiungere comportamenti alle applicazioni realizzate in Flash Professional. È possibile aggiungere codice a qualunque fotogramma della linea temporale principale o della linea temporale di qualsiasi simbolo MovieClip. Tuttavia, questa flessibilità ha un prezzo. Quando si creano applicazioni di grandi dimensioni, diventa difficile ricordare quali script sono stati inseriti in quali fotogrammi, di conseguenza l'applicazione può diventare complicata da gestire con il passare del tempo.

Molti sviluppatori scelgono di semplificare l'organizzazione del codice ActionScript usato in Flash Professional inserendolo solo nel primo fotogramma di una linea temporale oppure su un livello specifico del documento Flash. La separazione del codice semplifica l'individuazione e la gestione del codice contenuto nei file FLA di Flash. Tuttavia, lo stesso codice non può essere utilizzato in un altro progetto Flash Professional senza copiare e incollare il codice nel nuovo file.

Per semplificare l'utilizzo del codice ActionScript in altri progetti Flash Professional futuri, memorizzate il codice in file ActionScript esterni (file di testo con estensione .as).

Incorporazione del codice nei file MXML di Flex

In un ambiente di sviluppo Flex come Flash Builder, potete includere codice ActionScript all'interno di un tag `<fx:Script>` in un file MXML di Flex. Questa tecnica può, tuttavia, aggiungere complessità a progetti di grandi dimensioni e rendere più difficile utilizzare lo stesso codice in un altro progetto Flex. Per semplificare l'utilizzo di codice ActionScript in altri progetti Flex futuri, memorizzate il codice in file ActionScript esterni.

***Nota:** potete specificare un parametro source per un tag `<fx:Script>`. L'uso di un parametro source consente di "importare" codice ActionScript da un file esterno come se fosse digitato direttamente nel tag `<fx:Script>`. Tuttavia, il file di origine utilizzato non può definire la propria classe; questo ne limita la riutilizzabilità.*

Memorizzazione del codice in file ActionScript

Se un progetto prevede un uso intensivo di codice ActionScript, il modo migliore per organizzarlo è inserirlo in file di origine distinti con l'estensione .as. I file ActionScript possono essere strutturati in due modi diversi, a seconda di come si intenda utilizzarli all'interno dell'applicazione.

- Codice ActionScript non strutturato: righe di codice ActionScript, istruzioni e definizioni di funzioni comprese, scritte come se fossero immesse direttamente nello script della linea temporale o nel file MXML.

  Potete accedere a questo tipo di codice usando l'istruzione `include` di ActionScript o il tag `<fx:Script>` in un file MXML di Flex. L'istruzione `include` di ActionScript indica al compilatore di inserire il contenuto di un file ActionScript esterno in una posizione specifica e con un'area di validità specifica all'interno di uno script. Il risultato finale è identico a quello che si otterrebbe se il codice venisse inserito direttamente in tale posizione. Nel linguaggio MXML, l'utilizzo di un tag `<fx:Script>` con un attributo source identifica un codice ActionScript esterno caricato dal compilatore in quel punto dell'applicazione. Ad esempio, il tag seguente carica il file ActionScript esterno Box.as:

  ```
  <fx:Script source="Box.as" />
  ```

- Definizione classe ActionScript: definizione di una classe ActionScript, che comprende le definizioni dei metodi e delle proprietà.

  Quando definite una classe potete accedere al codice ActionScript nella classe creando un'istanza della classe e utilizzando le proprietà, i metodi e gli eventi corrispondenti. Utilizzare classi personali non è diverso dall'utilizzare qualsiasi classe ActionScript incorporata e richiede due parti:

  - Usare l'istruzione `import` per specificare il nome completo della classe affinché il compilatore ActionScript sappia dove trovarla. Ad esempio, per utilizzare la classe MovieClip in ActionScript, importate la classe specificando il nome completo, costituito da pacchetto e classe:

    ```
    import flash.display.MovieClip;
    ```

    In alternativa, si può importare il pacchetto che contiene la classe MovieClip, il che equivale a scrivere un'istruzione `import` separata per ogni classe del pacchetto:

    ```
    import flash.display.*;
    ```

    Si sottraggono a questa regola le classi di primo livello, che non sono definite all'interno del pacchetto.

  - Scrittura di codice che usa esplicitamente il nome classe. Ad esempio, dichiarate un variabile con la classe come tipo di dati e create un'istanza della classe da memorizzare nella variabile. L'uso di una classe nel codice ActionScript indica al compilatore di caricare la definizione di tale classe. Ad esempio, data una classe esterna chiamata Box, l'istruzione seguente crea una nuova istanza di tale classe:

    ```
    var smallBox:Box = new Box(10,20);
    ```

Quando incontra per la prima volta il riferimento alla classe Box, il compilatore cerca nel codice sorgente disponibile la definizione della classe Box.

Scelta dello strumento adeguato

Potete utilizzare diversi strumenti (o più strumenti insieme) per scrivere e modificare il codice ActionScript.

Flash Builder

Adobe Flash Builder è lo strumento privilegiato per la creazione di progetti con il framework Flex o progetti costituiti principalmente da codice ActionScript. Flash Builder include anche un editor ActionScript completamente funzionale nonché un layout visuale e funzionalità di modifica MXML. Può essere usato per creare progetti Flex o ActionScript puri. Le applicazioni Flex offrono svariati vantaggi tra cui: una ricca gamma di componenti dell'interfaccia utente prestabiliti, controlli flessibili per il layout dinamico e meccanismi incorporati per l'interazione con origini dati esterne e il collegamento di dati esterni agli elementi dell'interfaccia utente. Tuttavia, per via del codice supplementare necessario per attivare queste funzioni, i progetti che usano Flex generano file SWF di dimensioni superiori rispetto alle controparti non Flex.

Si consiglia di usare Flash Builder per creare con Flex applicazioni Internet sofisticate, basate su dati e interattive, per modificare il codice ActionScript e MXML e per creare un'anteprima visiva dell'applicazione, il tutto mediante un unico strumento.

Molti utenti Flash Professional che creano progetti ActionScript densi di codice usano Flash Professional per creare elementi visivi e Flash Builder come un editor per il codice ActionScript.

Flash Professional

Oltre alle funzionalità grafiche e di animazione, Flash Professional offre strumenti per utilizzare il codice ActionScript. Tale codice può essere allegato a elementi di un file FLA o in file ActionScript puri esterni. Flash Professional è ideale per i progetti che prevedono un uso intensivo di animazioni e video o quelli in cui desiderate creare personalmente la maggior parte delle risorse grafiche. Un altro motivo per utilizzare Flash Professional per sviluppare il progetto ActionScript è creare elementi visivi e scrivere codice nella stessa applicazione. Flash Professional include anche componenti dell'interfaccia utente prestabiliti che potete utilizzare per creare file SWF più piccoli, associandoli al progetto mediante strumenti visivi.

Flash Professional offre due strumenti per la scrittura del codice ActionScript:

- Pannello Azioni: pannello disponibile per la gestione dei file FLA e che consente di scrivere codice ActionScript e di associarlo a fotogrammi della linea temporale.
- Finestra di script: editor di testo dedicato ai file di codice ActionScript (.as).

Editor di ActionScript esterno

Dal momento che i file ActionScript (.as) sono semplici file di testo, è possibile scrivere file ActionScript usando un qualsiasi editor di testo. Oltre ai prodotti specifici per ActionScript di Adobe, sono disponibili vari editor di testo esterni dotati di funzionalità specifiche per ActionScript. Potete scrivere file MXML e classi ActionScript usando un qualsiasi editor di testo. Potete anche creare un'applicazione da questi file utilizzando l'SDK Flex. Il progetto può utilizzare Flex o essere un'applicazione ActionScript pura. In alternativa, alcuni sviluppatori scrivono le classi ActionScript con Flash Builder o un editor ActionScript di terze parti e usano Flash Professional per creare il contenuto grafico.

I diversi motivi per scegliere un editor ActionScript esterno includono:

- Preferite scrivere codice in un programma separato e progettare elementi visivi in Flash Professional.

- Usate un'applicazione per la programmazione di codice non ActionScript, come la creazione di pagine HTML o lo sviluppo di applicazioni con un linguaggio di programmazione diverso, e desiderate usare quella stessa applicazione anche per scrivere codice ActionScript.
- Volete creare dei progetti Flex o ActionScript puri tramite l'SDK Flex senza ricorrere a Flash Professional o Flash Builder.

Alcuni dei migliori editor di testo con supporto specifico per ActionScript sono:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (con pacchetti ActionScript e Flex)

Processo di sviluppo ActionScript

Indipendentemente dalle dimensioni del progetto ActionScript, definire un processo di progettazione e sviluppo dell'applicazione aiuta a lavorare in modo più produttivo ed efficace. Ecco le fasi fondamentali che deve comprendere un processo di sviluppo di un'applicazione tramite ActionScript 3.0:

1. Progettate l'applicazione.

   Descrivete l'applicazione prima di iniziare a crearla.

2. Componete il codice ActionScript 3.0.

   Potete creare il codice ActionScript utilizzando Flash Professional, Flex Builder, Dreamweaver o un editor di testo.

3. Create un progetto Flash o Flex per l'esecuzione del codice.

   In Flash Professional, create un file FLA, configurate le impostazioni di pubblicazione, aggiungete componenti dell'interfaccia utente all'applicazione e fate riferimento al codice ActionScript. In Flex, definite l'applicazione, aggiungete componenti dell'interfaccia utente utilizzando MXML e fate riferimento al codice ActionScript.

4. Pubblicate e provate l'applicazione ActionScript.

   Provare l'applicazione significa eseguirla all'interno dell'ambiente di sviluppo e verificare che il comportamento sia quello previsto.

Non è indispensabile eseguire questi passaggi nell'ordine in cui sono presentati né completare una fase prima di passare alla successiva. Ad esempio, potete progettare una schermata dell'applicazione (punto 1), quindi creare la grafica, i pulsanti e così via (punto 3), poi scrivere il codice ActionScript (punto 2) e infine provare l'applicazione (punto 4). In alternativa, potete progettare una parte dell'applicazione, aggiungere un pulsante o un elemento dell'interfaccia per volta, scrivere il codice ActionScript per ogni elemento e provare il risultato dopo averlo generato. Nonostante sia importante avere ben chiare queste quattro fasi del processo di sviluppo, nel concreto è spesso più utile spostarsi da una all'altra in base alla necessità del momento.

Creazione di classi personalizzate

Il processo di creazione delle classi da utilizzare nei progetti può incutere un certo timore. Tuttavia, la parte più impegnativa del processo è la progettazione dei metodi, delle proprietà e degli eventi della classe.

Strategie per la progettazione di una classe

La progettazione orientata agli oggetti è una disciplina molto complessa, a cui studiosi hanno consacrato intere carriere accademiche e professionali. Tuttavia, di seguito proponiamo alcuni approcci che consentono di orientarsi.

1. Considerate il ruolo che le istanze di questa classe svolgono all'interno dell'applicazione. Generalmente, gli oggetti espletano una di queste tre funzioni:
   - Oggetto Value: si tratta di oggetti destinati a contenere dei valori e sono caratterizzati da varie proprietà e pochi metodi (a volte nessun metodo). Sono spesso rappresentazioni in codice di elementi chiaramente definiti. Ad esempio, un'applicazione di riproduzione musicale può includere una classe Song che rappresenta una canzone e una classe Playlist che rappresenta un gruppo di canzoni.
   - Oggetto di visualizzazione: oggetti che vengono visualizzati sullo schermo, ad esempio elementi dell'interfaccia utente quali elenchi a discesa o comunicazioni dello stato, elementi grafici quali creature di un video game e così via.
   - Struttura applicazione: oggetti che espletano vari ruoli di supporto nell'ambito della logica e dell'elaborazione eseguita dalle applicazioni. Ad esempio, potete fare in modo che un oggetto esegua determinati calcoli in una simulazione biologica. Potete rendere un oggetto responsabile della sincronizzazione di valori tra il comando del quadrante e il valore del volume in un'applicazione di riproduzione musicale. Un altro oggetto può gestire le regole di un videogioco. Oppure potete fare in modo che una classe carichi un'immagine salvata in un'applicazione di disegno.
2. Scegliete la funzionalità che deve svolgere la classe. I vari tipi di funzionalità spesso diventano i metodi della classe.
3. Se la classe deve fungere da oggetto contenitore di valori, scegliete i dati che le istanze conterranno. Queste voci sono buoni candidati da convertire in proprietà.
4. Poiché la classe viene progettata espressamente per un progetto, quello che più conta è assegnarle la funzionalità che l'applicazione richiede. Provate a rispondere alle domande seguenti:
   - Che tipo di informazioni vengono memorizzate, monitorate e manipolate dall'applicazione? La risposta a questa domanda consente di individuare gli oggetti contenitore e le proprietà necessari.
   - Quali set di azioni esegue l'applicazione? Cosa accade, ad esempio, quando si carica l'applicazione, quando si seleziona un pulsante specifico, quando si interrompe la riproduzione di un filmato e così via? Questi sono candidati eccellenti per i metodi o per le proprietà se le "azioni" prevedono la modifica di valori singoli.
   - Per eseguire ogni singola azione, quali sono le informazioni necessarie? Questi dati diventeranno i parametri del metodo.
   - Durante l'esecuzione dell'applicazione, quali cambiamenti che si verificano nella classe devono essere comunicati alle altre parti dell'applicazione? Questi sono ottimi candidati per gli eventi.
5. Esiste un oggetto simile a quello che desiderate creare ma a cui mancano alcune funzionalità che desiderate aggiungere? Valutate la possibilità di creare una sottoclasse, cioè una classe che amplia la funzionalità di una classe esistente, anziché creare una classe completamente nuova. Ad esempio, per creare una classe che funge da oggetto visivo dello schermo, utilizzate come base il comportamento di un oggetto di visualizzazione esistente. In questo caso, l'oggetto di visualizzazione (ad esempio, MovieClip o Sprite) è la *classe base* la cui funzionalità viene ampliata nella nuova classe.

Scrittura del codice per una classe

Una volta progettata la classe, o perlomeno, una volta stabilite le informazioni che deve memorizzare e le azioni che deve eseguire, la sintassi da usare per scrivere il codice della classe è abbastanza semplice.

Ecco i passaggi fondamentali da seguire per creare una classe ActionScript:

1 Aprite un nuovo documento di testo nell'editor di testo di ActionScript.

2 Immettete un'istruzione `class` per definire il nome della classe Per aggiungere un'istruzione `class`, digitate le parole `public class` seguite dal nome della classe. Aggiungente parentesi graffe per racchiudere il contenuto della classe (le definizioni di metodi e proprietà). Ad esempio:

```
public class MyClass
{
}
```

Il termine `public` indica che la classe è accessibile da qualsiasi altra riga di codice. Per altre soluzioni alternative, vedete Attributi spazio dei nomi controllo di accesso.

3 Digitate un'istruzione `package` per indicare il nome del pacchetto che contiene la classe. La sintassi è la parola `package`, seguita dal nome pacchetto completo, seguito dalle parentesi graffe che racchiudono il blocco di istruzioni `class`. Ad esempio, modificate il codice nel passaggio precedente come segue:

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4 Definite tutte le proprietà della classe usando l'istruzione `var` all'interno del corpo della classe. La sintassi corrisponde a quella utilizzata per dichiarare una variabile (con l'aggiunta del modificatore `public`). Ad esempio, l'aggiunta di queste righe tra le parentesi graffe della definizione della classe crea le proprietà `textProperty`, `numericProperty` e `dateProperty`:

```
public var textProperty:String = "some default value";
public var numericProperty:Number = 17;
public var dateProperty:Date;
```

5 Definite tutti i metodi della classe replicando la sintassi utilizzata per definire una funzione. Ad esempio:

- Per creare il metodo `myMethod()`, digitare:

```
public function myMethod(param1:String, param2:Number):void
{
    // do something with parameters
}
```

- Per creare la funzione di costruzione, cioè lo speciale metodo invocato come parte del processo di creazione di un'istanza di una classe, create un metodo a cui assegnare l'esatto nome della classe:

```
public function MyClass()
{
    // do stuff to set initial values for properties
    // and otherwise set up the object
    textVariable = "Hello there!";
    dateVariable = new Date(2001, 5, 11);
}
```

Se non includete un metodo di costruzione nella classe, il compilatore crea automaticamente un costruttore vuoto nella classe, ovvero un costruttore senza parametri né istruzioni.

Esistono alcuni altri elementi che potete definire per la classe. Si tratta di elementi più complessi.

- Gli *accessor* sono vie di mezzo tra i metodi e le proprietà. Nel codice che definisce una classe, l'accessor viene scritto come un metodo che consente di eseguire varie azioni, al contrario di una proprietà che si limita a leggere o ad assegnare un valore. Tuttavia, quando create un'istanza della classe, l'accessor viene trattato come una proprietà e se ne usa il nome per leggere o assegnare il valore.
- In ActionScript, la definizione degli eventi non richiede una sintassi particolare. Gli eventi della classe vengono invece definiti mediante le funzionalità della classe EventDispatcher.

Altri argomenti presenti nell'Aiuto

Gestione degli eventi

Esempio: creazione di un'applicazione di base

ActionScript 3.0 può essere utilizzato in vari ambienti di sviluppo diversi, tra cui Flash Professional e Flash Builder o qualsiasi editor di testo.

Questo esempio è una guida per affrontare le varie fasi necessarie per creare e perfezionare una semplice applicazione ActionScript 3.0 utilizzando Flash Professional o Flash Builder. L'applicazione da realizzare presenta un metodo semplice per utilizzare file di classe ActionScript 3.0 esterni in Flash Professional e Flex.

Progettazione dell'applicazione ActionScript

Questo esempio di applicazione ActionScript è un'applicazione "Hello World" standard, per cui la sua progettazione è semplice:

- L'applicazione si chiama HelloWorld.
- Visualizza un unico campo di testo contenente le parole "Hello World!".
- L'applicazione usa una singola classe a oggetti chiamata Greeter. Questa architettura consente di usare la classe in un progetto Flash Professional o Flex.
- Dopo aver creato una versione di base dell'applicazione, aggiungerete funzionalità per consentire all'utente di inserire un nome utente e all'applicazione di verificare la presenza del nome specificato in un elenco di utenti conosciuti.

Con questa semplice definizione del progetto, è possibile iniziare a realizzare l'applicazione vera e propria.

Creazione del progetto HelloWorld e della classe Greeter

La definizione del progetto per l'applicazione Hello World afferma che il codice deve essere facile da riutilizzare. A tal fine, l'applicazione impiega una classe a oggetti singola, chiamata Greeter, che viene utilizzata all'interno di un'applicazione creata in Flash Builder o Flash Professional.

Per creare il progetto HelloWorld e la classe Greeter in Flex:

1. In Flash Builder, selezionate File > New (Nuovo) > Flex Project (Progetto Flex).
2. Digitate HelloWorld come nome del progetto. Verificate che il tipo di applicazione sia impostato su "Web (runs in Adobe Flash Player)", quindi fate clic su Finish (Fine).

   Flash Builder crea il progetto e lo visualizza in Package Explorer. Per impostazione predefinita, il progetto contiene già un file denominato HelloWorld.mxml aperto nell'editor.

3 Create un file di classe ActionScript personalizzato in Flash Builder selezionando File > New (Nuovo) > ActionScript Class (Classe ActionScript).

4 Nel campo Name (Nome) della finestra di dialogo New ActionScript Class (Nuova classe ActionScript), digitate **Greeter** come nome della classe e fate clic su Finish (Fine).

Viene visualizzata una nuova finestra di modifica ActionScript.

Continuate con l'aggiunta di codice alla classe Greeter

Per creare la classe Greeter in Flash Professional:

1 In Flash Professional, selezionate File > Nuovo.

2 Nella finestra di dialogo Nuovo documento, selezionate File ActionScript e fre clic su OK.

Viene visualizzata una nuova finestra di modifica ActionScript.

3 Selezionate File > Salva. Selezionate una cartella di destinazione per l'applicazione, chiamate il file **Greeter.as** e fate clic su OK.

Continuate con l'aggiunta di codice alla classe Greeter.

Aggiunta di codice alla classe Greeter

La classe Greeter definisce un oggetto, `Greeter`, che potete utilizzare nell'applicazione HelloWorld.

Per aggiungere codice alla classe Greeter:

1 Digitate il codice seguente nel nuovo file (parte di questo codice potrebbe essere stato aggiunto per voi):

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

La classe Greeter include un metodo `sayHello()` singolo, che restituisce una stringa di testo “Hello World!”.

2 Selezionate File > Save (Salva) per salvare il file ActionScript.

La classe Greeter è pronta per essere utilizzata nell'applicazione.

Creazione di un'applicazione che utilizza il codice ActionScript

La classe Greeter appena creata definisce un set autonomo di funzioni software, ma non rappresenta un'applicazione completa. Per utilizzare la classe, create un documento Flash Professional o un progetto Flex.

Il codice richiede un'istanza della classe Greeter. Di seguito viene spiegato come utilizzare la classe Greeter nell'applicazione.

Per creare un'applicazione ActionScript mediante Flash Professional:

1 Selezionate File > Nuovo.

2 Nella finestra di dialogo Nuovo documento, selezionate File Flash (ActionScript 3.0) e fate clic su OK.

Viene visualizzata una nuova finestra del documento.

3 Selezionate File > Salva. Selezionate la stessa cartella in cui si trova il file di classe Greeter.as, denominate il file **HelloWorld.fla** e fate clic su OK.

4 Nel pannello Strumenti di Flash Professional, selezionate lo strumento Testo. Trascinate il puntatore sullo stage in modo da definire un nuovo campo di testo largo circa 300 pixel e alto circa 100 pixel.

5 Nel pannello Proprietà, con il campo di testo ancora selezionato sullo stage, impostate il tipo di testo su "Testo dinamico". Digitate **mainText** come nome di istanza del campo di testo.

6 Fate clic sul primo fotogramma della linea temporale. Aprite il pannello Azioni scegliendo Finestra > Azioni.

7 Nel pannello Azioni, digitate il seguente script:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello();
```

8 Salvate il file.

Continuat con Pubblicazione e prova dell'applicazione ActionScript.

Per creare un'applicazione ActionScript utilizzando Flash Builder:

1 Aprite il file HelloWorld.mxml e aggiungete il codice per la corrispondenza con il listato seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400"/>

</s:Application>
```

Il progetto Flex include quattro tag MXML:

- Un tag `<s:Application>`, che definisce il contenitore dell'applicazione
- Un tag `<s:layout>`, che definisce lo stile layout (layout verticale) per il tag Application

- Un tag `<fx:Script>` che include codice ActionScript
- Un tag `<s:TextArea>`, che definisce un campo utilizzato per visualizzare messaggi di testo per l'utente

Il codice nel tag `<fx:Script>` definisce un metodo `initApp()` che viene chiamato nel momento in cui l'applicazione viene caricata. Il metodo `initApp()` imposta il valore di testo del componente TextArea `mainTxt` su "Hello World!" stringa restituita dal metodo `sayHello()` della classe personalizzata Greeter appena creata.

2 Selezionate File > Save (Salva) per salvare l'applicazione.

Continuate con Pubblicazione e prova dell'applicazione ActionScript.

Pubblicazione e prova dell'applicazione ActionScript

Lo sviluppo di software è un processo iterativo. Il codice viene scritto, compilato e corretto finché la compilazione non risulta priva di errori. L'applicazione compilata viene quindi eseguita e provata per verificare che soddisfi l'obiettivo per cui è stata progettata, in caso contrario, modificate il codice fino a ottenere il risultato previsto. Gli ambienti di sviluppo Flash Professional e Flash Builder offrono vari modi per pubblicare e provare le applicazioni ed eseguirne il debug.

Di seguito è indicata la procedura di base da seguire per provare l'applicazione HelloWorld in ciascun ambiente.

Per pubblicare e provare un'applicazione ActionScript utilizzando Flash Professional:

1 Pubblicate l'applicazione e verificare se vengono generati errori di compilazione. In Flash Professional, selezionate Controllo > Prova filmato per compilare il codice ActionScript ed eseguire l'applicazione HelloWorld.

2 Se nel pannello Output vengono visualizzati errori o avvisi durante la prova dell'applicazione, correggete gli errori nei file HelloWorld.fla o HelloWorld.as e provate nuovamente l'applicazione.

3 Se non vi sono errori di compilazione, appare una finestra di Flash Player con l'applicazione Hello World.

Avete creato un'applicazione orientata agli oggetti semplice ma completa che utilizza ActionScript 3.0. A questo punto proseguite con Perfezionamento dell'applicazione HelloWorld.

Per pubblicare e provare un'applicazione ActionScript utilizzando Flash Builder:

1 Selezionate Run (Esegui) > Run HelloWorld (Esegui HelloWorld).

2 L'applicazione HelloWorld viene avviata.

- Se nel pannello Output vengono visualizzati errori o avvisi durante la prova dell'applicazione, correggete gli errori nei file HelloWorld.mxml or Greeter.as e provate nuovamente l'applicazione.
- Se non si verificano errori di compilazione, viene aperta una finestra del browser contenente l'applicazione Hello World. Il testo "Hello World!" viene visualizzato.

Avete creato un'applicazione orientata agli oggetti semplice ma completa che utilizza ActionScript 3.0. A questo punto proseguite con Perfezionamento dell'applicazione HelloWorld.

Perfezionamento dell'applicazione HelloWorld

Per rendere un po' più interessante l'applicazione, potete fare in modo che chieda un nome utente e convalidi il nome specificato confrontandolo con un elenco di nomi predefinito.

Innanzi tutto occorre aggiornare la classe Greeter aggiungendo una nuova funzionalità, per poi aggiornare l'applicazione in modo che utilizzi tale funzionalità.

Per aggiornare il file Greeter.as:

1 Aprite il file Greeter.as.

2 Modificate il contenuto del file in base all'esempio seguente (le righe nuove o modificate sono evidenziate in grassetto):

```
package
{
    public class Greeter
    {
        /**
         * Defines the names that receive a proper greeting.
         */
        public static var validNames:Array = ["Sammy", "Frank", "Dean"];

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                           + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

        /**
         * Checks whether a name is in the validNames list.
         */
        public static function validName(inputName:String = ""):Boolean
        {
            if (validNames.indexOf(inputName) > -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

A questo punto la classe Greeter presenta alcune nuove caratteristiche:

- L'array `validNames` elenca i nomi utente validi. L'array viene inizializzato su un elenco di tre nomi nel momento in cui la classe Greeter viene caricata.

- Il metodo `sayHello()` ora accetta un nome utente e modifica il saluto in base ad alcune condizioni. Se `userName` è una stringa vuota (`""`), la proprietà `greeting` viene impostata in modo da richiedere un nome all'utente. Se il nome utente è valido, il saluto diventa `"Hello, `*nomeUtente*`."` Infine, se nessuna di queste due condizioni è soddisfatta, la variabile `greeting` viene impostata su `"Sorry, `*nomeUtente*`, you are not on the list."`
- Il metodo `validName()` restituisce `true` se `inputName` viene trovato nell'elenco `validNames` oppure `false` se non viene trovato. L'istruzione `validNames.indexOf(inputName)` confronta ciascuna delle stringhe dell'array `validNames` con la stringa `inputName`. Il metodo `Array.indexOf()` restituisce la posizione di indice della prima istanza di un oggetto di un array, oppure il valore -1 se l'oggetto non viene trovato nell'array.

Quindi, modificate il file dell'applicazione che fa riferimento a questa classe ActionScript.

Per modificare l'applicazione utilizzando Flash Professional:

1. Aprite il file HelloWorld.fla.
2. Modificate lo script nel fotogramma 1 in modo che una stringa vuota (`""`) venga passata al metodo `sayHello()` della classe Greeter:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. Selezionate lo strumento Testo nel pannello Strumenti. Create due nuovi campi di testo sullo stage e affiancateli direttamente sotto il campo di testo `mainText` esistente.
4. Nel primo nuovo campo di testo, che è l'etichetta, digitate il testo **User Name:**.
5. Selezionate l'altro campo di testo aggiunto e, nella finestra di ispezione Proprietà, selezionate Testo di input come tipo di campo. Selezionate Riga singola come tipo di riga. Digitate **textIn** come nome di istanza.
6. Fate clic sul primo fotogramma della linea temporale.
7. Nel pannello Azioni, aggiungete le seguenti righe alla fine dello script esistente:

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

Il nuovo codice aggiunge la seguente funzionalità:

- Le prime due righe definiscono semplicemente i bordi dei due campi di testo.
- Un campo di testo di input, come `textIn`, può inviare una serie di eventi. Il metodo `addEventListener()` permette di definire una funzione che viene eseguita quando si verifica un determinato tipo di evento. In questo caso, l'evento è la pressione di un tasto della tastiera.
- La funzione personalizzata `keyPressed()` controlla se il tasto premuto è il tasto Invio. In questo caso, la funzione chiama il metodo `sayHello()` dell'oggetto `myGreeter`, passando il testo del campo di testo `textIn` come parametro. Tale metodo restituisce la stringa greeting basata sul valore passato. La stringa restituita viene quindi assegnata alla proprietà `text` del campo di testo `mainText`.

Lo script completo del fotogramma 1 è il seguente:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

8 Salvate il file.

9 Selezionate Controllo > Prova filmato per eseguire l'applicazione.

Quando eseguite l'applicazione viene richiesto di immettere un nome utente. Se il nome immesso è valido (Sammy, Frank o Dean), l'applicazione visualizza il messaggio di conferma “hello”.

Per modificare l'applicazione utilizzando Flash Builder:

1 Aprite il file HelloWorld.mxml.

2 Quindi, modificate il tag `<mx:TextArea>` per indicare che è di sola visualizzazione. Modificate il colore di sfondo su un grigio chiaro e impostate l'attributo `editable` su `false`:

```
    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false" />
```

3 Aggiungete le righe seguenti dopo il tag di chiusura `<s:TextArea>`. Queste righe creano un componente TextInput che consente all'utente di immettere un valore nome utente:

```
    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>
```

L'attributo `enter` definisce cosa accade quando l'utente preme il tasto Invio nel campo `userNameTxt`. Il questo esempio, il codice passa il testo nel campo al metodo `Greeter.sayHello()`. Il saluto nel campo `mainTxt` cambia di conseguenza.

Il file HelloWorld.mxml ha un aspetto simile al seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false"/>

    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>

</s:Application>
```

4 Salvate il file HelloWorld.mxml modificato. Selezionate Run (Esegui) > Run HelloWorld (Esegui HelloWorld) per eseguire l'applicazione.

Quando eseguite l'applicazione viene richiesto di immettere un nome utente. Se il nome immesso è valido (Sammy, Frank o Dean), l'applicazione visualizza il messaggio di conferma "`Hello, `*userName*".

Capitolo 3: Linguaggio e sintassi ActionScript

ActionScript 3.0 include sia il linguaggio ActionScript di base che l'API di Adobe Flash Player. Il linguaggio di base è la parte di ActionScript che definisce la sintassi del linguaggio e i tipi di dati di livello più alto. ActionScript 3.0 fornisce l'accesso programmatico ai runtime della piattaforma Adobe Flash: Adobe Flash Player e Adobe AIR.

Panoramica del linguaggio

Al centro del linguaggio ActionScript 3.0 vi sono gli oggetti che rappresentano i suoi blocchi costituitivi. Ogni variabile che dichiarate, ogni funzione che scrivete e ogni istanza di classe che create è un oggetto. Potete pensare a un programma ActionScript 3.0 come se fosse un gruppo di oggetti che svolgono certe attività, rispondono agli eventi e comunicano gli uni con gli altri.

I programmatori che hanno esperienza con la programmazione orientata agli oggetti (OOP) in Java o C++ possono pensare agli oggetti come moduli contenenti due tipi di membri: dati memorizzati in variabili membro o proprietà e comportamento accessibile tramite metodi. ActionScript 3.0 definisce oggetti in modo simile ma leggermente diverso. In ActionScript 3.0 gli oggetti sono semplicemente raccolte di proprietà, le quali sono contenitori non solo di dati, ma anche di funzioni e altri oggetti. Se una funzione è associata a un oggetto in questo modo, prende il nome di metodo.

Sebbene la definizione di ActionScript 3.0 possa sembrare un po' strana per i programmatori Java o C++, in pratica la definizione dei tipi di oggetto con le classi ActionScript 3.0 è molto simile al modo in cui le classi vengono definite in Java o C++. La distinzione tra le due definizioni di oggetto è importante ai fini della discussione sul modello di oggetti di ActionScript e per altri argomenti avanzati, ma nella maggior parte delle situazioni il termine *proprietà* equivale a variabile membro di classe e si contrappone a metodo. In ActionScript 3.0 Reference for the Adobe Flash Platform (Guida di riferimento di Adobe ActionScript 3.0 per la piattaforma Adobe Flash), ad esempio, viene utilizzato il termine *proprietà* per indicare variabili o proprietà di acquisizione-impostazione e il termine *metodi* per indicare le funzioni che fanno parte di una classe.

Una sottile differenza tra le classi di ActionScript e quelle di Java o C++ è rappresentata dal fatto che in ActionScript le classi non sono semplicemente entità astratte, ma sono rappresentate da *oggetti di classe* che memorizzano le proprietà e i metodi della classe. Ciò rende possibile l'uso di tecniche che possono sembrare inusuali ai programmatori Java e C++, come l'inclusione di istruzioni o di codice eseguibile al livello principale di una classe o di un pacchetto.

Un'altra differenza tra le classi ActionScript e le classi Java o C++ è costituita dal fatto che ogni classe ActionScript possiede un elemento chiamato *oggettoprototipo*. Nelle versioni precedenti di ActionScript, gli oggetti prototipo, collegati tra loro nelle *catene di prototipi*, costituivano insieme la base dell'intera gerarchia di ereditarietà delle classi. In ActionScript 3.0, invece, gli oggetti prototipo svolgono una funzione secondaria nel sistema dell'ereditarietà. L'oggetto prototipo può tuttavia essere ancora utile come alternativa a proprietà e metodi statici quando si desidera condividere una proprietà e il relativo valore tra tutte le istanze di una classe.

In passato, i programmatori ActionScript esperti potevano modificare direttamente la catena di prototipi con elementi di linguaggio incorporati speciali. Ora che il linguaggio fornisce un'implementazione più avanzata di un'interfaccia di programmazione basata su classi, molti di questi elementi speciali, ad esempio `__proto__` e `__resolve`, non fanno più parte del linguaggio. Inoltre, ottimizzazioni del meccanismo di ereditarietà interno che forniscono miglioramenti significativi delle prestazioni precludono l'accesso diretto a tale meccanismo.

Oggetti e classi

In ActionScript 3.0 ogni oggetto è definito da una classe. Una classe può essere considerata come un modello di base per un tipo di oggetto. Le definizioni di classe possono riguardare variabili e costanti, che memorizzano valori di dati, oppure metodi, ovvero funzioni che incorporano un comportamento associato alla classe. I valori memorizzati nelle proprietà possono essere *valori di base* (primitive values) o altri oggetti. I valori di base sono numeri, stringhe o valori booleani.

ActionScript contiene una serie di classi incorporate che fanno parte del linguaggio di base. Alcune di tali classi, quali Number, Boolean e String, rappresentano i valori di base disponibili in ActionScript. Altre, come ad esempio le classi Array, Math e XML, definiscono oggetti complessi.

Tutte le classi, incorporate o definite dall'utente, derivano dalla classe Object. Per i programmatori che hanno già utilizzato ActionScript in precedenza è importante notare che il tipo di dati Object non è più quello predefinito, anche se tutte le altre classi derivano ancora da esso. In ActionScript 2.0 le due righe di codice seguenti erano equivalenti perché l'assenza di un'annotazione di tipo comportava che una variabile fosse del tipo Object:

```
var someObj:Object;
var someObj;
```

ActionScript 3.0 invece introduce il concetto di "variabili senza tipo", che possono essere indicate nei due modi seguenti:

```
var someObj:*;
var someObj;
```

Una variabile senza tipo non è la stessa cosa di una variabile del tipo Object. La differenza fondamentale consiste nel fatto che le variabili senza tipo possono contenere il valore speciale `undefined`, mentre una variabile del tipo Object non può.

Si possono definire classi personalizzate utilizzando la parola chiave `class`. Sono disponibili tre metodi per dichiarare le proprietà della classe: le constanti possono essere definite con la parola chiave `const`, le variabili vengono definite con la parola chiave `var` e le proprietà di acquisizione e impostazione sono definite utilizzando gli attributi `get` e `set` in una dichiarazione di metodo. I metodi vengono dichiarati con la parola chiave `function`.

Un'istanza di una classe viene creata utilizzando l'operatore `new`. Nell'esempio seguente viene creata un'istanza della classe Date denominata `myBirthday`.

```
var myBirthday:Date = new Date();
```

Pacchetti e spazi dei nomi

Pacchetti e spazi dei nomi sono concetti correlati. I pacchetti consentono di "impacchettare" insieme le definizioni di classe in modo da facilitare la condivisione del codice e ridurre al minimo i conflitti tra nomi. Gli spazi dei nomi permettono di controllare la visibilità degli identificatori, quali i nomi di proprietà e di metodi, e possono essere applicati al codice sia all'interno che all'esterno di un pacchetto. I pacchetti consentono di organizzare i file di classe e gli spazi dei nomi di gestire la visibilità di singole proprietà e metodi.

Pacchetti

In ActionScript 3.0, i pacchetti sono implementati insieme agli spazi dei nomi, ma non sono la stessa cosa. Quando dichiarate un pacchetto, implicitamente create un tipo speciale di spazio dei nomi con la garanzia che sarà noto in fase di compilazione. Al contrario, uno spazio dei nomi creato esplicitamente non è necessariamente noto in fase di compilazione.

Nell'esempio seguente viene utilizzata la direttiva `package` per creare un pacchetto semplice che contiene un'unica classe:

```
package samples
{
    public class SampleCode
    {
        public var sampleGreeting:String;
        public function sampleFunction()
        {
            trace(sampleGreeting + " from sampleFunction()");
        }
    }
}
```

Il nome della classe di questo esempio è SampleCode. Poiché la classe è all'interno del pacchetto samples, il compilatore qualifica automaticamente il nome della classe in fase di compilazione con il rispettivo nome completo: samples.SampleCode. Il compilatore qualifica anche i nomi di qualsiasi proprietà o metodo, in modo che `sampleGreeting` e `sampleFunction()` diventino rispettivamente `samples.SampleCode.sampleGreeting` e `samples.SampleCode.sampleFunction()`.

Molti sviluppatori, specialmente quelli con esperienza Java, potrebbero scegliere di collocare solo le classi al livello principale di un pacchetto. Tuttavia, ActionScript 3.0 supporta non solo le classi in tale posizione, ma anche le variabili, le funzioni e persino le istruzioni. Uno degli usi avanzati di questa caratteristica consiste nel definire uno spazio dei nomi al livello principale di un pacchetto, in modo che sia disponibile a tutte le classi del pacchetto. Notate, tuttavia, che soltanto due specificatori di accesso (`public` e `internal`) sono consentiti al livello principale di un pacchetto. A differenza di Java, che consente di dichiarare come private le classi nidificate, ActionScript 3.0 non supporta classi nidificate o private.

Per molti altri aspetti, tuttavia, i pacchetti di ActionScript 3.0 sono simili a quelli del linguaggio di programmazione Java. Come potete notare nell'esempio precedente, i riferimenti ai nomi completi dei pacchetti vengono espressi utilizzando l'operatore punto (.), proprio come in Java. Potete sfruttare i pacchetti per organizzare il codice in una struttura gerarchica intuitiva che possa essere utilizzata da altri programmatori. In questo modo si facilita la condivisione del codice, poiché si possono creare pacchetti personalizzati da mettere a disposizione degli altri e utilizzare nel proprio codice pacchetti creati da altri programmatori.

Inoltre, l'uso dei pacchetti fa sì che i nomi di identificazione utilizzati siano univoci e non in conflitto con quelli di altri pacchetti. Si potrebbe addirittura sostenere che questo sia il vantaggio principale dei pacchetti. Ad esempio, supponete che due programmatori abbiano deciso di condividere il loro codice e che ciascuno dei due abbia creato una classe chiamata SampleCode. Senza i pacchetti si creerebbe un conflitto tra i nomi e l'unica soluzione sarebbe quella di rinominare una delle due classi. Grazie ai pacchetti, invece, il conflitto viene evitato facilmente collocando una o (preferibilmente) entrambe le classi in pacchetti con nomi univoci.

È anche possibile includere dei punti (.) incorporati in un pacchetto per creare pacchetti nidificati, allo scopo di realizzare un'organizzazione gerarchica dei pacchetti. Un ottimo esempio è il pacchetto flash.display fornito da ActionScript 3.0 e nidificato all'intero del pacchetto flash.

La maggior parte di ActionScript 3.0 è organizzato sotto il pacchetto flash. Ad esempio, il pacchetto flash.display contiene l'API dell'elenco di visualizzazione e il pacchetto flash.events contiene il nuovo modello di eventi.

Creazione di pacchetti

ActionScript 3.0 offre una notevole flessibilità nell'organizzazione di pacchetti, classi e file di origine. Le versioni precedenti di ActionScript consentivano l'uso di un'unica classe per ogni file di origine e imponevano la corrispondenza tra il nome del file di origine e quello della classe. ActionScript 3.0 invece permette di includere più classi nello stesso file di origine, ma una sola classe di ogni file può essere resa disponibile al codice esterno al file stesso. In altre parole, solo una classe di ogni file può essere dichiarata all'interno di una dichiarazione di pacchetto. Le eventuali altre classi devono essere dichiarate all'esterno, pertanto risultano invisibili al codice esterno al file di origine. Il nome della classe dichiarata nella definizione del pacchetto deve corrispondere al nome del file di origine.

La maggiore flessibilità di ActionScript 3.0 si può notare anche nel modo in cui vengono definiti i pacchetti. Nelle versioni precedenti di ActionScript, i pacchetti rappresentavano semplicemente directory in cui inserire i file di origine; inoltre, i pacchetti non venivano dichiarati con l'istruzione `package`, ma piuttosto il nome del pacchetto veniva incluso come parte del nome completo della classe nella dichiarazione della classe. In ActionScript 3.0 i pacchetti rappresentano ancora delle directory, ma il loro contenuto non è limitato alle classi. L'istruzione `package` di ActionScript 3.0 consente di dichiarare non solo un pacchetto ma anche variabili, funzioni e spazi dei nomi al livello principale di un pacchetto. È anche possibile includere istruzioni eseguibili, sempre al livello principale del pacchetto. Se scegliete di dichiarare variabili, funzioni o spazi dei nomi al livello principale di un pacchetto, gli unici attributi disponibili a quel livello sono `public` e `internal` e soltanto una dichiarazione a livello di pacchetto per ciascun file può utilizzare l'attributo `public`, a prescindere che venga dichiarata una classe, una variabile, una funzione o uno spazio dei nomi.

I pacchetti sono utili per organizzare il codice e per evitare conflitti tra i nomi. Il concetto di pacchetto non va confuso con quello di ereditarietà delle classi, al quale non è correlato. Due classi che risiedono nello stesso pacchetto hanno uno spazio dei nomi in comune, ma non sono necessariamente correlate tra loro in alcun altro modo. Analogamente, un pacchetto nidificato potrebbe non avere alcune relazione semantica con il pacchetto di livello superiore.

Importazione dei pacchetti

Per utilizzare una classe che si trova all'interno di un pacchetto, dovete importare il pacchetto stesso o la classe specifica. In ActionScript 2.0, invece, l'importazione delle classi era opzionale.

Ad esempio, considerate l'esempio di classe SampleCode presentato in precedenza. Se la classe risiede in un pacchetto chiamato samples, dovete utilizzare una delle seguenti istruzioni import per utilizzare la classe SampleCode:

```
import samples.*;
```

oppure

```
import samples.SampleCode;
```

In generale, le istruzioni `import` devono essere il più possibile specifiche. Se prevedete di utilizzare solo la classe SampleCode del pacchetto samples, dovete importare solo la classe SampleCode anziché l'intero pacchetto al quale appartiene. L'importazione di interi pacchetti può determinare conflitti imprevisti tra i nomi.

Inoltre, dovete collocare il codice che definisce il pacchetto o la classe all'interno del *percorso di classe*. Il percorso di classe (classpath) è un elenco definito dall'utente dei percorsi di directory locali, che determina dove il compilatore cerca i pacchetti e le classi importate. Talvolta il percorso di classe viene anche definito *percorso di compilazione (build path)* o *percorso di origine (source path)*.

Dopo aver importato correttamente la classe o il pacchetto, potete utilizzare il nome completo della classe (samples.SampleCode) oppure semplicemente il nome della classe (SampleCode).

I nomi completi sono utili quando classi, metodi o proprietà con nomi identici possono creare ambiguità nel codice, ma possono risultare più difficili da gestire se utilizzati per tutti gli identificatori. Ad esempio, l'uso del nome completo determina un codice troppo verboso quando si crea un'istanza della classe SampleCode:

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

Man mano che aumentano i livelli di nidificazione dei pacchetti, diminuisce la leggibilità del codice. Nei casi in cui siete sicuri che la presenza di identificatori ambigui non rappresenta un problema, l'uso degli identificatori semplici consente di ottenere codice più leggibile. Ad esempio, il codice che crea un'istanza della classe SampleCode è molto più semplice se si include solo l'identificatore di classe:

```
var mySample:SampleCode = new SampleCode();
```

Se tentate di utilizzare nomi di identificatori senza aver prima importato il pacchetto o la classe corrispondente, il compilatore non è in grado di trovare le definizioni di classe. D'altro canto, tuttavia, se importate un pacchetto o una classe e tentate di definire un nome che è in conflitto con un nome importato, viene generato un errore.

Quando si crea un pacchetto, lo specificatore di accesso predefinito per tutti i membri del pacchetto è `internal`, vale a dire che, per impostazione predefinita, tutti i membri del pacchetto sono visibili soltanto agli altri membri dello stesso pacchetto. Se desiderate che una classe sia disponibile al codice esterno al pacchetto, dovete dichiararla come `public`. Ad esempio, il pacchetto seguente contiene due classi, SampleCode e CodeFormatter:

```
// SampleCode.as file
package samples
{
    public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
    class CodeFormatter {}
}
```

La classe SampleCode è visibile all'esterno del pacchetto perché è dichiarata come una classe `public`. La classe CodeFormatter, al contrario, è visibile solo nel pacchetto samples. Se tentate di accedere alla classe CodeFormatter esternamente al pacchetto samples, viene generato un errore, come mostrato nell'esempio seguente:

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

Se desiderate che ambedue le classi siano disponibili al codice esterno al pacchetto, dovete dichiararle entrambe come `public`. Non è possibile applicare l'attributo `public` alla dichiarazione del pacchetto.

I nomi completi sono utili per risolvere conflitti tra nomi che potrebbero verificarsi quando si usano i pacchetti. Uno scenario di questo tipo si potrebbe presentare se si importano due pacchetti che definiscono classi con lo stesso identificatore. Ad esempio, esaminate il pacchetto seguente, che contiene a sua volta una classe chiamata SampleCode:

```
package langref.samples
{
    public class SampleCode {}
}
```

Se importate entrambe le classi, come nell'esempio seguente, si determina un conflitto di nomi quando utilizzate la classe SampleCode:

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

Il compilatore non ha modo di sapere quale classe SampleCode deve essere utilizzata. Per risolvere il conflitto, dovete utilizzare il nome completo di ciascuna classe, come indicato di seguito:

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

***Nota:** i programmatori con esperienza in C++ spesso confondono l'istruzione `import` con `#include`. La direttiva `#include` è necessaria in C++ perché i compilatori C++ elaborano un file alla volta e non cercano le definizioni di classe in altri file a meno che non venga incluso esplicitamente un file di intestazione. Anche in ActionScript 3.0 è presente una direttiva `include`, che tuttavia non è progettata per l'importazione di classi e pacchetti. Per importare classi o pacchetti in ActionScript 3.0, dovete utilizzare l'istruzione `import` e inserire nel percorso di classe il file di origine che contiene il pacchetto.*

Spazi dei nomi

Gli spazi dei nomi permettono di controllare la visibilità delle proprietà e dei metodi creati dal programmatore. Gli specificatori di controllo dell'accesso `public`, `private`, `protected` e `internal` possono essere considerati come spazi dei nomi incorporati. Se riscontrate che questi specificatori predefiniti non soddisfano tutte le vostre esigenze, potete creare spazi dei nomi personalizzati.

Se avete esperienza con gli spazi dei nomi XML, questo argomento vi risulterà già noto, benché la sintassi e i dettagli dell'implementazione ActionScript siano leggermente diversi rispetto al linguaggio XML. Se è la prima volta che utilizzate gli spazi dei nomi, il concetto in sé è di facile comprensione ma l'implementazione prevede una terminologia specifica che occorrerà apprendere.

Per comprendere il funzionamento degli spazi dei nomi, è utile sapere che il nome di una proprietà o metodo contiene sempre due parti: un identificatore e uno spazio dei nomi. L'identificatore può essere considerato equivalente a un nome. Ad esempio, gli identificatori nella definizione di classe seguente sono `sampleGreeting` e `sampleFunction()`:

```
class SampleCode
{
    var sampleGreeting:String;
    function sampleFunction () {
        trace(sampleGreeting + " from sampleFunction()");
    }
}
```

Ogni qual volta le definizioni non sono precedute da un attributo spazio dei nomi, i loro nomi sono qualificati dallo spazio dei nomi `internal` predefinito, il che significa che sono visibili solo a chiamanti nello stesso pacchetto. Se è impostata la modalità rigorosa, il compilatore genera un'avvertenza per segnalare che lo spazio dei nomi `internal` viene applicato a tutti gli identificatori privi di attributo namespace. Per far sì che un identificatore sia disponibile senza limiti, dovete anteporre esplicitamente al suo nome l'attributo `public`. Nel codice di esempio precedente, `sampleGreeting` e `sampleFunction()` hanno un valore spazio dei nomi di `internal`.

L'utilizzo degli spazi dei nomi prevede tre passaggi fondamentali. Innanzi tutto, dovete definire lo spazio dei nomi mediante la parola chiave `namespace`. Ad esempio, il codice seguente definisce lo spazio dei nomi `version1`:

```
namespace version1;
```

In secondo luogo, applicate lo spazio dei nomi utilizzandolo al posto di uno specificatore di accesso in una proprietà o nella dichiarazione di un metodo. L'esempio seguente inserisce una funzione denominata `myFunction()` nello spazio dei nomi `version1`:

```
version1 function myFunction() {}
```

Terzo, dopo che è stato applicato, potete fare riferimento allo spazio dei nomi con la direttiva `use` o qualificando il nome di un identificatore con uno spazio dei nomi. L'esempio seguente fa riferimento alla funzione `myFunction()` mediante la direttiva `use`:

```
use namespace version1;
myFunction();
```

Potete anche utilizzare un nome qualificato per fare riferimento alla funzione `myFunction()`, come mostrato nell'esempio seguente:

```
version1::myFunction();
```

Definizione degli spazi dei nomi

Gli spazi dei nomi contengono un solo valore, l'URI (Uniform Resource Identifier), che talvolta viene definito *nome dello spazio dei nomi*. Un URI consente di rendere univoca la definizione di uno spazio dei nomi.

Per creare uno spazio dei nomi, occorre dichiarare una definizione namespace. Quest'ultima può contenere un URI esplicito (come per gli spazi dei nomi XML) oppure esserne priva. L'esempio seguente mostra uno spazio dei nomi definito con un URI:

```
namespace flash_proxy = "http://www.adobe.com/flash/proxy";
```

L'URI funge da stringa di identificazione univoca per lo spazio dei nomi. Se viene omesso, come nell'esempio seguente, il compilatore crea al suo posto una stringa di identificazione univoca interna, alla quale non è possibile accedere.

```
namespace flash_proxy;
```

Una volta definito uno spazio dei nomi, con o senza URI, non è possibile ridefinirlo all'interno della stessa area di validità. Un eventuale tentativo in questo senso produce un errore del compilatore.

Se uno spazio dei nomi viene definito all'interno di un pacchetto o di una classe, potrebbe non essere visibile per il codice esterno a tale pacchetto o classe, a meno che non venga utilizzato lo specificatore di controllo accesso appropriato. Ad esempio, nel codice seguente lo spazio dei nomi `flash_proxy` è definito all'interno del pacchetto flash.utils. Nell'esempio, l'assenza di uno specificatore di controllo accesso fa sì che lo spazio dei nomi `flash_proxy` sia visibile solo al codice che si trova nel pacchetto flash.utils e invisibile a tutto il codice esterno a tale pacchetto:

```
package flash.utils
{
    namespace flash_proxy;
}
```

Il codice seguente utilizza l'attributo `public` per rendere visibile lo spazio dei nomi `flash_proxy` a codice all'esterno del pacchetto:

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Applicazione degli spazi dei nomi

Applicare uno spazio dei nomi significa inserire una definizione all'interno di uno spazio dei nomi. Le definizioni che possono essere inserite negli spazi dei nomi sono le funzioni, le variabili e le costanti (non è possibile inserire una classe in uno spazio dei nomi personalizzato).

Considerate, ad esempio, una funzione dichiarata mediante lo specificatore di controllo accesso `public`. L'uso dell'attributo `public` in una definizione di funzione determina l'inserimento della funzione nello spazio dei nomi pubblico e la conseguente disponibilità della funzione stessa per tutto il codice. Una volta definito uno spazio dei nomi personalizzato, potete utilizzarlo come l'attributo `public`, rendendo disponibile la definizione al codice che può fare riferimento allo spazio dei nomi. Ad esempio, se definite lo spazio dei nomi `example1`, potete aggiungere un metodo chiamato `myFunction()` usando `example1` come attributo, come nell'esempio seguente:

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

Se dichiarate il metodo `myFunction()` utilizzando lo spazio dei nomi `example1` come un attributo, significa che il metodo appartiene allo spazio dei nomi `example1`.

Quando applicate uno spazio dei nomi, tenete presente quanto segue:

- È possibile applicare un solo spazio dei nomi a ogni dichiarazione.
- Non è possibile applicare un attributo namespace a più di una definizione alla volta. In altre parole, se volete applicare uno spazio dei nomi a dieci diverse funzioni, dovete aggiungere l'attributo namespace a ciascuna delle dieci definizioni di funzione.
- Se applicate uno spazio dei nomi, non potete specificare anche uno specificatore di controllo accesso, poiché i due elementi si escludono a vicenda. In altre parole, non è possibile dichiarare una funzione o una proprietà con l'attributo `public`, `private`, `protected` o `internal` dopo aver già applicato lo spazio dei nomi corrispondente.

Riferimenti agli spazi dei nomi

Non occorre fare riferimento in modo esplicito a uno spazio dei nomi quando utilizzate un metodo o una proprietà dichiarata con uno degli specificatori di controllo accesso, quali `public`, `private`, `protected` e `internal`, dal momento che l'accesso a questi spazi dei nomi è controllato dal contesto. Ad esempio, le definizioni inserite nello spazio dei nomi `private` sono automaticamente disponibili al codice che si trova all'interno della stessa classe. Nel caso degli spazi dei nomi personalizzati, tuttavia, questa "sensibilità al contesto" non esiste. Per utilizzare un metodo o una proprietà inserita in uno spazio dei nomi personalizzato, dovete fare riferimento in modo esplicito allo spazio dei nomi.

È possibile fare riferimento agli spazi dei nomi mediante la direttiva `use namespace` oppure qualificando il nome con lo spazio dei nomi mediante la sintassi `::`. Fare riferimento a uno spazio dei nomi con la direttiva `use namespace` consente di "aprire" lo spazio dei nomi, in modo che possa essere applicato a qualsiasi identificatore non qualificato. Ad esempio, se avete definito lo spazio dei nomi `example1`, potete accedere a nomi nello spazio dei nomi utilizzando `use namespace example1`:

```
use namespace example1;
myFunction();
```

È possibile aprire più spazi dei nomi nello stesso momento. Quando utilizzate la direttiva `use namespace`, lo spazio dei nomi rimane aperto nell'intero blocco di codice in cui è stato aperto. Non esiste un modo per chiudere esplicitamente uno spazio dei nomi.

La presenza contemporanea di due o più spazi dei nomi aperti, tuttavia, aumenta la probabilità di conflitti tra nomi. Se preferite non aprire uno spazio dei nomi, potete evitare di usare la direttiva `use namespace` oppure qualificando il nome del metodo o della proprietà con lo spazio dei nomi e il segno ::. Ad esempio, nel codice seguente il nome `myFunction()` viene qualificato con lo spazio dei nomi `example1`:

```
example1::myFunction();
```

Uso degli spazi dei nomi

Nella classe flash.utils.Proxy di ActionScript 3.0 potete trovare un esempio di uno spazio dei nomi reale che viene utilizzato per evitare conflitti tra nomi. La classe Proxy, che sostituisce la proprietà `Object.__resolve` di ActionScript 2.0, permette di intercettare i riferimenti a proprietà o metodi non definiti prima che si verifichi un errore. Tutti i metodi della classe Proxy risiedono nello spazio dei nomi `flash_proxy` per impedire conflitti tra i nomi.

Per comprendere meglio come viene utilizzato lo spazio dei nomi `flash_proxy`, è necessario capire come funziona la classe Proxy. La funzionalità di questa classe è disponibile unicamente alle classi che ereditano da essa. In altre parole, se desiderate utilizzare i metodi della classe Proxy di un oggetto, la definizione di classe dell'oggetto deve estendere la classe Proxy. Ad esempio, se volete intercettare i tentativi di chiamare un metodo non definito, potete estendere la classe Proxy ed eseguire l'override del metodo `callProperty()` della stessa classe.

Si è detto in precedenza che l'implementazione degli spazi dei nomi è solitamente un processo che prevede tre passaggi: definire uno spazio dei nomi, applicarlo e farvi riferimento. Tuttavia, poiché non viene mai fatto riferimento in modo esplicito a nessuno dei metodi della classe Proxy, lo spazio dei nomi `flash_proxy` viene solo definito e applicato e non specificato in un riferimento. ActionScript 3.0 definisce lo spazio dei nomi `flash_proxy` e lo applica nella classe Proxy. Il vostro codice deve soltanto applicare lo spazio dei nomi `flash_proxy` alle classi che estendono la classe Proxy.

Lo spazio dei nomi `flash_proxy` è definito nel pacchetto flash.utils in un modo simile al seguente:

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Lo spazio dei nomi viene applicato ai metodi della classe Proxy come indicato nel seguente codice estratto da tale classe:

```
public class Proxy
{
    flash_proxy function callProperty(name:*, ... rest):*
    flash_proxy function deleteProperty(name:*):Boolean
    ...
}
```

Come mostrato nel codice seguente, è necessario innanzitutto importare la classe Proxy e lo spazio dei nomi `flash_proxy`, quindi occorre dichiarare la classe in modo che estenda la classe Proxy (nonché aggiungere l'attributo `dynamic`, se il compilatore è in modalità rigorosa). Se sostituite il metodo `callProperty()`, dovete utilizzare lo spazio dei nomi `flash_proxy`.

```
package
{
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    dynamic class MyProxy extends Proxy
    {
        flash_proxy override function callProperty(name:*, ...rest):*
        {
            trace("method call intercepted: " + name);
        }
    }
}
```

Se create un'istanza della classe MyProxy e chiamate un metodo non definito, ad esempio il metodo `testing()` chiamato nell'esempio che segue, l'oggetto Proxy intercetta la chiamata al metodo ed esegue le istruzioni contenute nel metodo `callProperty()` sostituito (in questo caso, una semplice istruzione `trace()`).

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

L'inclusione dei metodi della classe Proxy all'interno dello spazio dei nomi `flash_proxy` presenta due vantaggi. Innanzi tutto, la presenza di uno spazio dei nomi separato riduce la quantità di elementi nell'interfaccia pubblica di qualunque classe che estende la classe Proxy. (Vi sono circa una dozzina di metodi nella classe Proxy che possono essere sostituiti, nessuno dei quali è progettato per essere chiamato direttamente. Includerli tutti nello spazio dei nomi pubblico potrebbe creare confusione.) In secondo luogo, l'uso dello spazio dei nomi `flash_proxy` permette di evitare conflitti tra i nomi qualora la sottoclasse Proxy contenga nomi di metodi di istanze che corrispondono a nomi di metodi della classe Proxy. Ad esempio, supponete che uno dei metodi sia denominato `callProperty()`. Il codice seguente è valido perché la versione personalizzata del metodo `callProperty()` si trova in uno spazio dei nomi diverso:

```
dynamic class MyProxy extends Proxy
{
    public function callProperty() {}
    flash_proxy override function callProperty(name:*, ...rest):*
    {
        trace("method call intercepted: " + name);
    }
}
```

Gli spazi dei nomi sono utili anche quando desiderate fornire accesso a metodi e proprietà in un modo che non sarebbe possibile con i quattro specificatori di controllo accesso (`public`, `private`, internal e `protected`). Ad esempio, se avete vari metodi di utilità sparsi in diversi pacchetti e desiderate renderli disponibili a tutti i pacchetti senza tuttavia renderli pubblici, potete creare uno spazio dei nomi e utilizzarlo come specificatore di controllo accesso personalizzato.

L'esempio seguente utilizza uno spazio dei nomi definito dall'utente per raggruppare due funzioni che si trovano in pacchetti differenti. Grazie a questa operazione, potete rendere visibili entrambe le funzioni a una classe o a un pacchetto tramite una singola istruzione `use namespace`.

L'esempio che segue usa quattro file per dimostrare questa tecnica. Tutti i file devono trovarsi all'interno del percorso di classe. Il primo, myInternal.as, viene utilizzato per definire lo spazio dei nomi `myInternal`. Poiché si trova in un pacchetto denominato example, è necessario inserirlo in una cartella con lo stesso nome. Lo spazio dei nomi viene contrassegnato come `public` in modo che possa essere importato in altri pacchetti.

```
// myInternal.as in folder example
package example
{
    public namespace myInternal = "http://www.adobe.com/2006/actionscript/examples";
}
```

Il secondo e il terzo file, Utility.as e Helper.as, definiscono le classi che contengono i metodi da rendere disponibili agli altri pacchetti. Poiché la classe Utility è nel pacchetto example.alpha, il file deve essere inserito in una cartella denominata alpha a sua volta contenuta nella cartella example. Allo stesso modo, la classe Helper si trova nel pacchetto example.alpha e il file corrispondente deve essere inserito nella sottocartella beta della cartella example. Entrambi i pacchetti, example.alpha e example.beta, devono importare lo spazio dei nomi per poterlo utilizzare.

```
// Utility.as in the example/alpha folder
package example.alpha
{
    import example.myInternal;

    public class Utility
    {
        private static var _taskCounter:int = 0;

        public static function someTask()
        {
            _taskCounter++;
        }

        myInternal static function get taskCounter():int
        {
            return _taskCounter;
        }
    }
}

// Helper.as in the example/beta folder
package example.beta
{
    import example.myInternal;

    public class Helper
    {
        private static var _timeStamp:Date;

        public static function someTask()
        {
            _timeStamp = new Date();
        }

        myInternal static function get lastCalled():Date
        {
            return _timeStamp;
        }
    }
}
```

Il quarto file, NamespaceUseCase.as, è la classe principale dell'applicazione e deve essere allo stesso livello della cartella example. In Flash Professional questa classe verrebbe utilizzata come classe documento per il file FLA. Anche la classe NamespaceUseCase importa lo spazio dei nomi `myInternal` e lo utilizza per chiamare i due metodi statici contenuti negli altri pacchetti. L'esempio utilizza i metodi statici solo per semplificare il codice. Nello spazio dei nomi `myInternal` è possibile inserire sia metodi statici che di istanza.

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

Variabili

Le variabili consentono di memorizzare valori da utilizzare in un programma. Per dichiarare una variabile, è necessario utilizzare l'istruzione `var` nel suo nome, mentre è obbligatorio in tutti i casi in ActionScript 3.0. Ad esempio, la seguente riga di codice ActionScript dichiara una variabile denominata `i`:

```
var i;
```

Se omettete l'istruzione `var` quando dichiarate una variabile, viene generato un errore del compilatore in modalità rigorosa e un errore runtime in modalità standard. Ad esempio, la seguente riga di codice genera un errore se la variabile `i` non è stata definita in precedenza:

```
i; // error if i was not previously defined
```

L'associazione di una variabile a un tipo di dati deve essere effettuata al momento di dichiarare la variabile. La dichiarazione di una variabile senza indicazione del tipo è consentita ma genera un'avvertenza del compilatore in modalità rigorosa. Per designare il tipo di una variabile, aggiungete al nome un carattere di due punti (:) seguito dal tipo. Ad esempio, il codice seguente dichiara una variabile denominata `i` del tipo int:

```
var i:int;
```

Potete assegnare un valore a una variabile utilizzando l'operatore di assegnazione (=). Ad esempio, il codice seguente dichiara la variabile `i` e vi assegna il valore 20:

```
var i:int;
i = 20;
```

Può risultare più pratico assegnare un valore a una variabile nel momento in cui viene dichiarata, come nell'esempio seguente:

```
var i:int = 20;
```

Questa tecnica viene comunemente utilizzata non solo quando si assegnano valori di base come numeri interi e stringhe, ma anche quando si un crea un array o un'istanza di una classe. L'esempio seguente mostra un array che viene dichiarato e al quale viene assegnato un valore nella stessa riga di codice.

```
var numArray:Array = ["zero", "one", "two"];
```

Potete creare un'istanza di una classe utilizzando l'operatore `new`. L'esempio seguente crea un'istanza della classe `CustomClass` e assegna alla variabile `customItem` un riferimento alla nuova istanza creata:

```
var customItem:CustomClass = new CustomClass();
```

Potete dichiarare più variabili nella stessa riga di codice utilizzando l'operatore virgola (`,`) per separarle. Ad esempio, il codice seguente dichiara tre variabili nella stessa riga:

```
var a:int, b:int, c:int;
```

È anche possibile assegnare valori a ciascuna delle variabili nella stessa riga di codice. Ad esempio, il codice seguente dichiara tre variabili (`a`, `b` e `c`) e assegna un valore a ciascuna:

```
var a:int = 10, b:int = 20, c:int = 30;
```

L'uso dell'operatore virgola per raggruppare le dichiarazioni di variabili nella stessa istruzione può tuttavia ridurre la leggibilità del codice.

Area di validità delle variabili

L'*area di validità* (scope) di una variabile è la parte di codice all'interno della quale è possibile accedere alla variabile mediante un riferimento lessicale. Una variabile *globale* è definita in tutte le aree del codice, mentre una variabile *locale* è definita in una sola parte. In ActionScript 3.0, alle variabili viene sempre assegnata l'area di validità della funzione o della classe nella quale sono dichiarate. Una variabile globale è una variabile definita all'esterno di qualunque funzione o definizione di classe. Ad esempio, il codice seguente crea una variabile globale `strGlobal` dichiarandola all'esterno di qualunque funzione. L'esempio mostra che una variabile globale è disponibile sia all'interno che all'esterno della definizione di funzione.

```
var strGlobal:String = "Global";
function scopeTest()
{
    trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

Una variabile locale viene dichiarata all'interno di una definizione di funzione. L'area di codice più piccola per la quale è possibile definire una variabile locale è una definizione di funzione. Una variabile locale dichiarata all'interno di una funzione esiste solo in tale funzione. Se, ad esempio, dichiarate una variabile denominata `str2` all'interno di una funzione denominata `localScope()`, tale variabile non è disponibile all'esterno della funzione.

```
function localScope()
{
    var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

Se la variabile locale è già stata dichiarata con lo stesso nome come variabile globale, la definizione locale ha la precedenza sulla definizione globale all'interno dell'area di validità della variabile locale. La variabile globale rimane valida all'esterno della funzione. Il codice seguente, ad esempio, crea una variabile globale di tipo String denominata `str1`, quindi crea una variabile locale con lo stesso nome all'interno della funzione `scopeTest()`. L'istruzione `trace` all'interno della funzione genera il valore locale della variabile, ma all'esterno della funzione genera il valore globale della variabile.

```
var str1:String = "Global";
function scopeTest ()
{
    var str1:String = "Local";
    trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

La variabili ActionScript, a differenza delle variabili C++ e Java, non possono avere un'area di validità a livello di blocco. Un blocco di codice è qualunque gruppo di istruzioni compreso tra una parentesi graffa di apertura ({) e una di chiusura (}). In alcuni linguaggi di programmazione, come C++ e Java, le variabili dichiarate in un blocco di codice non sono disponibili all'esterno del blocco. Questa restrizione viene definita "area di validità a livello di blocco" e non esiste in ActionScript. Una variabile dichiarata all'interno di un blocco di codice è disponibile non solo in tale blocco ma anche nelle altre parti della funzione alla quale esso appartiene. Ad esempio, la funzione seguente contiene variabili definite in vari blocchi di codice, ciascuna delle quali è disponibile nell'intera funzione.

```
function blockTest (testArray:Array)
{
    var numElements:int = testArray.length;
    if (numElements > 0)
    {
        var elemStr:String = "Element #";
        for (var i:int = 0; i < numElements; i++)
        {
            var valueStr:String = i + ": " + testArray[i];
            trace(elemStr + valueStr);
        }
        trace(elemStr, valueStr, i); // all still defined
    }
    trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

Un'implicazione interessante dell'assenza dell'area di validità a livello di blocco è la possibilità di leggere o scrivere una variabile prima che venga dichiarata, a condizione che la dichiarazione sia inclusa prima della fine della funzione. Ciò è possibile grazie a una tecnica chiamata *hoisting*, in base alla quale il compilatore sposta tutte le dichiarazioni di variabili all'inizio della funzione. Ad esempio, il codice seguente viene compilato correttamente anche se la funzione iniziale `trace()` della variabile `num` ha luogo prima che tale variabile sia dichiarata:

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

Il compilatore tuttavia non esegue l'hoisting per le istruzioni di assegnazione, motivo per cui l'istruzione iniziale `trace()` di `num` produce `NaN` (not a number), che è il valore predefinito per le variabili del tipo di dati Number. Pertanto, è possibile assegnare valori alle variabili anche prima che siano dichiarate, come nell'esempio seguente:

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

Valori predefiniti

Per *valore predefinito* si intende il valore che una variabile contiene prima che ne venga impostato il valore. Una variabile viene *inizializzata* quando se ne imposta il valore per la prima volta. Se si dichiara una variabile ma non se ne imposta il valore, tale variabile rimane *non inizializzata*. Il valore di una variabile non inizializzata dipende dal suo tipo di dati. La tabella seguente descrive i valori predefiniti delle variabili, organizzati per tipo di dati:

Tipo di dati	**Valore predefinito**
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`
uint	0
Non dichiarato (equivalente all'annotazione di tipo `*`)	`undefined`
Tutte le altre classi, comprese quelle definite dall'utente.	`null`

Per le variabili di tipo Number, il valore predefinito è `NaN` (not a number), che è un valore speciale definito dallo standard IEEE-754 per indicare un valore che non rappresenta un numero.

Se dichiarate una variabile senza dichiararne il tipo di dati, viene applicato il tipo predefinito `*`, che indica che la variabile è priva di tipo. Inoltre, se una variabile senza tipo non viene inizializzata con un valore, il valore predefinito è `undefined`.

Per i tipi di dati diversi da Boolean, Number, int e uint, il valore predefinito di qualunque variabile non inizializzata è `null`, sia per le classi definite da ActionScript 3.0 che per quelle personalizzate definite dall'utente.

Il valore `null` non è valido per le variabili di tipo Boolean, Number, int o uint. Se tentate di assegnarlo a una di queste variabili, viene convertito nel valore predefinito per il tipo di dati corrispondente. È invece possibile assegnare il valore `null` alle variabili di tipo Object. Se tentate di assegnare il valore `undefined` a una variabile Object, il valore viene convertito in `null`.

Per le variabili di tipo Number esiste una funzione speciale di primo livello denominata `isNaN()`, che restituisce il valore booleano `true` se la variabile non è un numero oppure `false` in caso contrario.

Tipi di dati

Un *tipo di dati* definisce un set di valori. Ad esempio, il tipo di dati Boolean comprende esattamente due valori: `true` e `false`. Oltre a Boolean, ActionScript 3.0 definisce vari altri tipi di dati di uso comune, come String, Number e Array. È inoltre possibile creare tipi di dati personalizzati utilizzando le classi o le interfacce per definire un set di valori specifici. In ActionScript 3.0 tutti i valori, di base o complessi, sono oggetti.

Un *valore di base* appartiene a uno dei tipi di dati seguenti: Boolean, int, Number, String e uint. I valori di base solitamente consentono di lavorare con maggiore rapidità rispetto a quando si utilizzano valori complessi, perché ActionScript memorizza i valori di base con un metodo speciale che permette di ottimizzare la memoria e la velocità di elaborazione.

***Nota:** per i lettori interessati ai dettagli tecnici: ActionScript memorizza i valori di base internamente come oggetti immutabili. Questo tipo di memorizzazione permette di passare un riferimento anziché un valore, ottenendo lo stesso effetto. In questo modo è possibile ridurre l'uso della memoria e aumentare la velocità, dal momento che i riferimenti hanno dimensioni notevolmente inferiori ai valori veri e propri.*

Un *valore complesso* è qualunque valore diverso da un valore di base. I tipi di dati che definiscono set di valori complessi includono Array, Date, Error, Function RegExp, XML e XMLList.

Molti linguaggi di programmazione fanno distinzione tra i valori di base e i relativi oggetti wrapper. Java, ad esempio, prevede un valore di base int e la classe wrapper java.lang.Integer che lo contiene. I valori di base Java non sono oggetti, ma i relativi wrapper lo sono, il che rende i valori di base utili per determinate operazioni e gli oggetti wrapper più indicati per altre. In ActionScript 3.0, i valori di base e i relativi oggetti sono, ai fini pratici, indistinguibili. Tutti i valori, compresi quelli di base, sono oggetti. Il runtime tratta questi tipi di base come casi speciali che si comportano come oggetti ma non richiedono il lavoro di elaborazione solitamente associato alla creazione di oggetti. Ciò significa che le due righe di codice seguenti sono equivalenti:

```
var someInt:int = 3;
var someInt:int = new int(3);
```

Tutti i tipi di dati di base e complessi elencati sopra sono definiti dalle classi di base di ActionScript 3.0. Tali classi consentono di creare oggetti utilizzando valori letterali anziché l'operatore `new`. Ad esempio, è possibile creare un array specificando un valore letterale oppure la funzione di costruzione della classe Array, come nell'esempio seguente:

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

Verifica del tipo

La verifica del tipo può avvenire sia in fase di compilazione che di runtime. I linguaggi che prevedono tipi statici, come C++ e Java, eseguono la verifica del tipo in fase di compilazione, mentre quelli che prevedono tipi dinamici, come Smalltalk e Python, gestiscono la verifica del tipo in fase di runtime. ActionScript 3.0 appartiene alla seconda categoria ed esegue la verifica del tipo in fase di runtime, ma supporta anche la verifica in fase di compilazione tramite una modalità speciale del compilatore definita *modalità rigorosa*. Nella modalità rigorosa la verifica del tipo avviene sia in fase di compilazione che di runtime, mentre in modalità standard viene eseguita solo in fase di runtime.

I linguaggi con assegnazione dinamica del tipo offrono una grande flessibilità in fase di strutturazione del codice, tuttavia comportano il rischio che gli errori di tipo si manifestino in fase di runtime. Al contrario, i linguaggi con tipi statici segnalano tutti gli errori di tipo in fase di compilazione ma richiedono che le informazioni sui tipi siano già note in tale fase.

Verifica del tipo in fase di compilazione

La verifica del tipo in fase di compilazione è spesso preferita nei progetti di grandi dimensioni perché, con il crescere delle dimensioni del progetto, la flessibilità dei tipi di dati perde generalmente importanza a favore della possibilità di rilevare tutti gli errori di tipo con la massima tempestività. È per questo motivo che, per impostazione predefinita, il compilatore ActionScript in Flash Professional e in Flash Builder è configurato per essere eseguito in modalità rigorosa.

Adobe Flash Builder
Potete disabilitare la modalità rigorosa in Flash Builder tramite le impostazioni del compilatore ActionScript nella finestra di dialogo delle proprietà del progetto.

Per eseguire la verifica del tipo in fase di compilazione, il compilatore deve conoscere il tipo di dati delle variabili o delle espressioni contenute nel codice. Per dichiarare esplicitamente il tipo di dati di una variabile, aggiungete al nome della variabile l'operatore due punti (`:`) seguito dal tipo di dati come suffisso. Per associare un tipo di dati a un parametro, utilizzate l'operatore due punti seguito dal tipo di dati. Ad esempio, il codice seguente aggiunge il tipo di dati al parametro `xParam` e dichiara una variabile `myParam` con un tipo di dati esplicito:

```
function runtimeTest(xParam:String)
{
    trace(xParam);
}
var myParam:String = "hello";
runtimeTest(myParam);
```

In modalità rigorosa, il compilatore ActionScript segnala i casi di mancata corrispondenza del tipo come errori di compilazione. Ad esempio, il codice seguente dichiara un parametro di funzione `xParam`, del tipo Object, ma successivamente tenta di assegnare valori del tipo String e Number allo stesso parametro. In modalità rigorosa, questo codice genera un errore del compilatore.

```
function dynamicTest(xParam:Object)
{
    if (xParam is String)
    {
        var myStr:String = xParam; // compiler error in strict mode
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam; // compiler error in strict mode
        trace("Number: " + myNum);
    }
}
```

Anche in modalità rigorosa, tuttavia, è possibile scegliere di non eseguire la verifica del tipo in fase di compilazione non specificando il tipo sul lato destro di un'istruzione di assegnazione. Per contrassegnare una variabile o un'espressione come priva di tipo, è possibile omettere l'annotazione di tipo o utilizzare l'annotazione speciale dell'asterisco (`*`). Ad esempio, se il parametro `xParam` contenuto nel codice precedente viene modificato omettendo l'annotazione di tipo, il codice viene compilato anche in modalità rigorosa:

```
function dynamicTest(xParam)
{
    if (xParam is String)
    {
        var myStr:String = xParam;
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam;
        trace("Number: " + myNum);
    }
}
dynamicTest(100)
dynamicTest("one hundred");
```

Verifica del tipo in fase di runtime

Il controllo del tipo in fase di runtime viene eseguito in ActionScript 3.0 sia nella modalità di compilazione rigorosa che in modalità standard. Considerate una situazione in cui il valore 3 viene passato come argomento a una funzione per la quale è previsto array. In modalità rigorosa, il compilatore genera un errore perché il valore 3 non è compatibile con il tipo di dati Array. Se disattivate la modalità rigorosa e attivate la modalità standard, il compilatore non rileva la mancata corrispondenza del tipo, ma la verifica in fase di runtime genera un errore runtime.

L'esempio seguente mostra una funzione denominata `typeTest()` per la quale è previsto un argomento Array ma alla quale viene invece passato il valore 3. Questa situazione genera un errore runtime in modalità standard perché il valore 3 non è un membro del tipo di dati dichiarato del parametro (Array).

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

In determinati casi può essere generato un errore runtime di tipo anche in modalità rigorosa. Questa situazione può verificarsi quando, pur essendo attiva la modalità rigorosa, si sceglie di non eseguire la verifica del tipo in fase di compilazione specificando una variabile senza tipo. L'uso di una variabile senza tipo non elimina la verifica del tipo, bensì la rimanda alla fase di runtime. Ad esempio, se la variabile `myNum` dell'esempio precedente non ha un tipo di dati dichiarato, il compilatore non è in grado di rilevare la mancata corrispondenza del tipo ma il codice genera un errore runtime perché esegue un confronto tra il valore runtime di `myNum`, che è impostato su 3 dall'istruzione di assegnazione, e il tipo di dati di `xParam`, che è Array.

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

La verifica in fase di runtime consente anche un uso più flessibile dell'ereditarietà rispetto a quella in fase di compilazione. Rimandando la verifica del tipo alla fase di runtime, la modalità standard permette di fare riferimento alle proprietà di una sottoclasse anche se eseguite un *upcast*, ovvero utilizzate una classe di base per dichiarare il tipo di un'istanza di classe ma una sottoclasse per creare l'istanza vera e propria. Ad esempio, potete creare una classe denominata ClassBase che può essere estesa (non è possibile estendere le classi con attributo `final`):

```
class ClassBase
{
}
```

Successivamente potete creare una sottoclasse di ClassBase denominata ClassExtender, dotata di una proprietà di nome `someString`, come nel codice seguente:

```
class ClassExtender extends ClassBase
{
    var someString:String;
}
```

Utilizzando entrambe le classi, è possibile definire un'istanza di classe dichiarata mediante il tipo di dati ClassBase ma creata utilizzando la funzione di costruzione ClassExtender. Un'operazione di upcast è considerata sicura perché la classe di base non contiene proprietà o metodi che non sono disponibili nella sottoclasse.

```
var myClass:ClassBase = new ClassExtender();
```

Una sottoclasse, al contrario, può contenere proprietà o metodi che non sono presenti nella rispettiva classe di base. Ad esempio, la classe ClassExtender contiene la proprietà `someString`, che non esiste nella classe ClassBase. Nella modalità standard di ActionScript 3.0, è possibile fare riferimento a questa proprietà usando l'istanza `myClass` senza generare un errore in fase di compilazione, come mostra l'esempio seguente:

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

L'operatore is

L'operatore `is` consente di verificare se una variabile o un'espressione è un membro di un determinato tipo di dati. Nelle versioni precedenti di ActionScript, la stessa funzionalità era fornita dall'operatore `instanceof`, che tuttavia in ActionScript 3.0 non va utilizzato per verificare l'appartenenza a un tipo di dati. È necessario utilizzare l'operatore `is` anziché `instanceof` per eseguire la verifica manuale del tipo, perché l'espressione `x instanceof y` si limita a controllare se nella catena di prototipi di `x` è presente `y` (in ActionScript 3.0, la catena di prototipi non fornisce un quadro completo della gerarchia di ereditarietà).

L'operatore `is` invece esamina la gerarchia di ereditarietà effettiva e consente di verificare non solo se un oggetto è un'istanza di una particolare classe, ma anche se è un'istanza di una classe che implementa una particolare interfaccia. L'esempio seguente crea un'istanza della classe Sprite denominata `mySprite` e utilizza l'operatore `is` per verificare se `mySprite` è un'istanza delle classi Sprite e DisplayObject e se implementa l'interfaccia IEventDispatcher:

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite); // true
trace(mySprite is DisplayObject);// true
trace(mySprite is IEventDispatcher); // true
```

L'operatore `is` controlla la gerarchia di ereditarietà e segnala correttamente che `mySprite` è compatibile con le classi Sprite e DisplayObject (Sprite è una sottoclasse della classe DisplayObject). Inoltre, verifica se `mySprite` eredita da qualunque classe che implementa l'interfaccia IEventDispatcher. Poiché la classe Sprite eredita dalla classe EventDispatcher, che implementa l'interfaccia IEventDispatcher, l'operatore `is` segnala correttamente che `mySprite` implementa la stessa interfaccia.

Il codice dell'esempio seguente esegue gli stessi controlli di quello precedente, ma con l'operatore `instanceof` al posto di `is`. L'operatore `instanceof` identifica correttamente `mySprite` come istanza di Sprite o DisplayObject, ma restituisce `false` quando viene utilizzato per verificare se `mySprite` implementa l'interfaccia IEventDispatcher.

```
trace(mySprite instanceof Sprite); // true
trace(mySprite instanceof DisplayObject);// true
trace(mySprite instanceof IEventDispatcher); // false
```

L'operatore as

Anche l'operatore `as` consente di verificare se un'espressione è un membro di un determinato tipo di dati. A differenza di `is`, tuttavia, `as` non restituisce un valore booleano, bensì il valore dell'espressione (invece di `true`) oppure `null` (invece di `false`). L'esempio seguente mostra il risultato che si ottiene utilizzando l'operatore `as` anziché `is` per controllare semplicemente se un'istanza Sprite è membro dei tipi di dati DisplayObject, IEventDispatcher e Number.

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);          // null
```

Quando si usa l'operatore `as`, l'operando sulla destra deve essere un tipo di dati. Un eventuale tentativo di utilizzare un'espressione diversa da un tipo di dati come operando sulla destra produrrebbe un errore.

Classi dinamiche

Una classe *dinamica* definisce un oggetto che può essere alterato in fase di runtime aggiungendo o modificandone le proprietà e i metodi. Un classe non dinamica, ad esempio la classe String, si definisce *chiusa* (sealed in inglese). Non è possibile aggiungere proprietà o metodi a una classe chiusa in runtime.

Per creare una classe dinamica si utilizza l'attributo `dynamic` nella relativa dichiarazione. Ad esempio, il codice seguente crea una classe dinamica denominata `Protean`:

```
dynamic class Protean
{
    private var privateGreeting:String = "hi";
    public var publicGreeting:String = "hello";
    function Protean()
    {
        trace("Protean instance created");
    }
}
```

Se in seguito create un'istanza della classe `Protean`, potete aggiungervi proprietà o metodi esternamente alla definizione della classe. Ad esempio, il codice seguente crea un'istanza della classe `Protean` e vi aggiunge due proprietà denominate rispettivamente `aString` e `aNumber`:

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

Le proprietà aggiunte a un'istanza di una classe dinamica sono entità runtime, quindi le verifiche di qualsiasi tipo vengono eseguite in fase di runtime. Non è possibile aggiungere un'annotazione di tipo a una proprietà creata in questo modo.

È anche possibile aggiungere un metodo all'istanza `myProtean` definendo una funzione e associandola a una proprietà dell'istanza `myProtean`. Il codice seguente sposta l'istruzione trace in un metodo denominato `traceProtean()`:

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
    trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

I metodi creati in questo modo, tuttavia, non hanno accesso a eventuali proprietà o metodi privati della classe Protean. Inoltre, anche i riferimenti a proprietà o metodi pubblici della classe `Protean` devono essere qualificati con la parola chiave `this` o con il nome della classe. Nell'esempio seguente, il metodo `traceProtean()` tenta di accedere alle variabili private e pubbliche della classe `Protean`.

```
myProtean.traceProtean = function ()
{
    trace(myProtean.privateGreeting); // undefined
    trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

Descrizione dei tipi di dati

I tipi di dati di base sono Boolean, int, Null, Number, String, uint e void. Le classi principali di ActionScript definiscono anche i seguenti tipi di dati complessi: Object, Array, Date, Error, Function, RegExp, XML e XMLList.

Tipo di dati Boolean

Il tipo di dati Boolean include due valori: `true` e `false`. Nessun altro valore è consentito per le variabili di questo tipo. Il valore predefinito di una variabile Boolean dichiarata ma non inizializzata è `false`.

Tipo di dati int

Il tipo di dati int è memorizzato internamente come numero intero a 32 bit e include la serie di numeri interi da

-2.147.483.648 (-2^{31}) a 2.147.483.647 ($2^{31} - 1$). Le versioni precedenti di ActionScript offrivano solo il tipo di dati Number, utilizzato sia per i numeri interi che per quelli a virgola mobile. In ActionScript 3.0 è ora possibile accedere ai tipi macchina di basso livello per i numeri interi a 32 bit con e senza segno. Se la variabile non deve usare numeri a virgola mobile, l'impiego del tipo di dati int al posto di Number garantisce maggiore velocità ed efficienza.

Per i numeri interi al di fuori dell'intervallo dei valori minimo e massimo, utilizzate il tipo di dati Number, che è in grado di gestire i valori compresi tra -9.007.199.254.740.992 e 9.007.199.254.740.992 (valori interi a 53 bit). Il valore predefinito per le variabili del tipo di dati int è 0.

Tipo di dati Null

Il tipo di dati Null contiene un solo valore, `null`. Questo è il valore predefinito per il tipo di dati String e per tutte le classi che definiscono tipi di dati complessi, compresa la classe Object. Nessuno degli altri tipi di dati di base, come Boolean, Number, int e uint, contiene il valore `null`. In fase di runtime, il valore `null` viene convertito nel valore predefinito appropriato se tentate di assegnare `null` a variabili di tipo Boolean, Number, int o uint. Non è possibile utilizzare questo tipo di dati come annotazione di tipo.

Tipo di dati Number

In ActionScript 3.0, il tipo di dati Number può rappresentare numeri interi, numeri interi senza segno e numeri a virgola mobile. Tuttavia, per ottimizzare le prestazioni, è meglio utilizzare il tipo di dati Number solo per i numeri interi maggiori dei numeri a 32 bit memorizzabili dai tipi `int` e `uint`, oppure per i numeri a virgola mobile. Per memorizzare un numero a virgola mobile, includete il punto dei decimali nel numero. Se omettete il punto dei decimali, il valore viene memorizzato come numero intero.

Il tipo di dati Number utilizza il formato a doppia precisione a 64 bit specificato dallo standard IEEE 754 per i calcoli aritmetici binari a virgola mobile, che prescrive come devono essere memorizzati i numeri a virgola mobile utilizzando i 64 bit disponibili. Un bit serve per designare il numero come positivo o negativo, undici bit sono utilizzati per l'esponente (memorizzato come base 2) e i rimanenti 52 bit servono per memorizzare il *significante* (chiamato anche *mantissa*), ovvero il numero che viene elevato alla potenza indicata dall'esponente.

Utilizzando alcuni dei bit per memorizzare un esponente, il tipo di dati Number può memorizzare numeri a virgola mobile notevolmente più grandi che non se utilizzasse tutti i bit per il significante. Ad esempio, se il tipo di dati Number utilizzasse tutti i 64 bit per memorizzare il significante, il numero più grande che potrebbe memorizzare sarebbe 2^{65} - 1. Utilizzando 11 bit per memorizzare un esponente, il tipo di dati Number può elevare il significante fino a una potenza di ^{1023}.

I valori massimo e minimo che il tipo Number può rappresentare sono memorizzati in proprietà statiche della classe Number denominate `Number.MAX_VALUE` e `Number.MIN_VALUE`.

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

Un intervallo di numeri così ampio va a scapito della precisione. Il tipo di dati Number utilizza 52 bit per memorizzare il significante, con il risultato che i numeri che richiedono più di 52 bit per essere rappresentati con precisione, ad esempio la frazione 1/3, sono solo approssimazioni. Se l'applicazione richiede una precisione assoluta per i numeri decimali, è necessario utilizzare un software in grado di implementare i calcoli aritmetici a virgola mobile decimali anziché quelli binari.

Quando memorizzate valori interi con il tipo di dati Number, vengono utilizzati solo i 52 bit del significante. Il tipo di dati Number usa questi 52 bit e un bit speciale nascosto per rappresentare i numeri interi da -9.007.199.254.740.992 (-2^{53}) a 9.007.199.254.740.992 (2^{53}).

Il valore `NaN` non solo è il valore predefinito per le variabili di tipo `Number`, ma è anche il risultato di qualunque operazione che deve restituire un numero e invece restituisce un valore diverso. Ad esempio, se tentate di calcolare la radice quadrata di un numero negativo, il risultato è `NaN`. Altri valori Number speciali sono *infinito positivo* e *infinito negativo* .

***Nota:** il risultato della divisione per `0` è `NaN` solo se anche il divisore è `0`. La divisione per `0` dà il risultato `infinito` se il dividendo è positivo oppure `-infinito` se è negativo.*

Tipo di dati String

Il tipo di dati String rappresenta una sequenza di caratteri a 16 bit. Le stringhe vengono memorizzate internamente come caratteri Unicode, utilizzando il formato UTF-16. Come nel linguaggio di programmazione Java, le stringhe sono valori immutabili. Un'operazione su un valore String restituisce una nuova istanza della stringa. Il valore predefinito di una variabile dichiarata con il tipo di dati String è `null`. Il valore `null` non è equivalente a una stringa vuota (`""`). Il valore `null` indica che la variabile non contiene alcun valore, mentre la stringa vuota indica che il valore della variabile è un tipo String che non contiene nessun carattere.

Tipo di dati uint

Il tipo di dati uint è memorizzato internamente come numero intero a 32 bit senza segno e include la serie di numeri interi da 0 a 4.294.967.295 (2^{32} - 1). Utilizzate il tipo di dati uint in circostanze speciali che richiedono numeri interi non negativi. Ad esempio, va necessariamente utilizzato per rappresentare i valori di colore dei pixel, perché il tipo di dati int ha un bit di segno interno che non è appropriato per la gestione dei valori di colore. Per i numeri interi maggiori del valore uint massimo, utilizzate il tipo di dati Number, che è in grado di gestire i valori interi a 53 bit. Il valore predefinito per le variabili del tipo di dati uint è 0.

Tipo di dati void

Il tipo di dati void contiene un solo valore, `undefined`. Nelle versioni precedenti di ActionScript, `undefined` era il valore predefinito per le istanze della classe Object. In ActionScript 3.0, il valore predefinito delle istanze Object è `null`. Se tentate di assegnare il valore `undefined` a un'istanza della classe Object, il valore viene convertito in `null`. È possibile assegnare il valore `undefined` solo alle variabili senza tipo, ovvero quelle che sono prive di un'annotazione di tipo oppure utilizzano l'asterisco (`*`) come annotazione. Potete utilizzare `void` solo come annotazione del tipo restituito.

Tipo di dati Object

Il tipo di dati Object è definito dalla classe Object che viene utilizzata come classe di base per tutte le definizioni di classe in ActionScript. La versione ActionScript 3.0 del tipo di dati Object differisce dalle versioni precedenti per tre aspetti. Innanzi tutto, il tipo di dati Object non è più il tipo predefinito assegnato alle variabili prive di annotazione di tipo. In secondo luogo, il tipo Object non include più il valore `undefined`, che era il valore predefinito di tutte le istanze Object. Infine, in ActionScript 3.0 il valore predefinito delle istanze della classe Object è `null`.

Nelle versioni precedenti di ActionScript, a una variabile priva di annotazione di tipo veniva assegnato automaticamente il tipo di dati Object. ActionScript 3.0, al contrario, prevede la possibilità di variabili effettivamente prive di tipo. Pertanto, le variabili per le quali non viene fornita un'annotazione di tipo vengono ora considerate senza tipo. Se preferite rendere esplicito a chi leggerà il codice che la vostra intenzione è effettivamente quella di lasciare una variabile priva di tipo, potete utilizzare il simbolo di asterisco (`*`) come annotazione di tipo, che equivale a omettere l'annotazione. L'esempio seguente mostra due istruzioni equivalenti, che dichiarano entrambe una variabile senza tipo `x`:

```
var x
var x:*
```

Solo le variabili senza tipo possono contenere il valore `undefined`. Se tentate di assegnare il valore `undefined` a una variabile che appartiene a un tipo di dati, il runtime converte il valore `undefined` nel valore predefinito del tipo di dati in questione. Per le istanze del tipo di dati Object, il valore predefinito è `null`. Pertanto, se tentate di assegnare `undefined` a un'istanza Object il valore viene convertito in `null`.

Conversione del tipo di dati

Una conversione del tipo di dati ha luogo quando un valore viene trasformato in un valore appartenente a un tipo di dati diverso. Le conversioni di tipo possono essere *implicite* o *esplicite*. Una conversione implicita, chiamata anche *assegnazione forzata* (in inglese, coercion), viene talvolta eseguita in fase di runtime. Ad esempio, se il valore 2 è assegnato a una variabile del tipo di dati Boolean, il valore 2 viene convertito nel valore booleano `true` prima che il valore venga assegnato alla variabile. La conversione esplicita, chiamata anche *inserimento* (in inglese, casting), ha luogo quando il codice passa al compilatore l'istruzione di elaborare una variabile di un tipo di dati particolare come se appartenesse a un tipo di dati diverso. Quando l'operazione riguarda valori di base, l'inserimento ha effettivamente il risultato di convertire i valori da un tipo di dati a un altro. Per inserire un oggetto in un tipo diverso, occorre racchiudere il nome dell'oggetto tra parentesi e farlo precedere dal nome del nuovo tipo. Il codice seguente, ad esempio accetta un valore booleano e lo inserisce in un numero intero:

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

Conversione implicita

La conversione implicita viene eseguita in fase di runtime in una serie di casi:

- Nelle istruzioni di assegnazione
- Quando i valori vengono passati come argomenti di funzioni
- Quando i valori vengono restituiti da funzioni
- Nelle espressioni che utilizzano determinati operatori, ad esempio l'operatore di addizione (+)

  Per i tipi definiti dall'utente, la conversione implicita ha luogo quando il valore da convertire è un'istanza della classe di destinazione o di una classe derivata da essa. Se una conversione implicita ha esito negativo, viene generato un errore. Ad esempio, il codice seguente contiene una conversione implicita corretta e una con esito negativo:

  ```
  class A {}
  class B extends A {}

  var objA:A = new A();
  var objB:B = new B();
  var arr:Array = new Array();

  objA = objB; // Conversion succeeds.
  objB = arr; // Conversion fails.
  ```

  Per i tipi di base, le conversioni implicite vengono gestite chiamando gli stessi algoritmi di conversione interni che vengono chiamati dalle funzioni di conversione esplicita.

Conversione esplicita

È utile ricorrere alla conversione esplicita (detta anche inserimento o casting) quando compilate il codice in modalità rigorosa, in particolare quando volete evitare che una mancata corrispondenza di tipo generi un errore in fase di compilazione. Questa situazione può verificarsi quando si ha la certezza che i valori saranno convertiti correttamente in fase di runtime mediante l'assegnazione forzata. Ad esempio, quando utilizzate i dati ricevuti da un form, potete scegliere di ricorrere all'assegnazione forzata per convertire determinati valori di stringa in valori numerici. Il codice seguente genera un errore di compilazione anche se, in modalità standard, verrebbe eseguito senza problemi.

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

Se desiderate continuare a utilizzare la modalità rigorosa ma volete convertire la stringa in numero intero, potete utilizzare la conversione esplicita, come nell'esempio seguente:

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

Inserimento nei tipi int, uint e Number

Potete inserire qualunque tipo di dati nei tre seguenti tipi numerici: int, uint e Number. Se per vari motivi il numero non può essere convertito, il valore predefinito 0 viene assegnato ai tipi di dati int e uint, mentre il valore predefinito `NaN` viene assegnato al tipo di dati Number. Se convertite un valore booleano in numero, `true` diventa 1 e `false` diventa 0.

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

I valori stringa che contengono solo cifre possono essere convertiti correttamente in uno dei tre tipi numerici, i quali consentono inoltre di convertire stringhe che assomigliano a numeri negativi o che rappresentano un valore esadecimale (ad esempio, `0x1A`). Il processo di conversione ignora gli eventuali caratteri di spazio vuoto presenti all'inizio e alla fine del valore stringa. È anche possibile convertire le stringhe che hanno l'aspetto di numeri a virgola mobile utilizzando `Number()`. Se viene incluso il punto dei decimali, `uint()` e `int()` restituiscono un numero intero in cui i caratteri che seguono il punto sono troncati. Ad esempio, i valori stringa seguenti possono essere inseriti in valori numerici:

```
trace(uint("5")); // 5
trace(uint("-5")); // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 ")); // 27
trace(uint("3.7")); // 3
trace(int("3.7")); // 3
trace(int("0x1A")); // 26
trace(Number("3.7")); // 3.7
```

I valori stringa che contengono caratteri non numerici restituiscono 0 se vengono convertiti con `int()` o `uint()` oppure `NaN` se convertiti con `Number()`. Il processo di conversione ignora lo spazio vuoto all'inizio e alla fine, ma restituisce 0 o `NaN` se una stringa contiene uno spazio vuoto tra due numeri.

```
trace(uint("5a")); // 0
trace(uint("ten")); // 0
trace(uint("17 63")); // 0
```

In ActionScript 3.0, la funzione `Number()` non supporta più gli ottali, ovvero i numeri a base 8. Se passate una stringa con uno zero iniziale alla funzione `Number()` di ActionScript 2.0, il numero viene interpretato come ottale e convertito nell'equivalente decimale. Lo stesso non vale per la funzione `Number()` di ActionScript 3.0, che invece ignora lo zero iniziale. Ad esempio, il codice seguente genera un output diverso se viene compilato con versioni differenti di ActionScript:

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

L'inserimento non è necessario quando un valore di un tipo numerico viene assegnato a una variabile di un altro tipo numerico. Anche in modalità rigorosa, i tipi numerici vengono convertiti implicitamente in altri tipi numerici. Ciò significa che in alcuni casi potete ottenere valori imprevisti quando viene superato l'intervallo di un tipo numerico. Gli esempi seguenti vengono compilati correttamente in modalità rigorosa, ma alcuni di essi generano valori imprevisti:

```
var myUInt:uint = -3; // Assign int/Number value to uint variable
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // Assign int/uint value to Number variable
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // Assign Number value to uint variable
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // Assign uint/Number value to int variable
trace(myInt); // -2147483648
```

La tabella che segue riepiloga i risultati dell'inserimento nei tipi Number, int o uint da altri tipi di dati.

Tipo di dati o valore	Risultato della conversione in Number, int o uint
Boolean	Se il valore è `true`, 1; altrimenti, 0.
Date	La rappresentazione interna dell'oggetto Date, che corrisponde al numero di millisecondi trascorsi dalla mezzanotte del 1 gennaio 1970 (ora universale).
`null`	0
Object	Se l'istanza è `null` e viene convertita in Number, `NaN`; altrimenti, 0.
String	Un numero se la stringa può essere convertita in un numero; in caso contrario, `NaN` viene convertito in Number o 0 nel caso di conversione in int o uint.
`undefined`	Se convertito in Number, `NaN`; se convertito in int o uint, 0.

Inserimento nel tipo Boolean

L'inserimento di un tipo di dati numerico (uint, int o Number) nel tipo Boolean restituisce `false` se il valore numerico è 0 e `true` in tutti gli altri casi. Per il tipo di dati Number, anche il valore `NaN` restituisce `false`. L'esempio seguente illustra i risultati di un inserimento dei numeri -1, 0 e 1:

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
    trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

L'output dell'esempio mostra che dei tre numeri solo 0 restituisce il valore `false`:

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

L'inserimento di un valore String nel tipo Boolean restituisce `false` se la stringa è `null` o vuota (`""`) e `true` in tutti gli altri casi.

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

L'inserimento di un'istanza della classe Object nel tipo Boolean restituisce `false` se l'istanza è `null` e `true` in tutti gli altri casi:

```
var myObj:Object; // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object(); // instantiate
trace(Boolean(myObj)); // true
```

Le variabili Boolean vengono elaborate in un modo speciale in modalità rigorosa, ovvero è possibile assegnare valori di qualunque tipo di dati a una variabile Boolean senza eseguire l'inserimento. L'assegnazione forzata implicita da tutti i tipi di dati al tipo Boolean viene eseguita anche in modalità rigorosa. In altri termini, contrariamente a quasi tutti gli altri tipi di dati, l'inserimento nel tipo Boolean non è necessario per evitare gli errori della modalità rigorosa. Gli esempi seguenti vengono compilati correttamente in modalità rigorosa e danno i risultati previsti anche in fase di runtime:

```
var myObj:Object = new Object(); // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

La tabella che segue riepiloga i risultati dell'inserimento nel tipo Boolean da altri tipi di dati.

Tipo di dati o valore	**Risultato della conversione in Boolean**
String	`false` se il valore è `null` o una stringa vuota (`""`); `true` negli altri casi.
`null`	`false`
Number, int o uint	`false` se il valore è `NaN` o 0; `true` negli altri casi.
Object	`false` se l'istanza è `null`; `true` negli altri casi.

Inserimento nel tipo String

L'inserimento di un tipo di dati numerico nel tipo di dati String restituisce una rappresentazione del numero sotto forma di stringa. Se invece inserite un valore booleano nel tipo di dati String, viene restituita la stringa `"true"` se il valore è `true` e la stringa `"false"` se il valore è `false`.

Se inserite un'istanza della classe Object nel tipo di dati String, viene restituita la stringa `"null"` se l'istanza è `null`. In tutti gli altri casi, l'inserimento della classe Object nel tipo di dati String restituisce la stringa `"[object Object]"` .

Quando inserite un'istanza della classe Array nel tipo String, viene restituita una stringa composta da un elenco delimitato da virgole di tutti gli elementi dell'array. Ad esempio, il seguente inserimento nel tipo di dati String restituisce una sola stringa contenente tutti e tre gli elementi dell'array:

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

Se inserite un'istanza della classe Date nel tipo di dati String, viene restituita una rappresentazione della data contenuta nell'istanza sotto forma di stringa. Ad esempio, il codice seguente restituisce una rappresentazione sotto forma di stringa dell'istanza della classe Date. L'output mostra i risultati per l'ora legale del Pacifico:

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

La tabella che segue riepiloga i risultati dell'inserimento nel tipo String da altri tipi di dati.

Tipo di dati o valore	Risultato della conversione in String
Array	Una stringa composta da tutti gli elementi dell'array.
Boolean	`"true"` o `"false"`
Date	La rappresentazione in formato stringa dell'oggetto Date.
`null`	`"null"`
Number, int o uint	La rappresentazione in formato stringa del numero.
Object	Se l'istanza è null, `"null"`; negli altri casi, `"[object Object]"`.

Sintassi

La sintassi di un linguaggio definisce una serie di regole che devono essere rispettate quando si scrive codice eseguibile.

Distinzione tra maiuscole e minuscole

Nel linguaggio ActionScript 3.0 viene fatta distinzione tra maiuscole e minuscole. Gli identificatori che presentano gli stessi caratteri ma lettere minuscole e maiuscole differenti vengono considerati identificatori diversi. Ad esempio, il codice seguente crea due variabili diverse:

```
var num1:int;
var Num1:int;
```

Sintassi del punto

L'operatore punto (.) permette di accedere alle proprietà e ai metodi di un oggetto. Utilizzando la sintassi del punto, potete fare riferimento a una proprietà o a un metodo di una classe specificando un nome di istanza seguito dall'operatore punto e dal nome della proprietà o del metodo. Ad esempio, considerate la seguente definizione di classe:

```
class DotExample
{
    public var prop1:String;
    public function method1():void {}
}
```

Mediante la sintassi del punto, potete accedere alla proprietà `prop1` e al metodo `method1()` utilizzando il nome di istanza creato nel codice seguente:

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = "hello";
myDotEx.method1();
```

Potete ricorrere alla sintassi del punto anche per la definizione dei pacchetti. L'operatore punto viene utilizzato per fare riferimento ai pacchetti nidificati. Ad esempio, la classe EventDispatcher si trova in un pacchetto denominato events che è nidificato all'interno del pacchetto flash. Per fare riferimento al pacchetto events, usate l'espressione seguente:

```
flash.events
```

Potete fare riferimento alla classe EventDispatcher anche mediante l'espressione seguente:

```
flash.events.EventDispatcher
```

Sintassi della barra

La sintassi della barra non è supportata in ActionScript 3.0 ma era utilizzata in versioni di ActionScript precedenti per indicare il percorso di un clip filmato o una variabile.

Valori letterali

Per *valore letterale* si intende un valore che compare direttamente nel codice. I seguenti esempi sono valori letterali:

```
17
"hello"
-3
9.4
null
undefined
true
false
```

I valori letterali possono essere raggruppati a formare valori letterali composti. I valori letterali di array sono racchiusi tra parentesi quadre (`[]`) e utilizzano la virgola per separare gli elementi dell'array.

Un valore letterale array può essere utilizzato per inizializzare un array. Gli esempi seguenti mostrano due array inizializzati tramite valori letterali array. Potete utilizzare l'istruzione `new` e passare il valore letterale composto come parametro alla funzione di costruzione della classe Array, ma potete anche assegnare valori letterali direttamente durante la creazione di istanze delle seguenti classi principali ActionScript: Object, Array, String, Number, int, uint, XML, XMLList e Boolean.

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

I valori letterali possono essere anche utilizzati per inizializzare un oggetto generico, ovvero un'istanza della classe Object. I valori letterali oggetto sono racchiusi tra parentesi graffe (`{}`) e utilizzano la virgola per separare le proprietà dell'oggetto. Ogni proprietà viene dichiarata con il carattere di due punti (`:`) che separa il nome della proprietà dal relativo valore.

È possibile creare un oggetto generico utilizzando l'istruzione `new` e passando il valore letterale dell'oggetto come parametro alla funzione di costruzione della classe Object, oppure assegnare il valore letterale dell'oggetto direttamente all'istanza che si dichiara. L'esempio seguente mostra due metodi alternativi per creare un nuovo oggetto generico e inizializzarlo con tre proprietà (`propA`, `propB` e `propC`), con valori impostati rispettivamente su 1, 2 e 3:

```
// Use new statement and add properties.
var myObject:Object = new Object();
myObject.propA = 1;
myObject.propB = 2;
myObject.propC = 3;

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

Altri argomenti presenti nell'Aiuto

Operazioni con le stringhe

Uso delle espressioni regolari

Inizializzazione delle variabili XML

Punto e virgola

Il carattere di punto e virgola (`;`) può essere utilizzato per terminare un'istruzione. Se omettete il carattere del punto e virgola, il compilatore presuppone che ogni riga di codice rappresenti una singola istruzione. Poiché molti programmatori sono abituati a usare il punto e virgola per indicare la fine di un'istruzione, il vostro codice può risultare più leggibile se adottate questa convezione in modo omogeneo.

L'uso di un punto e virgola per terminare un'istruzione consente di includere più di un'istruzione su una sola riga; tuttavia, un "affollamento" eccessivo di istruzioni può rendere più difficoltosa la lettura del codice.

Parentesi

In ActionScript 3.0 potete utilizzare le parentesi (`()`) in tre modi. Innanzi tutto, potete utilizzare le parentesi per modificare l'ordine delle operazioni in un'espressione. Le operazioni raggruppate all'interno di parentesi vengono sempre eseguite per prime. Ad esempio, nel codice seguente le parentesi sono state utilizzate per cambiare l'ordine delle operazioni:

```
trace(2 + 3 * 4); // 14
trace((2 + 3) * 4); // 20
```

In secondo luogo, potete utilizzare le parentesi con l'operatore virgola (`,`) per valutare una serie di espressioni e restituire il risultato dell'espressione finale, come nell'esempio seguente:

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

Infine, potete utilizzare le parentesi per passare uno o più parametri a funzioni o metodi, come nell'esempio seguente, dove viene passato un valore String alla funzione `trace()`:

```
trace("hello"); // hello
```

Commenti

Il codice ActionScript 3.0 supporta due tipi di commenti: a riga singola e su più righe. Questi meccanismi per l'inserimento di commenti sono simili a quelli di C++ e Java. Tutto il testo contrassegnato come commento viene ignorato dal compilatore.

I commenti a riga singola iniziano con due caratteri di barra (`//`) e continuano fino alla fine della riga. Ad esempio, il codice seguente contiene un commento a riga singola:

```
var someNumber:Number = 3; // a single line comment
```

I commenti su più righe invece iniziano con una barra seguita da un asterisco (`/*`) e finiscono con un asterisco seguito da una barra (`*/`).

```
/* This is multiline comment that can span
more than one line of code. */
```

Parole chiave e riservate

Per *parole riservate* si intendono parole che non possono essere utilizzate come identificatori nel codice perché sono riservate per l'uso in ActionScript. Le parole riservate comprendono anche le *parole chiave lessicali*, che vengono rimosse dallo spazio dei nomi del programma dal compilatore. Se utilizzate una parola chiave lessicale come identificatore, il compilatore genera un errore. Nella tabella seguente sono elencate tutte le parole chiave lessicali di ActionScript 3.0.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
con			

Esiste inoltre un piccolo gruppo di parole chiave, chiamate *parole chiave sintattiche*, che possono essere utilizzate come identificatori ma hanno un significato speciale in determinati contesti. Nella tabella seguente sono elencate tutte le parole chiave sintattiche di ActionScript 3.0.

each	get	set	namespace
include	dynamic	final	native
override	static		

Infine, vi sono vari identificatori che talvolta vengono definiti *parole riservate future*. Questi identificatori non sono riservati da ActionScript 3.0, sebbene alcuni possano essere trattati come parole chiave da software che include ActionScript 3.0. Potete utilizzare molti di questi identificatori nel codice, tuttavia Adobe ne sconsiglia l'impiego poiché potrebbero apparire come parole chiave in versioni successive del linguaggio.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

Costanti

ActionScript 3.0 supporta l'istruzione `const`, che permette di creare una costante. Le costanti sono proprietà il cui valore è fisso, ovvero non modificabile. Potete assegnare un valore a una costante una sola volta, e l'assegnazione deve avvenire in prossimità della dichiarazione della costante. Ad esempio, se una costante viene dichiarata come membro di una classe, è possibile assegnare ad essa un valore solo nella dichiarazione stessa oppure nella funzione di costruzione della classe.

Il codice seguente dichiara due costanti. Alla prima, `MINIMUM`, viene assegnato un valore nell'istruzione della dichiarazione, mentre il valore della seconda costante, `MAXIMUM`, viene assegnato nella funzione di costruzione. Notate che in questo esempio la compilazione viene eseguita in modalità standard poiché la modalità rigorosa consente di assegnare il valore di una costante solo in fase di inizializzazione.

```
class A
{
    public const MINIMUM:int = 0;
    public const MAXIMUM:int;

    public function A()
    {
        MAXIMUM = 10;
    }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

Se tentate di assegnare un valore iniziale a una costante in qualunque altro modo, viene generato un errore. Ad esempio, se impostate il valore iniziale di `MAXIMUM` all'esterno della classe, si verifica un errore runtime.

```
class A
{
public const MINIMUM:int = 0;
public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

ActionScript 3.0 definisce un'ampia gamma di costanti che potete utilizzare. Per convenzione, le costanti in ActionScript contengono solo lettere maiuscole, con le parole separate dal carattere di sottolineatura (_). Ad esempio, la definizione della classe MouseEvent utilizza questa convenzione di denominazione per le proprie costanti, ciascuna delle quali rappresenta un evento relativo all'input del mouse:

```
package flash.events
{
    public class MouseEvent extends Event
    {
    public static const CLICK:String = "click";
    public static const DOUBLE_CLICK:String = "doubleClick";
    public static const MOUSE_DOWN:String = "mouseDown";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...
    }
}
```

Operatori

Gli operatori sono funzioni speciali che accettano uno o più operandi e restituiscono un valore. Un *operando* è un valore (solitamente un valore letterale, una variabile o un'espressione) che svolge la funzione di input per un operatore. Ad esempio, nel seguente codice gli operatori di addizione (`+`) e moltiplicazione (`*`) sono utilizzati con tre operandi letterali (2, 3 e 4) per restituire un valore, il quale viene quindi utilizzato dall'operatore di assegnazione `=` () per assegnare il valore restituito, 14, alla variabile `sumNumber`.

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

Gli operatori possono essere unari, binari o ternari. Un operatore *unario* accetta un solo operando. Ad esempio, l'operatore di incremento (`++`) è di tipo unario perché accetta un unico operando. Un operatore *binario* accetta due operandi. È il caso, ad esempio, dell'operatore di divisione (`/`), che accetta due operandi. Un operatore *ternario* accetta tre operandi. Ad esempio, l'operatore condizionale (`?:`) accetta tre operandi.

Alcuni operatori, definiti *overloaded*, si comportano in modo diverso a seconda del tipo o del numero di operandi che vengono specificati. L'operatore di addizione (`+`) è un esempio di operatore overloaded che presenta un comportamento diverso a seconda del tipo di dati degli operandi. Se entrambi gli operandi sono numeri, l'operatore di addizione restituisce la somma dei valori. Se invece entrambi gli operandi sono stringhe, l'operatore restituisce la concatenazione dei due operandi. L'esempio di codice seguente mostra come l'operatore si comporta in modo diverso a seconda degli operandi specificati.

```
trace(5 + 5); // 10
trace("5" + "5"); // 55
```

Gli operatori possono presentare un comportamento diverso anche in base al numero di operandi specificati. L'operatore di sottrazione (`-`), ad esempio, è sia di tipo unario che binario. Se viene fornito un unico operando, l'operatore di sottrazione ne esegue la negazione e restituisce il risultato. Se gli operandi sono due, l'operatore ne restituisce la differenza. L'esempio seguente mostra l'operatore di sottrazione utilizzato prima nella versione unaria e poi in quella binaria.

```
trace(-3); // -3
trace(7 - 2); // 5
```

Priorità e associatività degli operatori

La priorità e l'associatività degli operatori determina l'ordine in cui questi vengono elaborati. Se avete familiarità con le operazioni matematiche, può sembrarvi ovvio che il compilatore elabori l'operatore di moltiplicazione (`*`) prima di quello di addizione (`+`). Tuttavia, il compilatore necessita di istruzioni esplicite sull'ordine di elaborazione degli operatori. Queste istruzioni vengono dette globalmente *priorità degli operatori*. ActionScript definisce un ordine di priorità predefinito per gli operatori che è possibile modificare tramite l'operatore parentesi tonda (`()`). Ad esempio, il codice seguente modifica l'ordine di priorità predefinito dell'esempio precedente per forzare il compilatore a elaborare l'operatore di addizione prima di quello di moltiplicazione:

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

In alcune situazioni, nella stessa espressione possono essere presenti due o più operatori con la stessa priorità. In questi casi, il compilatore utilizza le regole di *associatività* per determinare l'operatore da elaborare per primo. Tutti gli operatori binari, ad eccezione degli operatori di assegnazione, hanno un'associatività *da sinistra a destra*, ovvero gli operatori a sinistra vengono elaborati prima di quelli a destra. Gli operatori binari di assegnazione e l'operatore condizionale (`?:`) hanno un'associatività *da destra a sinistra*, ovvero gli operatori a destra vengono elaborati prima di quelli a sinistra.

Considerate, ad esempio, gli operatori minore di (<) e maggiore di (>) che hanno la stessa priorità. Se entrambi gli operatori vengono utilizzati nella stessa espressione, l'operatore a sinistra viene elaborato per primo, perché l'associatività di entrambi è da sinistra a destra. Le due istruzioni seguenti producono pertanto lo stesso risultato:

```
trace(3 > 2 < 1); // false
trace((3 > 2) < 1); // false
```

L'operatore maggiore di viene elaborato per primo e restituisce `true`, perché l'operando 3 è maggiore dell'operando 2. Il valore `true` viene quindi passato all'operatore minore di insieme all'operando 1. Il codice seguente illustra questo stato intermedio:

```
trace((true) < 1);
```

L'operatore minore di converte il valore `true` nel valore numerico 1 e confronta quest'ultimo con il secondo operando 1, restituendo il valore `false` (perché il valore di 1 non è minore di 1).

```
trace(1 < 1); // false
```

L'associatività a sinistra predefinita può essere modificata mediante l'operatore parentesi. Potete istruire il compilatore a elaborare per primo l'operatore minore di racchiudendolo tra parentesi insieme ai relativi operandi. L'esempio seguente utilizza l'operatore parentesi per produrre un output diverso con gli stessi numeri dell'esempio precedente:

```
trace(3 > (2 < 1)); // true
```

L'operatore minore di viene elaborato per primo e restituisce il valore `false` perché l'operando 2 non è minore dell'operando 1. Il valore `false` viene quindi passato all'operatore maggiore di insieme all'operando 3. Il codice seguente illustra questo stato intermedio:

```
trace(3 > (false));
```

L'operatore maggiore di converte il valore `false` nel valore numerico 0 e confronta quest'ultimo con l'altro operando 3, restituendo il valore `true` (perché il valore di 3 è maggiore di 0).

```
trace(3 > 0); // true
```

La tabella seguente elenca gli operatori di ActionScript 3.0 in ordine di priorità decrescente. Ogni riga della tabella contiene operatori che hanno la stessa priorità. Ciascuna riga di operatori ha una priorità più alta della riga sottostante.

Gruppo	**Operatori**
Primari	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Forma suffissa	`x++ x--`
Unari	`++x --x + - ~ ! delete typeof void`
Moltiplicativi	`* / %`
Additivi	`+ -`
Spostamento bit a bit	`<< >> >>>`
Relazionali	`< > <= >= as in instanceof is`
Uguaglianza	`== != === !==`
AND bit a bit	`&`
XOR bit a bit	`^`
OR bit a bit	`|`
AND logico	`&&`

Gruppo	Operatori
OR logico	`||`
Condizionale	`?:`
Assegnazione	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Virgola	`,`

Operatori primari

Gli operatori primari consentono di creare valori letterali Array e Object, raggruppare espressioni, chiamare funzioni, creare istanze di classe e accedere alle proprietà.

Tutti gli operatori primari, elencati nella tabella seguente, hanno priorità equivalente. Gli operatori inclusi nella specifica E4X sono contrassegnati con l'indicazione (E4X).

Operatore	Operazione eseguita
`[]`	Inizializza un array
`{x:y}`	Inizializza un oggetto
`()`	Raggruppa espressioni
`f(x)`	Chiama una funzione
`new`	Chiama una funzione di costruzione
`x.y x[y]`	Accede a una proprietà
`<></>`	Inizializza un oggetto XMLList (E4X)
`@`	Accede a un attributo (E4X)
`::`	Qualifica un nome (E4X)
`..`	Accede a un elemento XML discendente (E4X)

Operatori in forma suffissa

Gli operatori in forma suffissa accettano un operatore e ne incrementano o decrementano il valore. Sebbene si tratti di operatori unari, sono classificati separatamente dal resto degli operatori di questo tipo perché hanno una priorità maggiore e si comportano in modo particolare. Quando utilizzate un operatore in forma suffissa all'interno di un'espressione complessa, il valore dell'espressione viene restituito prima dell'elaborazione dell'operatore in forma suffissa. Nel codice seguente, ad esempio, il valore dell'espressione `xNum++` viene restituito prima che il valore venga incrementato:

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum); // 1
```

Tutti gli operatori in forma suffissa, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`++`	Incremento (in forma suffissa)
`--`	Decremento (in forma suffissa)

Operatori unari

Gli operatori unari accettano un unico operando. Gli operatori di incremento (`++`) e decremento (`--`) di questo gruppo sono operatori *in forma prefissa*, ovvero vengono utilizzati prima dell'operando in un'espressione. Gli operatori in forma prefissa sono diversi da quelli in forma suffissa perché l'operazione di incremento o decremento viene completata prima della restituzione del valore di tutta l'espressione. Nel codice seguente, ad esempio, il valore dell'espressione `++xNum` viene restituito dopo l'incremento del valore.

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum); // 1
```

Tutti gli operatori unari, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`++`	Incremento (in forma prefissa)
`--`	Decremento (in forma prefissa)
`+`	+ unario
`-`	- unario (negazione)
`!`	NOT logico
`~`	NOT bit a bit
`delete`	Elimina una proprietà
`typeof`	Restituisce informazioni sul tipo
`void`	Restituisce un valore indefinito

Operatori moltiplicativi

Gli operatori moltiplicativi accettano due operandi ed eseguono moltiplicazioni, divisioni e moduli.

Tutti gli operatori moltiplicativi, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`*`	Moltiplicazione
`/`	Divisione
`%`	Modulo

Operatori additivi

Gli operatori additivi accettano due operandi ed eseguono addizioni o sottrazioni. Tutti gli operatori additivi, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`+`	Addizione
`-`	Sottrazione

Operatori di spostamento bit a bit

Gli operatori di spostamento bit a bit accettano due operandi e spostano i bit del primo operando in base a quanto specificato dal secondo operando. Tutti gli operatori di spostamento bit a bit, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`<<`	Spostamento a sinistra bit a bit
`>>`	Spostamento a destra bit a bit
`>>>`	Spostamento a destra bit a bit senza segno

Operatori relazionali

Gli operatori relazionali accettano due operandi, ne confrontano il valore e restituiscono un valore booleano. Tutti gli operatori relazionali, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`<`	Minore di
`>`	Maggiore di
`<=`	Minore o uguale a
`>=`	Maggiore o uguale a
`as`	Verifica il tipo di dati
`in`	Verifica le proprietà dell'oggetto
`instanceof`	Controlla la catena di prototipi
`is`	Verifica il tipo di dati

Operatori di uguaglianza

Gli operatori di uguaglianza accettano due operandi, ne confrontano il valore e restituiscono un valore booleano. Tutti gli operatori di uguaglianza, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`==`	Uguaglianza
`!=`	Diseguaglianza
`===`	Uguaglianza rigorosa
`!==`	Diseguaglianza rigorosa

Operatori logici bit a bit

Gli operatori logici bit a bit accettano due operandi ed eseguono operazioni logiche a livello di bit. Gli operatori logici bit a bit hanno priorità diversa. Nella tabella seguente, gli operatori sono elencati in ordine di priorità, da maggiore a minore:

Operatore	Operazione eseguita
`&`	AND bit a bit
`^`	XOR bit a bit
`|`	OR bit a bit

Operatori logici

Gli operatori logici accettano due operandi e restituiscono un valore booleano. Gli operatori logici hanno priorità diversa. Nella tabella seguente, gli operatori sono elencati in ordine di priorità, da maggiore a minore:

Operatore	Operazione eseguita
`&&`	AND logico
`||`	OR logico

Operatore condizionale

L'operatore condizionale è un operatore ternario, ovvero accetta tre operandi. Può essere utilizzato come metodo rapido per applicare l'istruzione condizionale `if..else`.

Operatore	Operazione eseguita
`?:`	Condizionale

Operatori di assegnazione

Gli operatori di assegnazione accettano due operandi e assegnano un valore a uno di essi in base al valore dell'altro operando. Tutti gli operatori di assegnazione, elencati nella tabella seguente, hanno priorità equivalente.

Operatore	Operazione eseguita
`=`	Assegnazione
`*=`	Assegnazione moltiplicazione
`/=`	Assegnazione divisione
`%=`	Assegnazione modulo
`+=`	Assegnazione addizione
`-=`	Assegnazione sottrazione
`<<=`	Assegnazione spostamento a sinistra bit a bit
`>>=`	Assegnazione spostamento a destra bit a bit
`>>>=`	Assegnazione spostamento a destra senza segno bit a bit
`&=`	Assegnazione AND bit a bit
`^=`	Assegnazione XOR bit a bit
`|=`	Assegnazione OR bit a bit

Istruzioni condizionali

ActionScript 3.0 offre tre istruzioni condizionali di base che possono essere utilizzate per controllare il flusso del programma.

if..else

L'istruzione condizionale `if..else` consente di provare una condizione e quindi eseguire un blocco di codice se la condizione è soddisfatta o un blocco di codice alternativo in caso contrario. Il codice seguente, ad esempio, verifica se il valore di `x` è maggiore di 20 e genera un'istruzione `trace()` in caso affermativo oppure un'istruzione `trace()` diversa in caso negativo.

```
if (x > 20)
{
    trace("x is > 20");
}
else
{
    trace("x is <= 20");
}
```

Se non si desidera eseguire un blocco di codice alternativo, è possibile utilizzare l'istruzione `if` senza l'istruzione `else`.

if..else if

L'istruzione condizionale `if..else if` permette di provare più di una condizione. Il codice seguente, ad esempio, non solo controlla se il valore di `x` è maggiore di 20, ma anche se è negativo:

```
if (x > 20)
{
    trace("x is > 20");
}
else if (x < 0)
{
    trace("x is negative");
}
```

Se un'istruzione `if` o `else` è seguita da un'unica istruzione, non è necessario racchiuderla tra parentesi graffe. Il codice seguente, ad esempio, non utilizza le parentesi graffe:

```
if (x > 0)
    trace("x is positive");
else if (x < 0)
    trace("x is negative");
else
    trace("x is 0");
```

Adobe tuttavia consiglia di includere sempre le parentesi graffe per evitare risultati imprevisti nel caso che in un secondo momento vengano aggiunte altre istruzioni a un'istruzione condizionale priva di parentesi graffe. Ad esempio, nel codice seguente il valore di `positiveNums` viene incrementato di 1 indipendentemente dal fatto che la condizione restituisce o meno il valore `true`:

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
    trace("x is positive");
    positiveNums++;

trace(positiveNums); // 1
```

switch

L'istruzione `switch` è utile quando sono presenti vari percorsi di esecuzione che dipendono dalla stessa espressione condizionale. Fornisce funzionalità simili a una lunga serie di istruzioni `if..else if`, ma è più semplice da leggere. Anziché verificare se una condizione ha un valore booleano, l'istruzione `switch` valuta un'espressione e usa il risultato per determinare quale blocco di codice deve essere eseguito. I blocchi di codice iniziano con un'istruzione `case` e terminano con un'istruzione `break`. Ad esempio, l'istruzione `switch` seguente genera il giorno della settimana in base al numero del giorno restituito dal metodo `Date.getDay()`:

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

Cicli

Le istruzioni di ripetizione ciclica permettono di eseguire ripetutamente un blocco di codice specifico utilizzando una serie di valori o di variabili. Adobe consiglia di racchiudere sempre il blocco di codice tra parentesi graffe ({}). Sebbene sia possibile omettere le parentesi se il blocco di codice contiene un'unica istruzione, questa abitudine non è consigliata per lo stesso motivo esposto per le istruzioni condizionali: incrementa la possibilità che istruzioni aggiunte in seguito vengano inavvertitamente escluse dal blocco di codice. Se, in un secondo momento, aggiungete un'istruzione da includere nel blocco di codice ma omettete di aggiungere anche le necessarie parentesi graffe, l'istruzione viene ignorata al momento dell'esecuzione del ciclo.

for

Il ciclo `for` permette di eseguire iterazioni su una variabile per verificare un intervallo di valori specifico. In un'istruzione `for` è necessario fornire tre espressioni: una variabile impostata su un valore iniziale, un'istruzione condizionale che determina quando il ciclo termina e un'espressione che cambia il valore della variabile a ogni ciclo. Il codice seguente, ad esempio, esegue cinque iterazioni. Il valore della variabile `i` inizia a 0 e termina a 4 e l'output è rappresentato dai numeri da 0 a 4, ciascuno su una riga separata.

```
var i:int;
for (i = 0; i < 5; i++)
{
    trace(i);
}
```

for..in

Il ciclo `for..in` esegue un'iterazione sulle proprietà di un oggetto o sugli elementi di un array. Potete, ad esempio, ricorrere a un ciclo `for..in` per eseguire iterazioni sulle proprietà di un oggetto generico (le proprietà degli oggetti non vengono ordinate in base a criteri particolari, ma inserite in ordine casuale):

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
    trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

È possibile anche eseguire iterazioni sugli elementi di un array:

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
    trace(myArray[i]);
}
// output:
// one
// two
// three
```

Non è invece possibile eseguire iterazioni sulle proprietà di un oggetto che è un'istanza di una classe chiusa (comprese le classi incorporate e le classi definite dall'utente), a meno che la classe non sia dinamica. Anche in quest'ultimo caso, comunque, è possibile eseguire iterazioni solo sulle proprietà aggiunte in modo dinamico.

for each..in

Il ciclo `for each..` esegue un'iterazione sulle voci di una raccolta, che possono essere tag contenuti in un oggetto XML o XMLList, i valori delle proprietà di un oggetto o gli elementi di un array. Come mostrato nel seguente estratto di codice, potete, ad esempio, utilizzare un ciclo `for each..in` per eseguire iterazioni sulle proprietà di un oggetto generico, ma a differenza del ciclo `for..in`, la variabile di iterazione in un ciclo `for each..in` contiene il valore della proprietà anziché il nome della stessa:

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
    trace(num);
}
// output:
// 20
// 30
```

È possibile eseguire l'iterazione su un oggetto XML o XMLList, come mostra l'esempio seguente:

```
var myXML:XML = <users>
    <fname>Jane</fname>
    <fname>Susan</fname>
    <fname>John</fname>
</users>;

for each (var item in myXML.fname)
{
    trace(item);
}
/* output
Jane
Susan
John
*/
```

Potete anche eseguire iterazioni sugli elementi di un array, come nel codice seguente:

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
    trace(item);
}
// output:
// one
// two
// three
```

Non è invece possibile eseguire iterazioni sulle proprietà degli oggetti che sono istanze di classi chiuse. Nel caso di classi dinamiche, non è possibile eseguire l'iterazione sulle proprietà fisse, ovvero le proprietà definite nella definizione della classe.

while

Il ciclo `while` è come un'istruzione `if` che viene ripetuta fintanto che la condizione è `true`. Ad esempio, il codice seguente produce lo stesso output dell'esempio di ciclo `for`:

```
var i:int = 0;
while (i < 5)
{
    trace(i);
    i++;
}
```

Il ciclo `while` presenta uno svantaggio rispetto al ciclo `for`: è più facile scrivere cicli infiniti. A differenza del ciclo `for`, l'esempio di codice del ciclo `while` viene compilato anche se si omette l'espressione che incrementa la variabile del contatore. Senza l'espressione che incrementa `i`, il ciclo diventa infinito.

do..while

Il ciclo `do..` è un ciclo `while` che garantisce che il blocco di codice venga eseguito almeno una volta, perché la condizione viene verificata dopo l'esecuzione del blocco di codice. Il codice seguente mostra un semplice esempio di un ciclo `do..while` che genera un risultato anche se la condizione non è soddisfatta:

```
var i:int = 5;
do
{
    trace(i);
    i++;
} while (i < 5);
// output: 5
```

Funzioni

Le *funzioni* sono blocchi di codice che eseguono operazioni specifiche e possono essere riutilizzati all'interno del programma. In ActionScript 3.0 sono disponibili due tipi di funzioni: *metodi* e *chiusure di funzione*. Una funzione viene chiamata metodo oppure chiusura di funzione a seconda del contesto nel quale è definita. Se la funzione viene definita all'interno di una definizione di classe o associata a un'istanza di un oggetto, prende il nome di metodo. Se viene definita in qualunque altro modo, viene chiamata chiusura di funzione.

Le funzioni hanno sempre avuto un ruolo estremamente importante in ActionScript. In ActionScript 1.0, ad esempio, la parola chiave `class` non esisteva, quindi le "classi" erano definite dalle funzioni di costruzione. Anche se nel frattempo la parola chiave `class` è stata aggiunta al linguaggio, una conoscenza approfondita delle funzioni è ancora importante per sfruttare nel modo migliore le possibilità offerte da ActionScript. Questo compito può risultare più impegnativo per i programmatori che si aspettano un comportamento delle funzioni ActionScript analogo a quello di linguaggi come C++ o Java. Anche se le procedure di base per definire e chiamare le funzioni non dovrebbero rappresentare un problema per i programmatori esperti, alcune delle caratteristiche più avanzate di ActionScript richiedono un approfondimento.

Concetti di base delle funzioni

Chiamate di funzione

È possibile chiamare una funzione specificando il relativo identificatore seguito dall'operatore parentesi (`()`). L'operatore parentesi ha il compito di racchiudere gli eventuali parametri che si desidera inviare alla funzione. Ad esempio, la funzione `trace()` è una funzione di primo livello in ActionScript 3.0:

```
trace("Use trace to help debug your script");
```

Se chiamate una funzione senza parametri, dovete includere una coppia di parentesi vuote. Ad esempio, potete utilizzare il metodo `Math.random()`, che non accetta parametri, per generare un numero casuale:

```
var randomNum:Number = Math.random();
```

Definizione di funzioni personalizzate

In ActionScript 3.0 sono disponibili due metodi per definire una funzione: potete utilizzare un'istruzione di funzione o un'espressione di funzione. A seconda che preferiate uno stile di programmazione più statico o dinamico, potete scegliere una o l'altra tecnica. In genere, chi usa le istruzioni per definire le funzioni preferisce la programmazione statica (modalità rigorosa). Le espressioni vengono invece utilizzate per definire le funzioni quando esiste un'esigenza specifica in questo senso, solitamente nella programmazione dinamica (modalità standard).

Istruzioni di funzione

Le istruzioni di funzione sono la tecnica preferita per definire le funzioni in modalità rigorosa. Un'istruzione di funzione inizia con la parola chiave `function`, seguita da:

- Il nome della funzione
- I parametri, separati da virgole e racchiusi tra parentesi
- Il corpo della funzione, ovvero il codice ActionScript da eseguire quando la funzione viene chiamata, racchiuso tra parentesi graffe

Ad esempio, il codice seguente crea una funzione che definisce un parametro, quindi chiama la funzione utilizzando la stringa "`hello`" come valore del parametro:

```
function traceParameter(aParam:String)
{
    trace(aParam);
}

traceParameter("hello"); // hello
```

Espressioni di funzione

Il secondo modo per dichiarare una funzione prevede l'uso di un'istruzione di assegnazione con un'espressione di funzione, che talvolta viene anche definita letterale di funzione o funzione anonima. Si tratta di un metodo più verboso, largamente utilizzato nelle versioni precedenti di ActionScript.

Un'istruzione di assegnazione con un'espressione di funzione inizia con la parola chiave `var`, seguita da:

- Il nome della funzione
- L'operatore due punti (`:`)
- La classe `Function` per indicare il tipo di dati
- L'operatore di assegnazione (`=`)
- La parola chiave `function`
- I parametri, separati da virgole e racchiusi tra parentesi
- Il corpo della funzione, ovvero il codice ActionScript da eseguire quando la funzione viene chiamata, racchiuso tra parentesi graffe

  Ad esempio, il codice seguente dichiara la funzione `traceParameter` utilizzando un'espressione di funzione:

```
var traceParameter:Function = function (aParam:String)
{
    trace(aParam);
};
traceParameter("hello"); // hello
```

Notate che non occorre specificare un nome di funzione come avviene nelle istruzioni di funzione. Un'altra importante differenza tra le espressioni di funzione e le istruzioni di funzione consiste nel fatto che un'espressione di funzione è appunto un'espressione e non un'istruzione. Ciò significa che un'espressione di funzione, a differenza di un'istruzione di funzione, non può esistere come elemento autonomo, bensì può essere utilizzata solo all'interno di un'istruzione, solitamente un'istruzione di assegnazione. L'esempio seguente mostra un'espressione di funzione assegnata a un elemento array:

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
    trace(aParam);
};
traceArray[0]("hello");
```

Scelta tra istruzioni ed espressioni

Come regola generale, utilizzate un'istruzione di funzione a meno che le circostanze specifiche non suggeriscano l'uso di un'espressione. Le istruzioni di funzione sono meno verbose e, rispetto alle espressioni di funzione, producono risultati più omogenei tra modalità rigorosa e modalità standard.

Le istruzioni di funzione sono più facili da leggere delle istruzioni di assegnazione che contengono espressioni di funzione, consentono di scrivere codice più conciso e creano meno confusione delle espressioni di funzione, che richiedono l'uso delle due parole chiave `var` e `function`.

Inoltre, le istruzioni di funzione producono risultati più omogenei tra le due modalità del compilatore perché consentono di utilizzare la sintassi del punto sia in modalità rigorosa che standard per chiamare un metodo dichiarato mediante un'istruzione di funzione, il che non è sempre possibile per i metodi dichiarati con un'espressione di funzione. Il codice seguente, ad esempio, definisce una classe denominata Example con due metodi: `methodExpression()`, dichiarato con un'espressione di funzione, e `methodStatement()`, dichiarato con un'istruzione di funzione. In modalità rigorosa non potete utilizzare la sintassi del punto per chiamare il metodo `methodExpression()`.

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

Le espressioni di funzione sono generalmente più indicate per la programmazione che privilegia il comportamento runtime, ovvero dinamico. Se preferite utilizzare la modalità rigorosa ma avete anche la necessità di chiamare un metodo dichiarato con un'espressione di funzione, potete utilizzare entrambe le tecniche. Innanzi tutto, potete chiamare il metodo utilizzando le parentesi quadre (`[]`) invece dell'operatore punto (`.`). Il metodo seguente ha esito positivo sia in modalità rigorosa che standard:

```
myExample["methodLiteral"]();
```

In secondo luogo, potete dichiarare l'intera classe come classe dinamica. Sebbene in questo modo sia possibile chiamare il metodo utilizzando l'operatore punto, la funzionalità della modalità rigorosa viene parzialmente sacrificata per tutte le istanze di tale classe. Ad esempio, il compilatore non genera un errore se si tenta di accedere a una proprietà non definita su un'istanza di una classe dinamica.

In determinate circostanze le espressioni di funzione risultano utili. Un caso frequente è quello delle funzioni che vengono utilizzate una sola volta e quindi eliminate. Un altro utilizzo, meno comune, riguarda l'associazione di una funzione a una proprietà prototype. Per ulteriori informazioni, vedete Oggetto prototype.

Esistono due sottili differenze tra le istruzioni di funzione e le espressioni di funzione di cui va tenuto conto quando si sceglie la tecnica da utilizzare. La prima è che un'espressione di funzione non viene considerata come oggetto indipendente ai fini della gestione della memoria e del processo di garbage collection. In altre parole, quando si assegna un'espressione di funzione a un altro oggetto, ad esempio a un elemento di array o una proprietà di un oggetto, nel codice viene creato semplicemente un riferimento all'espressione di funzione. Se l'array o l'oggetto al quale è associata l'espressione di funzione esce dall'area di validità o comunque cessa di essere disponibile, non è più possibile accedere all'espressione. Se l'array o l'oggetto viene eliminato, la memoria utilizzata dall'espressione di funzione diventa disponibile per il processo di garbage collection, ovvero può essere riutilizzata per altri scopi.

L'esempio seguente mostra che, se viene eliminata la proprietà alla quale è assegnata un'espressione di funzione, la funzione non è più disponibile. La classe Test è dinamica e consente quindi di aggiungere una proprietà denominata `functionExp` che contiene un'espressione di funzione. La funzione `functionExp()` può essere chiamata con l'operatore punto, ma non è più accessibile dopo l'eliminazione della proprietà `functionExp`.

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp(); // Function expression
delete myTest.functionExp;
myTest.functionExp(); // error
```

Se, al contrario, la funzione viene inizialmente definita mediante un'istruzione di funzione, esiste come oggetto autonomo e continua a esistere anche dopo l'eliminazione della proprietà alla quale è associata. L'operatore `delete` funziona solo sulle proprietà degli oggetti, quindi non ha effetto se viene utilizzato per eliminare la funzione `stateFunc()`.

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc; // no effect
stateFunc();// Function statement
myTest.statement(); // error
```

La seconda differenza tra un'istruzione di funzione e un'espressione di funzione consiste nel fatto che la prima esiste in tutta l'area di validità nella quale è definita, comprese le istruzioni che la precedono. Un'espressione di funzione, al contrario, viene definita solo per le istruzioni successive. Ad esempio, il codice seguente contiene una chiamata (che ha esito positivo) alla funzione `scopeTest()` prima che venga definita:

```
statementTest(); // statementTest

function statementTest():void
{
    trace("statementTest");
}
```

Le espressioni di funzione non sono disponibili prima della posizione in cui vengono definite, quindi il codice seguente genera un errore runtime:

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
    trace("expressionTest");
}
```

Restituzione di valori da funzioni

Per restituire un valore da una funzione, utilizzate l'istruzione `return` seguita dall'espressione o dal valore letterale da restituire. Il seguente codice, ad esempio, restituisce un'espressione corrispondente al parametro:

```
function doubleNum(baseNum:int):int
{
    return (baseNum * 2);
}
```

Notate che l'istruzione `return` termina la funzione, quindi eventuali istruzioni successive a un'istruzione `return` non vengono eseguite, come nell'esempio seguente:

```
function doubleNum(baseNum:int):int {
    return (baseNum * 2);
    trace("after return"); // This trace statement will not be executed.
}
```

In modalità rigorosa dovete restituire un valore del tipo appropriato se scegliete di specificare un tipo restituito. Ad esempio, il codice seguente genera un errore in modalità rigorosa perché non restituisce un valore valido:

```
function doubleNum(baseNum:int):int
{
    trace("after return");
}
```

Funzioni nidificate

È possibile nidificare le funzioni, cioè dichiararle all'interno di altre funzioni. Una funzione nidificata è disponibile solo all'interno della funzione principale in cui è contenuta, a meno che non venga passato al codice esterno un riferimento alla funzione. Ad esempio, il codice seguente dichiara due funzioni nidificate all'interno della funzione `getNameAndVersion()`:

```
function getNameAndVersion():String
{
    function getVersion():String
    {
        return "10";
    }
    function getProductName():String
    {
        return "Flash Player";
    }
    return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 10
```

Se passate al codice esterno, le funzioni nidificate vengono passate come chiusure di funzione, vale a dire che la funzione conserva le definizioni che si trovano nell'area di validità nel momento in cui viene definita. Per ulteriori informazioni, vedete Area di validità delle funzioni.

Parametri di funzione

ActionScript 3.0 introduce alcune funzionalità relative ai parametri di funzione che potrebbero sembrare inedite ai programmatori che iniziano a utilizzare questo linguaggio. Benché l'idea di passare i parametri mediante un valore o un riferimento risulti probabilmente familiare alla maggior parte dei programmatori, l'oggetto `arguments` e il parametro ... (rest) potrebbero invece rappresentare una novità.

Passaggio di argomenti mediante un valore o un riferimento

In molti linguaggi di programmazione, è importante comprendere la distinzione che esiste tra passare gli argomenti mediante un valore oppure mediante un riferimento, poiché tale distinzione può influire su come viene progettato il codice.

Passare un argomento mediante un valore significa copiarne il valore in una variabile locale da utilizzare con la funzione. Al contrario, se si specifica un argomento mediante un riferimento, viene passato solo un riferimento all'argomento e non il valore effettivo. Nel secondo caso non viene quindi creata una copia dell'argomento vero e proprio, bensì viene passato come argomento un riferimento alla variabile nel momento in cui l'argomento viene creato e assegnato a una variabile locale da utilizzare nella funzione. Poiché è un riferimento a una variabile esterna alla funzione, la variabile locale consente di modificare il valore della variabile originale.

In ActionScript 3.0, tutti gli argomenti vengono passati mediante riferimento perché tutti i valori sono memorizzati come oggetti. Tuttavia, gli oggetti che appartengono ai tipi di dati di base (Boolean, Number, int, uint e String) prevedono l'uso di operatori speciali grazie ai quali possono comportarsi come se venissero passati mediante un valore. Ad esempio, il codice seguente crea una funzione denominata `passPrimitives()` che definisce due parametri chiamati `xParam` e `yParam`, entrambi del tipo int. Questi parametri sono simili a variabili locali dichiarate nel corpo della funzione `passPrimitives()`. Quando la funzione viene chiamata con gli argomenti `xValue` e `yValue`, i parametri `xParam` e `yParam` vengono inizializzati mediante riferimenti agli oggetti int rappresentati da `xValue` e `yValue`. Poiché gli argomenti appartengono a un tipo di base, si comportano come se fossero passati mediante valore. Benché `xParam` e `yParam` contengano inizialmente solo riferimenti agli oggetti `xValue` e `yValue`, qualunque modifica delle variabili all'interno del corpo della funzione genera nuove copie dei valori in memoria.

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

All'interno della funzione `passPrimitives()`, i valori di `xParam` e `yParam` vengono incrementati, tuttavia ciò non influisce sui valori di `xValue` e `yValue`, come mostra l'ultima istruzione `trace`. Lo stesso varrebbe anche nel caso in cui i parametri avessero gli stessi nomi delle variabili, `xValue` e `yValue`, perché i valori `xValue` e `yValue` all'interno della funzione farebbero riferimento a nuove posizioni di memoria, distinte dalle variabili omonime esterne alla funzione.

Tutti gli altri oggetti, ovvero gli oggetti che non appartengono ai tipi di dati primitivi, vengono sempre passati mediante riferimento, quindi con la possibilità di modificare il valore della variabile originale. Ad esempio, il codice seguente crea un oggetto denominato `objVar` con due proprietà, `x` e `y`. L'oggetto viene passato come argomento alla funzione `passByRef()`. Poiché non appartiene a un tipo di base, l'oggetto non viene semplicemente passato mediante un riferimento, ma rimane un riferimento. Ciò significa che le modifiche apportate ai parametri all'interno della funzione non hanno effetto sulle proprietà dell'oggetto all'esterno della funzione.

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

Il parametro `objParam` fa riferimento allo stesso oggetto della variabile globale `objVar`. Come potete notare nelle istruzioni `trace` dell'esempio, le modifiche apportate alle proprietà `x` e `y` dell'oggetto `objParam` vengono applicate anche all'oggetto `objVar`.

Valori predefiniti dei parametri

In ActionScript 3.0, potete dichiarare *valori di parametro predefiniti* per una funzione. Se in una chiamata a una funzione con parametri predefiniti viene omesso un parametro con valori predefiniti, viene utilizzato il valore specificato per quel parametro nella definizione della funzione. Tutti i parametri con valori predefiniti devono essere posizionati alla fine dell'elenco dei parametri. I valori assegnati come predefiniti devono essere costanti della fase di compilazione. L'esistenza di un valore predefinito per un parametro fa sì che quel parametro diventi un *parametro opzionale*, mentre un parametro privo di valore predefinito viene considerato un *parametro obbligatorio*.

Ad esempio, il codice seguente crea una funzione con tre parametri, due dei quali hanno valori predefiniti. Quando la funzione viene chiamata con un solo parametro, vengono utilizzati i valori predefiniti dei parametri.

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
    trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

L'oggetto arguments

Quando passate dei parametri a una funzione, potete utilizzare l'oggetto `arguments` per accedere alle informazioni relative a tali parametri. Seguono alcune osservazioni importanti relative all'oggetto `arguments`:

- l'oggetto `arguments` è un array che include tutti i parametri passati alla funzione;
- la proprietà `arguments.length` segnala il numero di parametri passati alla funzione;
- la proprietà `arguments.callee` fornisce un riferimento alla funzione stessa, che è utile per le chiamate ricorsive alle espressioni di funzione.

  Nota: *l'oggetto `arguments` non è disponibile se è presente un parametro denominato `arguments` oppure se utilizzate il parametro ... (rest).*

  Se il corpo della funzione contiene un riferimento all'oggetto `arguments`, ActionScript 3.0 consente di includere nelle chiamate di funzione più parametri di quelli definiti nella definizione della funzione; tuttavia, in modalità rigorosa genera un errore del compilatore se il numero di parametri non corrisponde al numero di parametri obbligatori (e facoltativamente, qualsiasi parametro opzionale). Potete ricorrere alla funzionalità di array dell'oggetto `arguments` per accedere a qualunque parametro passato alla funzione, a prescindere che sia o meno definito nella definizione della funzione. L'esempio seguente, che viene compilato solo in modalità rigorosa, utilizza l'array `arguments` con la proprietà `arguments.length` per tracciare tutti i parametri passati alla funzione `traceArgArray()`:

  ```
  function traceArgArray(x:int):void
  {
      for (var i:uint = 0; i < arguments.length; i++)
      {
          trace(arguments[i]);
      }
  }

  traceArgArray(1, 2, 3);

  // output:
  // 1
  // 2
  // 3
  ```

  La proprietà `arguments.callee` viene spesso utilizzata nelle funzioni anonime per creare la ricorsività e rendere il codice più flessibile. Se il nome di una funzione ricorsiva cambia durante il ciclo di sviluppo, non occorre modificare la chiamata ricorsiva nel corpo della funzione se si utilizza `arguments.callee` al posto del nome della funzione. Nell'espressione di funzione seguente, la proprietà `arguments.callee` viene utilizzata per abilitare la ricorsività:

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

Se utilizzate il parametro ... (rest) nella dichiarazione della funzione, l'oggetto `arguments` non è disponibile e per accedere ai parametri è necessario utilizzare i rispettivi nomi che sono stati dichiarati.

È inoltre importante evitare di utilizzare la stringa `"arguments"` come nome di parametro perché impedisce l'uso dell'oggetto `arguments`. Ad esempio, se la funzione `traceArgArray()` viene riscritta con l'aggiunta di un parametro `arguments`, i riferimenti a `arguments` nel corpo della funzione sono relativi al parametro anziché all'oggetto `arguments`. Il codice seguente non produce alcun output:

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// no output
```

Nelle versioni precedenti di ActionScript, l'oggetto `arguments` conteneva anche una proprietà denominata `caller`, che era un riferimento alla funzione che chiamava la funzione corrente. La proprietà `caller` non è presente in ActionScript 3.0, ma se occorre fare riferimento alla funzione chiamante, potete modificare quest'ultima in modo che passi un parametro supplementare contenente un riferimento a se stessa.

Il parametro ... (rest)

In ActionScript 3.0 è stata introdotta una nuova dichiarazione di parametro, il parametro ... (rest), che consente di specificare un parametro array che accetta qualunque numero di argomenti separati da virgole. Il parametro può avere qualsiasi nome che non corrisponda a una parola riservata e deve essere l'ultimo parametro specificato. L'uso di questo parametro rende indisponibile l'oggetto `arguments`. Anche se il parametro ... (rest) offre la stessa funzionalità dell'array `arguments` e della proprietà `arguments.length`, non fornisce invece una funzionalità simile a quella di `arguments.callee`. Prima di usare il parametro ... (rest), assicuratevi che non sia necessario utilizzare `arguments.callee`.

Il seguente esempio riscrive la funzione `traceArgArray()` utilizzando il parametro ... (rest) invece dell'oggetto `arguments`:

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

Il parametro ... (rest) può anche essere utilizzato con altri parametri, a condizione che venga specificato per ultimo. L'esempio seguente modifica la funzione `traceArgArray()` in modo tale che il primo parametro, `x`, sia del tipo int, e il secondo utilizzi il parametro ... (rest). L'output ignora il primo valore perché il primo parametro non fa più parte dell'array creato dal parametro ... (rest).

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

Funzioni come oggetti

In ActionScript 3.0, le funzioni sono oggetti. Quando create una funzione, ciò che viene creato è in realtà un oggetto che non solo può essere passato come parametro a un'altra funzione, ma che può disporre anche di proprietà e metodi.

Le funzioni specificate come argomenti per altre funzioni vengono passate mediante riferimento e non mediante un valore. Quando passate una funzione come argomento, utilizzate solo l'identificatore e omettete l'operatore parentesi usato per chiamare il metodo. Ad esempio, il codice seguente passa una funzione denominata `clickListener()` come argomento al metodo `addEventListener()`:

```
addEventListener(MouseEvent.CLICK, clickListener);
```

Per quanto possa sembrare strano ai programmatori che iniziano a utilizzare ActionScript, le funzioni possono avere proprietà e metodi come qualunque altro oggetto. In effetti, ogni funzione dispone di una proprietà di sola lettura denominata `length` che memorizza il numero di parametri definiti per la funzione. Questa proprietà è diversa dalla proprietà `arguments.length`, che indica il numero di argomenti passati alla funzione. È bene ricordare che in ActionScript il numero di argomenti inviati a una funzione può superare quello dei parametri definiti per la stessa funzione. L'esempio seguente, che viene compilato solo in modalità standard perché la modalità rigorosa richiede una corrispondenza esatta tra il numero di argomenti passati e il numero di parametri definiti, mostra la differenza tra le due proprietà:

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

In modalità rigorosa, potete definire proprietà personalizzate per la funzione all'esterno del corpo della funzione. Le proprietà di funzione possono servire come proprietà "quasi statiche" che consentono di salvare lo stato di una variabile relativo alla funzione. Ad esempio, potrebbe essere utile registrare quante volte viene utilizzata una particolare funzione. Questa funzionalità può servire se si sta creando un videogame e si desidera registrare quante volte un utente utilizza un comando specifico (benché sia anche possibile utilizzare una proprietà di classe statica per lo stesso scopo). Nell'esempio seguente, che viene compilato solo in modalità standard poiché la modalità rigorosa non consente di aggiungere proprietà dinamiche alle funzioni, viene creata una proprietà di funzione all'esterno della dichiarazione di funzione e incrementata la proprietà ogni volta che la funzione viene chiamata:

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

Area di validità delle funzioni

L'area di validità di una funzione determina non solo l'area in cui, all'interno di un programma, quella funzione può essere chiamata, ma anche le definizioni alle quali la funzione può accedere. Le stesse regole dell'area di validità che valgono per gli identificatori delle variabili si applicano anche agli identificatori di funzione. Una funzione dichiarata nell'area di validità globale è disponibile in tutto il codice. Ad esempio, ActionScript 3.0 contiene funzioni globali, quali `isNaN()` e `parseInt()`, che sono disponibili in qualunque punto del codice. Una funzione nidificata (cioè dichiarata all'interno di un'altra funzione) può essere utilizzata in qualunque posizione all'interno della funzione in cui è stata dichiarata.

La catena dell'area di validità

Ogni volta che inizia l'esecuzione di una funzione, viene creata una serie di oggetti e di proprietà. Innanzi tutto, viene creato un oggetto speciale chiamato *oggetto di attivazione*, nel quale vengono memorizzati i parametri e le eventuali variabili locali o funzioni dichiarate nel corpo della funzione. Non è possibile accedere direttamente all'oggetto di attivazione perché è un meccanismo interno. In secondo luogo viene creata una *catena dell'area di validità* che contiene un elenco ordinato degli oggetti nei quali il runtime cerca le dichiarazioni di identificazione (gli

identificatori). Ogni funzione che viene eseguita ha una catena dell'area di validità che viene memorizzata in una proprietà interna. Nel caso di una funzione nidificata, la catena dell'area di validità inizia con il proprio oggetto di attivazione, seguito dall'oggetto di attivazione della relativa funzione principale. La catena prosegue nello stesso modo fino al raggiungimento dell'oggetto globale, ovvero l'oggetto che viene creato all'inizio di un programma ActionScript e che contiene tutte le variabili globali e le funzioni.

Chiusure di funzione

Una *chiusura di funzione* è un oggetto che contiene un'istantanea della funzione e il relativo *ambiente lessicale*, il quale comprende tutte le variabili, le proprietà, i metodi e gli oggetti inclusi nella catena dell'area di validità della funzione, con i rispettivi valori. Le chiusure di funzione vengono create ogni volta che una funzione viene eseguita indipendentemente da un oggetto o da una classe. Il fatto che una chiusura di funzione mantenga l'area di validità nella quale è stata definita produce risultati interessanti quando una funzione viene passata in un'area di validità diversa come argomento o come valore restituito.

Il seguente codice, ad esempio, crea due funzioni: `foo()`, che restituisce una funzione nidificata di nome `rectArea()` che calcola l'area di un rettangolo, e `bar()`, che chiama `foo()` e memorizza la chiusura di funzione restituita in una variabile denominata `myProduct`. Anche se la funzione `bar()` definisce la propria variabile locale `x` (con valore 2), quando la chiusura di funzione `myProduct()` viene chiamata, essa mantiene la variabile `x` (con valore 40) definita nella funzione `foo()`. La funzione `bar()`, pertanto, restituisce il valore `160` anziché `8`.

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

I metodi si comportano in modo analogo, perché conservano a loro volta le informazioni relative all'ambiente lessicale nel quale sono stati creati. Questa caratteristica è evidente soprattutto quando un metodo viene estratto dalla propria istanza per creare un metodo vincolato. La differenza principale tra una chiusura di funzione e un metodo vincolato consiste nel fatto che il valore della parola chiave `this` in un metodo vincolato fa sempre riferimento all'istanza alla quale è stata originariamente associata, mentre in una chiusura di funzione il valore di `this` può cambiare.

Capitolo 4: Programmazione orientata agli oggetti con ActionScript

Introduzione alla programmazione orientata agli oggetti

La programmazione orientata agli oggetti (OOP, Object Oriented Programming) è un sistema di organizzazione del codice di un programma mediante raggruppamento all'interno di oggetti. In questo senso, il termine *oggetto* indica un singolo elemento che include informazioni (valori di dati) e funzionalità. L'uso di un approccio orientato agli oggetti per organizzare un programma consente di raggruppare particolari informazioni insieme a funzionalità o azioni comuni associate a tali informazioni. Ad esempio, potete raggruppare informazioni sui brani musicali, quali titolo dell'album, titolo della traccia o nome dell'artista insieme a funzionalità quali "aggiungi traccia all'elenco di riproduzione" o "riproduci tutti i brani di questo artista". Tali informazioni vengono combinate in un'unica voce, un oggetto (ad esempio, un oggetto "Album" o "MusicTrack"). Il raggruppamento di valori e funzioni offre diversi vantaggi. Un vantaggio chiave è la necessità di utilizzare una sola variabile, anziché più variabili, oltre alla possibilità di mantenere insieme funzionalità correlate. La combinazione di informazioni e funzionalità consente, infine, di strutturare i programmi in modo più vicino al mondo reale.

Classi

Una classe è una rappresentazione astratta di un oggetto. In una classe sono memorizzate informazioni sui tipi di dati che un oggetto può contenere e sui comportamenti in base ai quali un oggetto può funzionare. L'utilità di tale astrazione non è necessariamente evidente quando si creano brevi script contenenti solo pochi oggetti che interagiscono tra loro. Tuttavia, con l'ampliarsi dell'area di validità di un programma aumenta anche il numero di oggetti da gestire. In tal caso, le classi consentono di controllare più accuratamente la creazione degli oggetti e il modo in cui essi interagiscono tra loro.

Fino alla versione di ActionScript 1.0, i programmatori potevano utilizzare gli oggetti funzione per creare costrutti somiglianti a classi. Con ActionScript 2.0 è stato introdotto il supporto delle classi con parole chiave quali `class` ed `extends`. In ActionScript 3.0, non solo è previsto il supporto delle parole chiave introdotte in ActionScript 2.0, ma vengono introdotte anche nuove funzionalità. Ad esempio, ActionScript 3.0 include il controllo dell'accesso ottimizzato mediante gli attributi `protected` e `internal`. Inoltre, fornisce un migliore controllo dell'ereditarietà grazie all'uso delle parole chiave `final` e `override`.

Per gli sviluppatori che hanno già esperienza nella creazione di classi con linguaggi di programmazione quali Java, C++ o C#, le procedure di ActionScript risulteranno familiari. ActionScript utilizza infatti molte parole chiave e nomi di attributo comuni a tali linguaggi di programmazione, ad esempio `class`, `extends` e `public`.

Nota: *nella documentazione Adobe ActionScript, il termine proprietà si riferisce a qualsiasi membro di una classe o di un oggetto, incluse variabili, costanti e metodi. Inoltre, anche se spesso i termini classe e statica vengono utilizzati in modo intercambiabile, in questo capitolo essi si riferiscono a concetti distinti. Ad esempio, il termine "proprietà di classe" indica tutti i membri di una classe e non solo i membri statici.*

Definizioni delle classi

Le definizioni delle classi in ActionScript 3.0 impiegano una sintassi simile a quella utilizzata in ActionScript 2.0. La sintassi corretta per la definizione di una classe richiede la parola chiave `class` seguita dal nome della classe. Il corpo della classe, racchiuso tra parentesi graffe ({}), segue il nome della classe. Ad esempio, il seguente codice consente di creare una classe denominata Shape contenente una variabile denominata `visible`:

```
public class Shape
{
    var visible:Boolean = true;
}
```

Una modifica significativa della sintassi riguarda la definizione delle classi che si trovano all'interno di un pacchetto. In ActionScript 2.0, se una classe si trovava all'interno di un pacchetto, il nome del pacchetto doveva essere incluso nella dichiarazione della classe. In ActionScript 3.0, con l'introduzione dell'istruzione `package`, il nome del pacchetto deve essere incluso nella dichiarazione del pacchetto, anziché nella dichiarazione della classe. Ad esempio, le seguenti dichiarazioni di classe mostrano come la classe BitmapData, che fa parte del pacchetto flash.display, viene definita in ActionScript 2.0 e in ActionScript 3.0:

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

Attributi di classe

ActionScript 3.0 consente di modificare le definizioni delle classi mediante l'impiego di uno dei seguenti attributi:

Attributo	Definizione
`dynamic`	Consente di aggiungere proprietà alle istanze in fase di runtime.
`final`	La classe non può essere estesa da un'altra classe.
`internal` (valore predefinito)	Visibile ai riferimenti che si trovano all'interno del pacchetto corrente.
`public`	Visibile a tutti i riferimenti.

In tutti questi casi, a eccezione di `internal`, dovete includere espressamente l'attributo per ottenere il comportamento associato. Ad esempio, se la definizione di una classe non include l'attributo `dynamic`, non è possibile aggiungere proprietà a un'istanza della classe in fase di runtime. Per assegnare esplicitamente un attributo dovete collocarlo all'inizio della definizione della classe, come indicato nel codice seguente:

```
dynamic class Shape {}
```

Tenete presente che nell'elenco non è incluso un attributo denominato `abstract`. Le classi abstract non sono supportate in ActionScript 3.0. Notate inoltre che l'elenco non contiene attributi denominati `private` e `protected`. Tali attributi hanno un significato unicamente all'interno di una definizione di classe e non possono essere applicati alle classi stesse. Per fare in modo che una classe non sia pubblicamente visibile al di fuori di un pacchetto, è necessario inserire la classe in un pacchetto e contrassegnarla con l'attributo `internal`. In alternativa, potete omettere sia l'attributo `internal` che l'attributo `public`; in tal modo il compilatore aggiunge automaticamente l'attributo `internal`. Potete anche definire una classe in modo che sia visibile solo all'interno del file di origine in cui è definita. Collocate la classe alla base del file sorgente, sotto la parentesi graffa che chiude la definizione del pacchetto.

Corpo della classe

Il corpo della classe, racchiuso tra parentesi graffe, consente di definire le variabili, le costanti e i metodi della classe. L'esempio seguente mostra la dichiarazione della classe Accessibility in ActionScript 3.0:

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

All'interno del corpo di una classe è possibile definire anche uno spazio dei nomi. Nell'esempio seguente è illustrato come uno spazio dei nomi può essere definito all'interno del corpo di una classe e utilizzato come attributo di un metodo di tale classe:

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 consente di inserire nel corpo di una classe non solo definizioni, ma anche istruzioni. Le istruzioni che si trovano all'interno del corpo di una classe, ma al di fuori della definizione di un metodo, vengono eseguite una sola volta, vale a dire nel momento in cui la definizione della classe viene rilevata la prima volta e l'oggetto classe associato viene creato. L'esempio seguente include una chiamata a una funzione esterna, `hello()`, e un'istruzione `trace` che consente di produrre un messaggio di conferma quando la classe viene definita:

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

In ActionScript 3.0 è consentito definire una proprietà statica e una proprietà di istanza con lo stesso nome all'interno di un medesimo corpo di classe. Ad esempio, il codice seguente indica una variabile statica denominata `message` e una variabile di istanza con lo stesso nome:

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

Attributi delle proprietà di classe

In relazione al modello a oggetti di ActionScript, il termine *proprietà* si riferisce a un qualsiasi membro di un una classe, incluse variabili, costanti e metodi. Tuttavia, in Adobe ActionScript 3.0 Reference for the Adobe Flash Platform (Guida di riferimento di Adobe ActionScript 3.0 per la piattaforma Adobe Flash) il termine viene invece utilizzato in senso più ristretto. In tale contesto, il termine proprietà include unicamente membri di classi corrispondenti a variabili o definiti mediante il metodo getter o setter. In ActionScript 3.0 è disponibile una serie di attributi utilizzabili con qualsiasi proprietà di classe. La tabella seguente riporta tali attributi.

Attributo	Definizione
`internal` (valore predefinito)	Visibile ai riferimenti che si trovano all'interno dello stesso pacchetto.
`private`	Visibile ai riferimenti che si trovano all'interno della stessa classe.
`protected`	Visibile ai riferimenti che si trovano all'interno della stessa classe e delle classi derivate.
`public`	Visibile a tutti i riferimenti.
`static`	Specifica che una proprietà appartiene alla classe e non alle sue istanze.
`UserDefinedNamespace`	Nome dello spazio dei nomi personalizzato definito dall'utente.

Attributi dello spazio dei nomi per il controllo dell'accesso

ActionScript 3.0 prevede quattro attributi speciali che controllano l'accesso alle proprietà definite all'interno di una classe: `public`, `private`, `protected` e `internal`.

L'attributo `public` rende una proprietà visible in qualsiasi punto dello script. Ad esempio, per rendere un metodo disponibile al codice che si trova al di fuori del pacchetto, dovete dichiarare il metodo con l'attributo `public`. Ciò vale per qualsiasi proprietà che sia stata dichiarata utilizzando le parole chiave `var`, `const` o `function`.

L'attributo `private` rende una proprietà visibile unicamente ai chiamanti entro la classe che definisce le proprietà. Tale comportamento si differenzia da quello dell'attributo `private` di ActionScript 2.0, che consentiva a una sottoclasse di accedere a una proprietà privata di una superclasse. Un'altra significativa modifica nel comportamento ha a che fare con l'accesso in fase di runtime. In ActionScript 2.0 la parola chiave `private` impediva l'accesso solo in fase di compilazione e poteva essere facilmente evitata in fase di runtime. In ActionScript 3.0 ciò non è più possibile. Le proprietà contrassegnate come `private` non risultano disponibili né in fase di compilazione né in fase di runtime.

Ad esempio, il codice seguente crea una semplice classe denominata PrivateExample con una variabile private, quindi tenta di accedere alla variabile privata dall'esterno della classe.

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this
is a run-time error.
```

In ActionScript 3.0, il tentativo di accedere a una proprietà privata mediante l'operatore punto (`myExample.privVar`) provoca un errore di compilazione, se ci si trova in modalità rigorosa. In caso contrario, l'errore viene riportato in fase di runtime, esattamente come accade quando si usa l'operatore di accesso alle proprietà (`myExample["privVar"]`).

Nella tabella seguente sono riportati i risultati dei tentativi di accesso a una proprietà privata appartenente a una classe chiusa (non dinamica):

	Modalità rigorosa	**Modalità standard**
operatore punto (`.`)	errore di compilazione	errore di runtime
operatore parentesi (`[]`)	errore di runtime	errore di runtime

Nelle classi dichiarate con l'attributo `dynamic`, i tentativi di accesso a una variabile privata non provocano errori di runtime. La variabile non è invece visibile, per cui viene restituito il valore `undefined`. Un errore in fase di compilazione si verifica invece se si utilizza un operatore punto in modalità rigorosa. L'esempio seguente è uguale a quello precedente, con la sola differenza che la classe PrivateExample viene dichiarata come classe dinamica:

```
dynamic class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

Le classi dinamiche restituiscono in genere il valore `undefined`, anziché generare un errore, se un codice esterno alla classe tenta di accedere a una proprietà privata. Nella tabella seguente si evidenzia che l'errore viene generato unicamente se si utilizza l'operatore punto per accedere a una proprietà privata in modalità rigorosa:

	Modalità rigorosa	**Modalità standard**
operatore punto (`.`)	errore di compilazione	`undefined`
operatore parentesi (`[]`)	`undefined`	`undefined`

L'attributo `protected`, nuovo in ActionScript 3.0, rende una proprietà visibile ai chiamanti della sua stessa classe o di una sottoclasse. In altre parole, una proprietà protetta è disponibile solo all'interno della propria classe o delle classi che si trovano sotto di essa nella gerarchia di ereditarietà. Ciò vale sia che la sottoclasse si trovi nello stesso pacchetto o in un pacchetto differente.

Questa funzionalità è analoga all'attributo `private` in ActionScript 2.0. Anche l'attributo `protected` di ActionScript 3.0 è simile all'attributo Java `protected`, ma, a differenza della versione Java, consente l'accesso anche ai chiamanti all'interno dello stesso pacchetto. L'attributo `protected` è utile se si dispone di una variabile o di un metodo necessario alle sottoclassi, ma che si desidera tenere nascosto da codice esterno alla catena di ereditarietà.

L'attributo `internal`, nuovo in ActionScript 3.0, rende una proprietà visibile ai chiamanti che si trovano all'interno del suo stesso pacchetto. Si tratta dell'attributo predefinito del codice all'interno di un pacchetto e può essere applicato a qualsiasi proprietà che non presenta alcuno dei seguenti attributi:

- `public`
- `private`
- `protected`
- uno spazio dei nomi definito dall'utente

L'attributo `internal` è simile al controllo dell'accesso predefinito in Java, anche se in Java non esiste un nome esplicito per tale livello di accesso, che può essere ottenuto solo mediante l'omissione di un qualsiasi altro modificatore dell'accesso. L'attributo `internal` di ActionScript 3.0 consente all'utente di dimostrare esplicitamente l'intenzione di rendere una proprietà visibile unicamente ai chiamanti che si trovano all'interno del suo stesso pacchetto.

Attributo static

L'attributo `static`, utilizzabile con proprietà dichiarate con le parole chiave `var`, `const` o `function`, consente di associare una proprietà alla classe anziché a istanze della classe. Il codice esterno alla classe deve chiamare proprietà statiche utilizzando il nome della classe anziché il nome di un'istanza.

Le proprietà statiche non vengono ereditate dalle sottoclassi, ma fanno parte della catena delle aree di validità di una sottoclasse. In altre parole, all'interno del corpo di una sottoclasse, è possibile utilizzare una variabile o un metodo senza fare riferimento alla classe in cui sono stati creati.

Attributi spazio dei nomi definiti dall'utente

In alternativa agli attributi di controllo dell'accesso predefiniti, è possibile creare uno spazio dei nomi personalizzato da utilizzare come attributo. Per definizione è possibile utilizzare un solo attributo spazio dei nomi e non è possibile utilizzare tale attributo in combinazione con un qualsiasi attributo di controllo dell'accesso (`public`, `private`, `protected`, `internal`).

Variabili

Le variabili possono essere dichiarate con le parole chiave `var` o `const`. I valori delle variabili dichiarate con la parola chiave `var` possono essere modificati più volte durante l'esecuzione di uno script. Le variabili dichiarate con la parola chiave `const` sono dette *costanti* e ad esse possono essere assegnati valori una sola volta. Se tentate di assegnare un nuovo valore a una costante inizializzata, si verifica un errore.

Variabili statiche

Le variabili statiche vengono dichiarate utilizzando una combinazione della parola chiave `static` e dell'istruzione `var` o `const`. Le variabili statiche che vengono associate a una classe, anziché all'istanza di una classe, sono utili per memorizzare e condividere informazioni applicabili a un'intera classe di oggetti. Ad esempio, è appropriato l'impiego di una variabile statica per registrare quante volte viene creata un'istanza di una determinata classe oppure per memorizzare il numero massimo di istanze di una classe consentito.

Nell'esempio seguente viene creata una variabile `totalCount` per registrare il numero di volte in cui viene creata l'istanza di una classe e una costante `MAX_NUM` per memorizzare il numero massimo di istanze consentite. Le variabili `totalCount` e `MAX_NUM` sono statiche, in quanto contengono valori applicabili all'intera classe e non a una sua particolare istanza.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

Il codice esterno alla classe StaticVars e alle sue sottoclassi può fare riferimento alle proprietà `totalCount` e `MAX_NUM` solo attraverso la classe stessa. Ad esempio, il codice seguente funziona:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

Non è possibile accedere a variabili statiche attraverso un'istanza della classe; il codice seguente restituisce degli errori:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Le variabili dichiarate sia con la parola chiave `static` che `const` devono essere inizializzate nel momento in cui viene dichiarata la costante, come avviene all'interno della classe StaticVars per `MAX_NUM`. Non è possibile assegnare un valore a `MAX_NUM` all'interno della funzione di costruzione o di un metodo di istanza. Il codice seguente genera un errore in quanto non è un metodo valido per inizializzare una costante statica:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Variabili di istanza

Le variabili di istanza includono proprietà dichiarate con le parole chiave `var` e `const`, ma senza la parola chiave `static`. Le variabili di istanza associate a istanze di classe, anziché a un'intera classe, sono utili per la memorizzazione di valori specifici di una particolare istanza. Ad esempio, la classe Array presenta una proprietà di istanza denominata `length` nella quale è memorizzato il numero di elementi di array contenuti in una particolare istanza della classe Array.

Le variabili di istanza, dichiarate sia con la parola chiave `var` che `const`, non possono essere sovrascritte all'interno di una sottoclasse. È tuttavia possibile ottenere una funzionalità analoga sostituendo i metodi getter e setter.

Metodi

I metodi sono funzioni che fanno parte di una definizione di classe. Una volta creata un'istanza della classe, un metodo viene associato a tale istanza. A differenza delle funzioni dichiarate all'esterno delle classi, i metodi possono essere utilizzati esclusivamente dall'istanza alla quale sono associati.

I metodi vengono definiti utilizzando la parola chiave `function`. Come avviene con qualsiasi proprietà delle classi, potete applicare uno degli attributi delle proprietà della classe ai metodi, inclusi spazi dei nomi privati, protetti, pubblici, interni, statici o personalizzati. Potete utilizzare un'istruzione di funzione quale la seguente:

```
public function sampleFunction():String {}
```

Oppure potete utilizzare una variabile alla quale viene assegnata un'espressione della funzione, come nell'esempio seguente:

```
public var sampleFunction:Function = function () {}
```

Nella maggior parte dei casi, si consiglia di utilizzare una funzione con istruzione anziché un'espressione della funzione per le ragioni seguenti:

- Le funzioni con istruzione sono più concise e più facili da leggere.
- Le funzioni con istruzione consentono di utilizzare le parole chiave `override` e `final`.
- Le istruzioni della funzione consentono di creare un legame più solido tra l'identificatore (il nome della funzione) e il codice, all'interno del corpo del metodo. Poiché il valore di una variabile può essere modificato mediante un'istruzione di assegnazione, il legame tra la variabile e la relativa espressione della funzione può essere sciolto in qualsiasi momento. Nonostante sia possibile ovviare a questo problema dichiarando la variabile con la parola chiave `const` anziché `var`, tale sistema non è consigliato, in quanto rende il codice di difficile lettura e impedisce l'uso delle parole chiave `override` e `final`.

Un caso in cui è consigliabile l'uso di un'espressione della funzione è quando si decide di associare una funzione all'oggetto prototype.

Metodi delle funzioni di costruzione

I metodi delle funzioni di costruzione, talvolta dette *funzioni di costruzione*, sono funzioni che condividono lo stesso nome della classe nella quale vengono definite. Qualsiasi codice incluso in un metodo di funzione di costruzione viene eseguito ogni volta che viene creata un'istanza della classe con la parola chiave `new`. Ad esempio, il codice seguente definisce una semplice classe chiamata Example, contenente un'unica proprietà denominata `status`. Il valore iniziale della variabile `status` viene definito all'interno della funzione di costruzione.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

I metodi delle funzioni di costruzione possono essere solo pubblici, tuttavia, l'impiego dell'attributo `public` è facoltativo. Nelle funzioni di costruzione non è possibile usare nessuno degli altri specificatori del controllo di accesso, incluso `private`, `protected` o `internal`. Con i metodi delle funzioni di costruzione non è inoltre possibile utilizzare spazi dei nomi definiti dall'utente.

Una funzione di costruzione può effettuare una chiamata esplicita alla funzione di costruzione della sua superclasse diretta mediante l'istruzione `super()`. Se la funzione di costruzione della superclasse non viene esplicitamente chiamata, il compilatore inserisce automaticamente una chiamata prima della prima istruzione nel corpo della funzione di costruzione. Per chiamare i metodi della superclasse, potete inoltre utilizzare il prefisso `super` come riferimento alla superclasse. Se decidete di utilizzare sia `super()` che `super` nello stesso corpo della funzione di costruzione, assicuratevi di chiamare prima `super()`. In caso contrario, il riferimento `super` non funziona correttamente. La funzione di costruzione `super()` deve inoltre essere chiamata prima di eventuali istruzioni `throw` o `return`.

Nell'esempio che segue è illustrato cosa accade se si tenta di utilizzare il riferimento `super` prima di chiamare la funzione di costruzione `super()`. Una nuova classe, ExampleEx, estende la classe Example. La funzione di costruzione ExampleEx tenta di accedere alla variabile status definita all'interno della sua superclasse, ma lo fa prima di chiamare `super()`. L'istruzione `trace()` che si trova all'interno della funzione di costruzione ExampleEx genera il valore `null`, in quanto la variabile `status` non risulta disponibile fino all'esecuzione della funzione di costruzione `super()`.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Nonostante sia possibile utilizzare l'istruzione `return` all'interno di una funzione di costruzione, non è consentito che venga restituito un valore. In altre parole, le istruzioni `return` non devono avere espressioni o valori associati. Di conseguenza, i metodi delle funzioni di costruzione non possono restituire valori, ovvero non è possibile specificare tipi restituiti.

Se non si definisce un metodo di funzione di costruzione nella classe, il compilatore crea automaticamente una funzione di costruzione vuota. Se la classe viene estesa a un'altra classe, il compilatore include una chiamata `super()` nella funzione di costruzione generata.

Metodi statici

I metodi statici, chiamati anche *metodi di classe*, sono metodi che vengono dichiarati con la parola chiave `static`. I metodi statici, generalmente associati a una classe anziché all'istanza di una classe, sono utili per incorporare funzionalità relative a qualcosa di diverso dallo stato di singole istanze. Poiché i metodi statici vengono associati a un'intera classe, è possibile accedervi solo attraverso la classe stessa e non attraverso un'istanza della classe.

I metodi statici sono utili per incorporare funzioni che non si limitano a modificare lo stato delle istanze di classe. In altre parole, un metodo si definisce statico se fornisce funzionalità che non intervengono direttamente sul valore di un'istanza di classe. Ad esempio, la classe Date presenta un metodo statico denominato `parse()` che converte una stringa in un numero. Tale metodo è statico in quanto non modifica una singola istanza della classe. Il metodo `parse()` prende invece una stringa che rappresenta un valore data, l'analizza e restituisce un numero in un formato compatibile con la rappresentazione interna dell'oggetto Date. Questo metodo non è un metodo di istanza, in quanto non avrebbe senso applicarlo a un'istanza della classe Date.

Il metodo statico `parse()` è in contrasto con i metodi di istanza della classe Date, quale `getMonth()`. Il metodo `getMonth()` è un metodo di istanza, in quanto opera direttamente sul valore recuperando un componente specifico, il mese, di un'istanza Date.

Poiché i metodi statici non sono associati a singole istanze, non è possibile utilizzare le parole chiave `this` o `super` all'interno del corpo di un metodo statico. Sia il riferimento `this` che `super` hanno significato solo nel contesto di un metodo di istanza.

Diversamente da quanto accade in altri linguaggi di programmazione basati sulle classi, i metodi statici in ActionScript 3.0 non vengono ereditati.

Metodi di istanza

I metodi di istanza sono metodi dichiarati senza la parola chiave `static`. Questi metodi, che vengono associati alle istanze di una classe anziché all'intera classe, sono utili per implementare funzionalità che riguardano singole istanze di una classe. Ad esempio, la classe Array contiene un metodo di istanza denominato `sort()` che opera direttamente sulle istanze Array.

Nel corpo di un metodo di istanza, sono valide sia le variabili statiche che di istanza, il che significa che è possibile fare riferimento alle variabili definite all'interno della stessa classe mediante un semplice identificatore. Ad esempio, la classe CustomArray seguente estende la classe Array. La classe CustomArray definisce una variabile statica denominata `arrayCountTotal` che registra il numero totale di istanze della classe, una variabile di istanza denominata `arrayNumber` che registra l'ordine in cui le istanze sono state create e un metodo di istanza denominato `getPosition()` che restituisce i valori di tali variabili.

```
public class CustomArray extends Array
{
    public static var arrayCountTotal:int = 0;
    public var arrayNumber:int;

    public function CustomArray()
    {
        arrayNumber = ++arrayCountTotal;
    }

    public function getArrayPosition():String
    {
         return ("Array " + arrayNumber + " of " + arrayCountTotal);
    }
}
```

Se il codice esterno alla classe deve accedere alla variabile statica `arrayCountTotal` mediante l'oggetto di classe, utilizzando `CustomArray.arrayCountTotal`, il codice che risiede nel corpo del metodo `getPosition()` può fare riferimento direttamente alla variabile statica `arrayCountTotal`. Ciò vale anche per le variabili statiche contenute nelle superclassi. Anche se le proprietà statiche non vengono ereditate in ActionScript 3.0, quelle presenti nelle superclassi sono nell'area di validità. Ad esempio, la classe Array presenta poche variabili statiche, una delle quali è una costante denominata `DESCENDING`. Il codice residente in una delle sottoclassi di Array può accedere alla costante statica `DESCENDING` mediante un semplice identificatore.

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

Il valore del riferimento `this` nel corpo di un metodo di istanza è un riferimento all'istanza alla quale il metodo è associato. Il codice seguente illustra come il riferimento `this` punti all'istanza contenente il metodo:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

L'ereditarietà dei metodi di istanza può essere controllata con le parole chiave `override` e `final`. Utilizzate l'attributo `override` per ridefinire un metodo ereditato e l'attributo `final` per impedire alle sottoclassi di sostituire un metodo.

Metodi supplementari get e set

Le funzioni get e set, chiamate anche *getter* e *setter*, consentono di rispettare i principi di programmazione dell'information hiding e dell'incapsulamento, fornendo un'interfaccia di programmazione facile da usare per le classi create. Le funzioni get e set permettono di mantenere le proprietà della classe private all'interno della classe stessa, pur consentendo agli utenti della classe di accedere a tali proprietà come se stessero accedendo a una variabile di classe anziché chiamare un metodo di classe.

Il vantaggio di questo tipo di approccio è che consente di non utilizzare le funzioni accessor tradizionali con nomi poco pratici, quali `getPropertyName()` e `setPropertyName()`. Un altro vantaggio delle funzioni getter e setter è che evitano di gestire due funzioni pubbliche per ciascuna proprietà che consente l'accesso sia di lettura che di scrittura.

Nell'esempio seguente, la classe GetSet include le funzioni accessor get e set denominate `publicAccess()` che consentono l'accesso alla variabile privata denominata `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Se tentate di accedere alla proprietà `privateProperty` direttamente, si verifica l'errore seguente:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Gli utenti della classe GetSet utilizzano invece qualcosa che appare come una proprietà denominata `publicAccess`, ma che in realtà è una coppia di funzioni get e set che operano sulla proprietà privata chiamata `privateProperty`. Nell'esempio seguente viene creata un'istanza della classe GetSet, quindi viene impostato il valore di `privateProperty` mediante l'accessor pubblico denominato `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Le funzioni getter e setter consentono inoltre di sostituire le proprietà ereditate da una superclasse, cosa non possibile se si utilizzano normali variabili membro di classe. Le variabili membro di classe dichiarate con la parola chiave `var` non possono essere sostituite in una sottoclasse. Le proprietà create con le funzioni getter e setter invece non presentano questo limite. È possibile utilizzare l'attributo `override` sulle funzioni getter e setter ereditate da una superclasse.

Metodi vincolati

Un metodo vincolato, o *chiusura di un metodo*, è semplicemente un metodo estratto dalla propria istanza. Tra gli esempi di metodi vincolati vi sono metodi passati come argomenti a una funzione o restituiti come valori da una funzione. Tra le novità di ActionScript 3.0 vi è un metodo vincolato simile a una chiusura di funzione, in quanto in grado di conservare il proprio ambiente lessicale anche se estratto dall'istanza a cui è associato. Tuttavia, la principale differenza tra un metodo vincolato e una chiusura di funzione è che il riferimento `this` del metodo vincolato resta collegato, o vincolato, all'istanza che implementa il metodo. In altre parole, il riferimento `this` in un metodo vincolato punta sempre all'oggetto originale che ha implementato il metodo. Nelle funzioni di chiusura, il riferimento `this` è generico, vale a dire che punta a qualsiasi oggetto associato alla funzione nel momento in cui viene chiamato.

Una corretta comprensione dei metodi vincolati è importante se si utilizza la parola chiave `this`. Tenete presente che la parola chiave `this` fornisce un riferimento a un oggetto principale del metodo. Per la maggior parte dei programmatori ActionScript la parola chiave `this` rappresenta sempre l'oggetto o la classe che contiene la definizione di un metodo. Senza i metodi vincolati, tuttavia, ciò non sarebbe sempre possibile. Nelle versioni precedenti di ActionScript, ad esempio, il riferimento `this` non era sempre riferito all'istanza che aveva implementato il metodo. Se in ActionScript 2.0 i metodi vengono estratti da un'istanza, non solo il riferimento `this` resta vincolato all'istanza originale, ma le variabili di membro e i metodi della classe dell'istanza non risultano disponibili. Il problema non esiste in ActionScript 3.0, in quanto i metodi vincolati vengono automaticamente creati quando un metodo viene passato come parametro. I metodi vincolati garantiscono che la parola chiave `this` faccia sempre riferimento all'oggetto o alla classe nella quale il metodo viene definito.

Il codice seguente definisce una classe denominata ThisTest, contenente un metodo chiamato `foo()` che definisce a sua volta il metodo vincolato e un metodo `bar()` che restituisce il metodo vincolato. Il codice esterno alla classe crea un'istanza della classe ThisTest, chiama il metodo `bar()` e memorizza il valore restituito in una variabile denominata `myFunc`.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

Le ultime due righe del codice mostrano che il riferimento `this` del metodo vincolato `foo()` continua a puntare a un'istanza della classe ThisTest, anche se il riferimento `this` che si trova nella linea precedente punta all'oggetto global. Inoltre, il metodo vincolato memorizzato nella variabile `myFunc` può ancora accedere alle variabili di membro della classe ThisTest. Se lo stesso codice viene eseguito in ActionScript 2.0, il riferimento `this` corrisponderebbe e la variabile `num` risulterebbe `undefined`.

Un'area in cui l'inserimento dei metodi vincolati è più evidente è quella dei gestori di eventi, in quanto il metodo `addEventListener()` richiede il passaggio di una funzione o di un metodo come argomento.

Enumerazioni con classi

Le *enumerazioni* sono tipi di dati personalizzati creati per incapsulare un piccolo gruppo di valori. ActionScript 3.0 non supporta una funzionalità di enumerazione specifica, a differenza di C++, che presenta la parola chiave `enum`, o di Java, che ha un'interfaccia di enumerazione propria. È tuttavia possibile creare enumerazioni mediante le classi e le costanti statiche. Ad esempio, la classe PrintJob in ActionScript 3.0 impiega un'enumerazione denominata PrintJobOrientation per memorizzare i valori `"landscape"` e `"portrait"`, come illustrato nel codice seguente:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Per convenzione, una classe di enumerazione viene dichiarata con l'attributo `final` in quanto non vi è necessità di estendere la classe. La classe include solo membri statici, di conseguenza non si potranno creare istanze della classe. Al contrario, è possibile accedere ai valori di enumerazione direttamente tramite l'oggetto di classe, come illustrato nel seguente estratto di codice:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Tutte le classi di enumerazione in ActionScript 3.0 contengono solo variabili di tipo String, int o uint. Il vantaggio di utilizzare enumerazioni anziché stringhe di caratteri o valori numerici è che gli errori tipografici sono più facilmente rilevabili nelle enumerazioni. Se digitate in modo errato il nome di un'enumerazione, il compilatore di ActionScript genera un errore. Se usate valori letterali, il compilatore non rileva gli errori ortografici o l'uso di un numero errato. Nell'esempio precedente, se il nome della costante di enumerazione non è corretto, il compilatore genera un errore, come illustrato nell'estratto seguente:

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Tuttavia, il compilatore non genera un errore se la stringa di caratteri digitata contiene un errore ortografico, come nel caso seguente:

```
    if (pj.orientation == "portrai") // no compiler error
```

Una seconda tecnica per la creazione di enumerazioni prevede anch'essa la creazione di una classe separata con proprietà statiche per l'enumerazione. Ciò che differenzia questa tecnica, tuttavia, è che le proprietà statiche contengono un'istanza della classe anziché una stringa o un valore intero. Ad esempio, il seguente codice consente di creare una classe di enumerazione per i giorni della settimana:

```
public final class Day
{
    public static const MONDAY:Day = new Day();
    public static const TUESDAY:Day = new Day();
    public static const WEDNESDAY:Day = new Day();
    public static const THURSDAY:Day = new Day();
    public static const FRIDAY:Day = new Day();
    public static const SATURDAY:Day = new Day();
    public static const SUNDAY:Day = new Day();
}
```

Questa tecnica non viene utilizzata da ActionScript 3.0, ma è impiegata da numerosi sviluppatori che ne apprezzano in particolare la funzionalità di verifica tipi ottimizzata. Ad esempio, un metodo in grado di restituire un valore di enumerazione può limitare il valore restituito al tipo di dati dell'enumerazione. Il codice seguente mostra una funzione in grado di restituire non solo un giorno della settimana, ma anche una chiamata a una funzione che impiega il tipo dell'enumerazione come annotazione di tipo:

```
function getDay():Day
{
    var date:Date = new Date();
    var retDay:Day;
    switch (date.day)
    {
        case 0:
            retDay = Day.MONDAY;
            break;
        case 1:
            retDay = Day.TUESDAY;
            break;
        case 2:
            retDay = Day.WEDNESDAY;
            break;
        case 3:
            retDay = Day.THURSDAY;
            break;
        case 4:
            retDay = Day.FRIDAY;
            break;
        case 5:
            retDay = Day.SATURDAY;
            break;
        case 6:
            retDay = Day.SUNDAY;
            break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

È inoltre possibile ottimizzare la classe Day in modo che a ogni giorno della settimana venga associato un numero intero e fornire un metodo `toString()` che restituisca una rappresentazione del giorno sotto forma di stringa.

Classi delle risorse incorporate

In ActionScript 3.0 vengono impiegate classi speciali, chiamate *classi delle risorse incorporate*, per rappresentare risorse incorporate. Una *risorsa incorporata* è una risorsa, quale un suono, un'immagine o un carattere, che viene inclusa in un file SWF in fase di compilazione. L'incorporamento, anziché il caricamento dinamico, di una risorsa ne garantisce la disponibilità in fase di runtime, ma al costo di un incremento delle dimensioni del file SWF.

Uso delle classi delle risorse incorporate in Flash Professional

Per incorporare una risorsa, in primo luogo inseritela nella libreria di un file FLA. In secondo luogo, utilizzate la finestra di dialogo Proprietà del concatenamento della risorsa per specificare il nome della classe della risorsa incorporata. Se una classe con questo nome non esiste nel percorso di classe, ne viene generata una automaticamente. A questo punto, potete creare un'istanza della classe della risorsa incorporata e utilizzare qualsiasi proprietà e metodo definito o ereditato dalla classe. Ad esempio, il codice seguente può essere utilizzato per riprodurre l'audio incorporato collegato alla classe di una risorsa incorporata di nome PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

In alternativa, potete utilizzare il tag per metadati `[Embed]` per incorporare risorse in un progetto Flash Professional, come descritto di seguito. Se utilizzate il tag per metadati `[Embed]` nel codice, Flash Professional utilizza il compilatore Flex, anziché il compilatore Flash Professional, per compilare il progetto.

Uso delle classi delle risorse incorporate mediante il compilatore Flex

Se compilate il codice utilizzando il compilatore Flex, per incorporare una risorsa nel codice ActionScript utilizzate il tag per metadati `[Embed]`. Inserite la risorsa nella cartella di origine principale o in un'altra cartella del percorso di compilazione del progetto. Quando rileva il tag per metadati Embed, il compilatore di Adobe Flex crea automaticamente la classe della risorsa incorporata. Potete accedere alla classe usando una variabile con tipo di dati Class dichiarata subito dopo il tag per metadati `[Embed]`. Ad esempio il codice seguente incorpora un file audio denominato sound1.mp3 e usa una variabile denominata `soundCls` per memorizzare un riferimento alla classe della risorsa incorporata associata al file audio. Nell'esempio viene poi creata un'istanza della classe della risorsa incorporata e viene chiamato il metodo `play()` per quella istanza:

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

Adobe Flash Builder

Per utilizzare il tag per metadati `[Embed]` in un progetto ActionScript di Flash Builder, importate tutte le classi richieste dal framework Flex. Per incorporare i suoni, ad esempio, importate la classe mx.core.SoundAsset. Per usare il framework Flex, inserite il file framework.swc nel percorso di compilazione di ActionScript. In questo modo le dimensioni del file SWF aumentano.

Adobe Flex

In alternativa, in Flex potete incorporare una risorsa usando la direttiva `@Embed()` in una definizione del tag MXML.

Interfacce

Un'interfaccia è una raccolta di dichiarazioni di metodi che consente a oggetti non correlati di comunicare tra loro. Ad esempio, ActionScript 3.0 definisce l'interfaccia IEventDispatcher, che contiene dichiarazioni di metodi utilizzabili dalle classi per gestire gli oggetti evento. L'interfaccia IEventDispatcher stabilisce una modalità standard per il passaggio degli oggetti evento da un oggetto all'altro. Il codice seguente mostra la definizione dell'interfaccia IEventDispatcher:

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Le interfacce si basano sulla distinzione tra l'interfaccia di un metodo e la sua implementazione. L'interfaccia di un metodo include tutte le informazioni necessarie per chiamare tale metodo, incluso il nome del metodo, tutti i suoi parametri e il tipo restituito. L'implementazione di un metodo include non solo le informazioni sull'interfaccia, ma anche le istruzioni eseguibili che attivano il comportamento del metodo. La definizione di un'interfaccia contiene soltanto interfacce di metodo e tutte le classi che implementano l'interfaccia sono responsabili della definizione delle implementazioni del metodo.

In ActionScript 3.0, la classe EventDispatcher implementa l'interfaccia IEventDispatcher mediante la definizione di tutti i metodi dell'interfaccia IEventDispatcher e l'aggiunta di corpi di metodo a ognuno dei metodi. Il codice seguente è estratto dalla definizione della classe EventDispatcher:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

L'interfaccia IEventDispatcher serve da protocollo per le istanze di EventDispatcher che devono elaborare oggetti evento e passarli ad altri oggetti che hanno anch'essi implementato l'interfaccia IEventDispatcher.

Per descrivere un'interfaccia è anche possibile affermare che essa definisce un tipo di dati esattamente come fa una classe. Di conseguenza, un'interfaccia può essere usata come un'annotazione di tipo, allo stesso modo di una classe. Come tipo di dati, l'interfaccia può inoltre essere utilizzata con operatori, quali `is` e `as`, che richiedono un tipo di dati. A differenza di quanto avviene per le classi, tuttavia, non è possibile creare istanze di un'interfaccia. Questa distinzione ha portato molti programmatori a considerare le interfacce come tipi di dati astratti e le classi come tipi di dati concreti.

Definizione di un'interfaccia

La struttura della definizione di un'interfaccia è simile a quella della definizione di una classe, a eccezione del fatto che l'interfaccia può contenere solo metodi e non corpi dei metodi. Le interfacce inoltre non possono includere variabili o costanti, ma possono incorporare funzioni getter e setter. Per definire un'interfaccia, utilizzate la parola chiave `interface`. Ad esempio, la seguente interfaccia, IExternalizable, fa parte del pacchetto flash.utils in ActionScript 3.0. L'interfaccia IExternalizable definisce un protocollo per la serializzazione di un oggetto, vale a dire la conversione di un oggetto in un formato idoneo per la memorizzazione su un dispositivo o per la trasmissione in rete.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

L'interfaccia IExternalizable viene dichiarata con il modificatore del controllo di accesso `public`. Le definizioni di interfaccia possono essere modificate unicamente mediante gli specificatori del controllo di accesso `public` e `internal`. Le dichiarazioni dei metodi all'interno di una definizione di interfaccia non possono avere nessuno specificatore del controllo di accesso.

ActionScript 3.0 applica una convenzione in base alla quale i nomi di interfaccia iniziano con una `I` maiuscola, ma è possibile utilizzare qualsiasi identificatore valido come nome di interfaccia. Le definizioni di interfaccia vengono spesso collocate nel livello più alto di un pacchetto. Le definizioni di interfaccia non possono essere collocate all'interno di una definizione di classe o di un'altra definizione di interfaccia.

Le interfacce possono estendere altre interfacce. Ad esempio, l'interfaccia IExample seguente estende l'interfaccia IExternalizable:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Tutte le classi che implementano l'interfaccia IExample devono includere implementazioni non solo del metodo `extra()`, ma anche dei metodi `writeExternal()` e `readExternal()` ereditati dall'interfaccia IExternalizable.

Implementazione di un'interfaccia in una classe

La classe è il solo elemento del linguaggio di ActionScript 3.0 in grado di implementare un'interfaccia. Per implementare una o più interfacce, utilizzate la parola chiave `implements` nella dichiarazione della classe. Negli esempi seguenti vengono definite due interfacce, IAlpha e IBeta, e una classe, Alpha, che le implementa entrambe:

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

In una classe che implementa un'interfaccia, i metodi implementati devono:

- Utilizzare l'identificatore del controllo di accesso `public`.
- Utilizzare lo stesso nome del metodo di interfaccia.
- Avere lo stesso numero di parametri, ciascuno con tipi di dati corrispondenti ai tipi di dati dei parametri del metodo di interfaccia.
- Utilizzare lo stesso tipo restituito.

```
public function foo(param:String):String {}
```

È tuttavia possibile scegliere quale nome assegnare ai parametri dei metodi implementati. Anche se il numero dei parametri e il tipo di dati di ciascun parametro del metodo implementato devono corrispondere a quelli del metodo di interfaccia, i nomi dei parametri possono essere differenti. Nell'esempio precedente, il parametro del metodo `Alpha.foo()` è denominato `param`:

Mentre è denominato `str` nel metodo di interfaccia `IAlpha.foo()`:

```
function foo(str:String):String;
```

Potete disporre di una certa flessibilità anche per quanto riguarda i valori di parametro predefiniti. Una definizione di interfaccia può includere dichiarazioni di funzioni con valori di parametro predefiniti. Un metodo che implementa una tale dichiarazione di funzione deve avere un valore di parametro predefinito che sia membro dello stesso tipo di dati del valore specificato nella definizione di interfaccia, anche se il valore vero e proprio può essere differente. Ad esempio, il codice seguente definisce un'interfaccia che contiene un metodo con un valore di parametro predefinito 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

La definizione di classe seguente implementa l'interfaccia IGamma, ma impiega un valore di parametro predefinito differente:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Il motivo di questa flessibilità sta nel fatto che le regole di implementazione dell'interfaccia sono state specificamente studiate per garantire la compatibilità dei tipi di dati e per raggiungere tale obiettivo non è richiesta una corrispondenza dei nomi dei parametri e dei valori predefiniti.

Ereditarietà

L'ereditarietà è una forma di riutilizzo del codice che consente ai programmatori di sviluppare nuove classi basate sulle classi esistenti. Le classi esistenti vengono spesso definite *classi base* o *superclassi*, mentre le nuove classi sono generalmente chiamate *sottoclassi*. Uno dei principali vantaggi dell'ereditarietà è che consente di riutilizzare il codice di una classe di base, senza modificare il codice esistente. Inoltre, l'ereditarietà non richiede di modificare il modo in cui le altre classi interagiscono con la classe di base. Anziché modificare una classe esistente già ampiamente testata o già in uso, l'impiego dell'ereditarietà consente di trattare tale classe come un modulo integrato da estendere con proprietà o metodi aggiuntivi. Di conseguenza, la parola chiave `extends` viene utilizzata per indicare che una classe eredita da un'altra classe.

L'ereditarietà consente inoltre di sfruttare i vantaggi del *polimorfismo* all'interno del codice. Si definisce polimorfismo la capacità di usare un unico nome metodo per un metodo in grado di comportarsi in modo differente se applicato a diversi tipi di dati. Un semplice esempio è costituito da una classe base denominata Shape con due sottoclassi denominate Circle e Square. La classe Shape definisce un metodo chiamato `area()` che restituisce l'area della figura geometrica. Se è implementato il polimorfismo, potete chiamare il metodo `area()` sugli oggetti di tipo Circle e Square e ottenere i calcoli corretti. L'ereditarietà abilita il polimorfismo consentendo alle sottoclassi di ereditare e ridefinire (*sostituire*) i metodi della classe base. Nell'esempio seguente, il metodo `area()` viene ridefinito dalle classi Circle e Square:

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Poiché ogni classe definisce un tipo di dati, l'uso dell'ereditarietà crea una speciale relazione tra la classe base e la classe che la estende. Una sottoclasse possiede sempre tutte le proprietà della sua classe base, di conseguenza una qualsiasi istanza di una sottoclasse può sempre essere sostituita con un'istanza della classe base. Ad esempio, se un metodo definisce un parametro di tipo Shape, è consentito il passaggio di un argomento di tipo Circle, in quanto Circle è un'estensione di Shape, come indicato di seguito:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Proprietà di istanza ed ereditarietà

Una proprietà di istanza, sia che venga dichiarata con la parola chiave `function`, `var` o `const`, viene ereditata da tutte le sottoclassi a condizione che essa non sia stata dichiarata con l'attributo `private` nella classe base. Ad esempio, la classe Event in ActionScript 3.0 ha un numero di sottoclassi che ereditano proprietà comuni a tutti gli oggetti evento.

Per alcuni tipi di eventi, la classe Event contiene tutte le proprietà necessarie per la definizione dell'evento. Questi tipi di eventi non richiedono proprietà di istanza oltre a quelle definite nella classe Event. Esempi di tali eventi sono l'evento `complete`, che si verifica quando i dati vengono caricati correttamente, e l'evento `connect`, che si verifica quando viene stabilita una connessione di rete.

L'esempio seguente è un estratto della classe Event che illustra alcune delle proprietà e dei metodi ereditati dalle sottoclassi. Poiché vengono ereditate, le proprietà sono accessibili da una qualsiasi istanza di una sottoclasse.

```
public class Event
{
    public function get type():String;
    public function get bubbles():Boolean;
    ...

    public function stopPropagation():void {}
    public function stopImmediatePropagation():void {}
    public function preventDefault():void {}
    public function isDefaultPrevented():Boolean {}
    ...
}
```

Vi sono poi altri tipi di eventi che richiedono proprietà univoche non disponibili nella classe Event. Tali eventi vengono definiti utilizzando le sottoclassi della classe Event, in modo da consentire l'aggiunta di nuove proprietà alle proprietà già definite nella classe Event. Un esempio di tali sottoclassi è la classe MouseEvent, che aggiunge proprietà univoche per gli eventi associate ai movimenti o ai clic del mouse, quali gli eventi `mouseMove` e `click`. L'esempio seguente è un estratto della classe MouseEvent che mostra la definizione di proprietà esistenti all'interno della sottoclasse, ma non nella classe base:

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

Specificatori del controllo di accesso ed ereditarietà

Se viene dichiarata con la parola chiave `public`, una proprietà risulta visibile al codice ovunque si trovi. Ciò significa che la parola chiave `public`, a differenza delle parole chiave `private`, `protected` e `internal`, non pone limiti all'ereditarietà delle proprietà.

Se viene dichiarata con la parola chiave `private`, una proprietà risulterà visibile solo nella classe che la definisce e non verrà ereditata da alcuna delle sottoclassi. Ciò non accadeva nelle versioni precedenti di ActionScript, dove la parola chiave `private` si comportava in modo simile alla parola chiave `protected` di ActionScript 3.0.

La parola chiave `protected` indica che una proprietà è visibile non solo all'interno della classe che la definisce, ma anche in tutte le sue sottoclassi. A differenza della parola chiave `protected` del linguaggio di programmazione Java, la parola chiave `protected` di ActionScript 3.0 non rende una proprietà visibile a tutte le altre classi dello stesso pacchetto. In ActionScript 3.0, solo le sottoclassi possono accedere a una proprietà dichiarata con la parola chiave `protected`. Inoltre, una proprietà protetta è visibile a una sottoclasse, sia che questa si trovi nello stesso pacchetto della classe base che in un pacchetto differente.

Per limitare la visibilità di una proprietà al pacchetto nel quale essa è stata definita, utilizzate la parola chiave `internal` oppure non utilizzate nessun specificatore del controllo di accesso. Lo specificatore del controllo di accesso `internal` è lo specificatore predefinito che viene applicato quando non ne viene specificato nessuno. Se viene contrassegnata come `internal`, una proprietà viene ereditata unicamente dalle sottoclassi che risiedono nello stesso pacchetto.

L'esempio seguente mostra come ogni specificatore del controllo di accesso può modificare l'ereditarietà nell'ambito dei pacchetti. Il codice seguente definisce una classe di applicazione principale chiamata AccessControl e due altre classi chiamate Base ed Extender. La classe Base si trova in un pacchetto denominato foo, mentre la classe Extender, che è una sottoclasse della classe Base, si trova in un pacchetto chiamato bar. La classe AccessControl importa unicamente la classe Extender e crea un'istanza di Extender che tenta di accedere a una variabile chiamata `str` definita nella classe Base. La variabile `str` è dichiarata come `public`, di conseguenza il codice viene compilato ed eseguito come nell'estratto seguente:

```
// Base.as in a folder named foo
package foo
{
    public class Base
    {
        public var str:String = "hello"; // change public on this line
    }
}

// Extender.as in a folder named bar
package bar
{
    import foo.Base;
    public class Extender extends Base
    {
        public function getString():String {
            return str;
        }
    }
}

// main application class in file named AccessControl.as
package
{
    import flash.display.MovieClip;
    import bar.Extender;
    public class AccessControl extends MovieClip
    {
        public function AccessControl()
        {
            var myExt:Extender = new Extender();
            trace(myExt.str);// error if str is not public
            trace(myExt.getString()); // error if str is private or internal
        }
    }
}
```

Per vedere come gli altri specificatori del controllo di accesso possono modificare la compilazione e l'esecuzione dell'esempio precedente, impostate lo specificatore del controllo di accesso alla variabile `str` su `private`, `protected` o `internal`, dopo avere eliminato o escluso mediante commento la seguente riga dalla classe `AccessControl`:

```
trace(myExt.str);// error if str is not public
```

Sostituzione delle variabili non consentita

Le proprietà dichiarate con le parole chiave `var` o `const` vengono ereditate, ma non possono essere sostituite. Sostituire una proprietà significa ridefinirla in una sottoclasse. I solo tipi di proprietà che è possibile sostituire sono proprietà accessor get e set, vale a dire le proprietà dichiarate con la parola chiave `function`. Sebbene non sia possibile sostituire una variabile di istanza, potete ottenere un risultato simile mediante la creazione di metodi getter e setter per la variabile di istanza, quindi sostituire tali metodi.

Sostituzione di metodi

Sostituire un metodo significa ridefinire il comportamento di un metodo ereditato. I metodi statici non vengono ereditati e non possono essere sostituiti. I metodi di istanza, tuttavia, vengono ereditati dalle sottoclassi e possono essere sostituiti a condizione che vengano soddisfatti i seguenti criteri:

- Il metodo di istanza non deve essere dichiarato con la parola chiave `final` nella classe base. Se utilizzata con un metodo di istanza, la parola chiave `final` indica l'intenzione del programmatore di impedire alle sottoclassi di sostituire il metodo.
- Il metodo di istanza non deve essere dichiarato con lo specificatore del controllo di accesso `private` nella classe base. Se un metodo viene contrassegnato come `private` nella classe base, non è necessario utilizzare la parola chiave `override` per la definizione di un metodo con nome identico nella sottoclasse, in quanto il metodo della classe base non è visibile alla sottoclasse.

  Per sostituire un metodo di istanza che soddisfi i criteri di cui sopra, è necessario che nella definizione del metodo all'interno della sottoclasse venga utilizzata la parola chiave `override` e che tale definizione corrisponda alla versione del metodo della superclasse, nei modi indicati di seguito:

- Il metodo di sostituzione deve avere lo stesso livello di controllo di accesso del metodo della classe base. I metodi contrassegnati come interni hanno lo stesso livello di controllo di accesso dei metodi che non presentano alcuno specificatore del controllo di accesso.
- Il metodo di sostituzione deve avere lo stesso numero di parametri del metodo della classe base.
- I parametri del metodo di sostituzione devono avere le stesse annotazioni di tipo di dati dei parametri del metodo della classe base.
- Il metodo di sostituzione deve avere lo stesso tipo restituito del metodo della classe base.

I nomi dei parametri del metodo di sostituzione, tuttavia, non devono corrispondere ai nomi dei parametri del metodo della classe base, a condizione che il numero dei parametri e il tipo di dati di ciascun parametro corrisponda.

Istruzione super

Quando sostituiscono un metodo, i programmatori spesso intendono aggiungere qualcosa al comportamento del metodo della superclasse da sostituire, anziché rimpiazzare completamente tale comportamento. Ciò richiede un meccanismo che consenta a un metodo in una sottoclasse di chiamare la versione di se stesso presente nella superclasse. L'istruzione `super` consente tale operazione, in quanto contiene un riferimento all'immediata superclasse. Nell'esempio seguente viene definita una classe chiamata Base contenente un metodo chiamato `thanks()` e una sottoclasse della classe Base denominata Extender che sostituisce il metodo `thanks()`. Il metodo `Extender.thanks()` impiega l'istruzione `super` per chiamare `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Sostituzione di getter e setter

Nonostante non sia possibile sostituire le variabili definite in una superclasse, è invece possibile sostituire le funzioni getter e setter. Ad esempio, il codice seguente consente di sostituire una funzione getter denominata `currentLabel` definita nella classe MovieClip in ActionScript 3.0:

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

Il risultato dell'istruzione `trace()` nella funzione di costruzione della classe OverrideExample è `Override: null`, che mostra come l'esempio sia riuscito a sostituire la proprietà `currentLabel` ereditata.

Proprietà statiche non ereditate

Le proprietà statiche non vengono ereditate dalle sottoclassi. Ciò significa che non è possibile accedere alle proprietà statiche attraverso un'istanza di una sottoclasse. Una proprietà statica è accessibile unicamente attraverso l'oggetto di classe sul quale viene definita. Ad esempio, il codice seguente definisce una classe base chiamata Base e una sottoclasse che la estende denominata Extender. Nella classe Base viene definita una variabile statica chiamata `test`. Il codice riportato nell'estratto seguente non viene compilato in modalità rigorosa e genera un errore di runtime in modalità standard.

```
package {
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// error
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base { }
```

Il solo modo per accedere alla variabile statica `test` è mediante l'oggetto di classe, come illustrato nel codice seguente:

```
Base.test;
```

È tuttavia consentito definire una proprietà di istanza con lo stesso nome della proprietà statica. Tale proprietà di istanza può essere definita nella stessa classe della proprietà statica o in una sua sottoclasse. Ad esempio, la classe Base dell'esempio precedente potrebbe avere una proprietà di istanza denominata `test`. Il codice seguente viene compilato ed eseguito perché la proprietà di istanza viene ereditata dalla classe Extender. Il codice verrebbe compilato ed eseguito anche se la definizione della variabile di istanza test venisse spostata, ma non copiata, nella classe Extender.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Proprietà statiche e catena delle aree di validità

Nonostante non vengano ereditate, le proprietà statiche rientrano in una catena delle aree di validità della classe che definisce tali proprietà e tutte le sottoclassi della classe. Di conseguenza, le proprietà statiche vengono dette *nell'area di validità* della classe nella quale vengono definite e di tutte le sue sottoclassi. Ciò significa che una proprietà statica risulta direttamente accessibile all'interno del corpo della classe che definisce la proprietà statica e tutte le sottoclassi della classe.

Nell'esempio che segue vengono modificate le classi definite nell'esempio precedente, al fine di mostrare come la variabile statica `test` definita nella classe Base si trovi nell'area di validità della classe Extender. In altre parole, la classe Extender può accedere alla variabile statica `test` senza aggiungervi un prefisso corrispondente al nome della classe che definisce `test`.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

Se viene definita una proprietà di istanza che impiega lo stesso nome di una proprietà statica presente nella stessa classe o in una superclasse, la proprietà di istanza ha una precedenza maggiore nella catena delle aree di validità. La proprietà di istanza *prevarica* la proprietà statica, ovvero il valore della proprietà di istanza viene utilizzato al posto del valore della proprietà statica. Ad esempio, il codice seguente mostra che, se la classe Extender definisce una variabile di istanza denominata `test`, l'istruzione `trace()` impiega il valore della variabile di istanza anziché quello della variabile statica.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base
{
    public static var test:String = "static";
}

class Extender extends Base
{
    public var test:String = "instance";
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

Argomenti avanzati

Storia del supporto per la programmazione a oggetti in ActionScript

Poiché ActionScript 3.0 si basa sulle versioni precedenti di ActionScript, potrebbe risultare interessante comprendere come si è evoluto il modello a oggetti di ActionScript. ActionScript è nato come un semplice meccanismo per la creazione di script per le prime versioni di Flash Professional. In seguito, i programmatori hanno iniziato a creare applicazioni sempre più complesse con ActionScript. In risposta alle esigenze dei programmatori, in ogni versione successiva sono state aggiunte funzionalità di linguaggio finalizzate a facilitare la creazione di applicazioni complesse.

ActionScript 1.0

ActionScript 1.0 è la versione del linguaggio utilizzata in Flash Player 6 e versioni precedenti. Già in questa prima fase della sua evoluzione, il modello a oggetti di ActionScript si basava sul concetto di oggetto come tipo di dati fondamentale. Un oggetto di ActionScript è un tipo di dati composti con un gruppo di *proprietà*. Quando si parla di modello a oggetti, il termine *proprietà* include tutto ciò che viene associato a un oggetto, ad esempio variabili, funzioni e metodi.

Nonostante la prima generazione di ActionScript non supportasse la definizione di classi con la parola chiave `class`, era possibile definire una classe impiegando un tipo di oggetto speciale denominato oggetto prototipo. Invece di utilizzare la parola chiave `class` per creare una definizione astratta di classe a partire dalla quale creare istanze in oggetti concreti, come accade nei linguaggi basati sulle classi, quali Java e C++, i linguaggi basati sui prototipi, quali ActionScript 1.0, impiegano un oggetto esistente come modello (o prototipo) per creare altri oggetti. Se nei linguaggi basati sulle classi gli oggetti puntano a una classe che funge da modello, nei linguaggi basati sui prototipi, gli oggetti puntano invece a un altro oggetto, il loro prototipo, che funge da modello.

Per creare una classe in ActionScript 1.0, è necessario definire una funzione di costruzione per tale classe. In ActionScript, le funzioni sono veri e propri oggetti, non solo definizioni astratte. La funzione di costruzione creata funge da oggetto prototipo per le istanze della classe. Il codice seguente consente di creare una classe denominata Shape e di definire una proprietà chiamata `visible` configurata su `true` per impostazione predefinita:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Questa funzione di costruzione definisce una classe Shape dalla quale è possibile creare un'istanza mediante l'operatore `new`, come segue:

```
myShape = new Shape();
```

L'oggetto funzione di costruzione `Shape()` oltre a servire da prototipo per la creazione di istanze della classe Shape, può inoltre fungere da prototipo per la creazione di sottoclassi di Shape, vale a dire di altre classi che estendono la classe Shape.

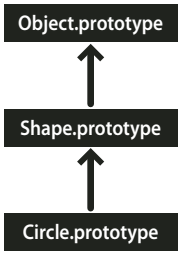
Per creare una sottoclasse della classe Shape è necessario seguire una procedura in due fasi. In primo luogo, si deve creare la classe mediante la definizione di una funzione di costruzione per la classe, come segue:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Quindi, è necessario usare l'operatore `new` per dichiarare che la classe Shape è il prototipo della classe Circle. Per impostazione predefinita, qualsiasi classe creata impiega la classe Object come prototipo, di conseguenza il valore `Circle.prototype` contiene un oggetto generico (un'istanza della classe Object). Per specificare che il prototipo di Circle deve essere Shape anziché Object, è necessario utilizzare il codice seguente per modificare il valore `Circle.prototype` in modo che contenga un oggetto Shape invece di un oggetto generico:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

La classe Shape e la classe Circle sono ora collegate in una relazione di ereditarietà comunemente conosciuta come *catena di prototipi*. Il diagramma seguente illustra le varie relazioni in una catena di prototipi:



La classe alla base di ogni catena di prototipi è la classe Object. La classe Object contiene una proprietà statica denominata `Object.prototype` che punta all'oggetto prototype di base di tutti gli oggetti creati con ActionScript 1.0. L'oggetto successivo nella catena di prototipi è l'oggetto Shape. La proprietà `Shape.prototype` infatti non è mai stata esplicitamente impostata e contiene ancora un oggetto generico (un'istanza della classe Object). Il collegamento finale della catena è costituito dalla classe Circle, collegata al suo prototipo, la classe Shape (la proprietà `Circle.prototype` contiene un oggetto Shape).

Se create un'istanza della classe Circle, come nell'esempio seguente, tale istanza eredita la catena di prototipi della classe Circle:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Ricordate che l'esempio comprende una proprietà denominata `visible`, membro della classe Shape. Nell'esempio, la proprietà `visible` non esiste come parte dell'oggetto `myCircle`, ma solo come membro dell'oggetto Shape, tuttavia, la riga di codice seguente restituisce `true`:

```
trace(myCircle.visible); // output: true
```

Il runtime è in grado di determinare che l'oggetto `myCircle` ha ereditato la proprietà `visible` risalendo la catena di prototipi. Quando esegue questo codice, il runtime cerca in primo luogo all'interno delle proprietà dell'oggetto `myCircle` una proprietà chiamata `visible`, ma non la trova. Il runtime esegue quindi una ricerca all'interno dell'oggetto `Circle.prototype`, ma neppure lì trova una proprietà denominata `visible`. Risalendo lungo la catena dei prototipi, il runtime trova infine la proprietà `visible` definita nell'oggetto `Shape.prototype` e restituisce il valore di tale proprietà.

Per una maggiore semplicità, in questa sezione sono stati omessi numerosi dettagli relativi alla catena di prototipi, per fornire solo le informazioni necessarie a comprendere il modello a oggetti di ActionScript 3.0.

ActionScript 2.0

In ActionScript 2.0 vengono introdotte nuove parole chiave, quali `class`, `extends`, `public` e `private`, che consentono di definire le classi in maniera simile ai linguaggi basati sulle classi, quali Java e C++. È importante tenere presente che il meccanismo di ereditarietà alla base non è cambiato con il passaggio da ActionScript 1.0 ad ActionScript 2.0. In ActionScript 2.0 è stato semplicemente aggiunto un nuovo tipo sintassi per la definizione delle classi. La catena dei prototipi funziona alla stessa maniera in entrambe le versioni del linguaggio.

La nuova sintassi introdotta da ActionScript 2.0, e illustrata nell'estratto seguente, consente di definire le classi in un modo che molti programmatori trovano più intuitivo:

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

Tenete inoltre presente che in ActionScript 2.0 sono state introdotte anche le annotazioni di tipo da utilizzare con la verifica del tipo in fase di compilazione, che consentono di dichiarare che la proprietà `visible` dell'esempio precedente contiene solamente un valore booleano. Anche la nuova parola chiave `extends` semplifica il processo di creazione delle sottoclassi. Nell'esempio seguente, la procedura in due fasi necessaria in ActionScript 1.0 viene portata a termine in una sola fase, grazie all'introduzione della parola chiave `extends`:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

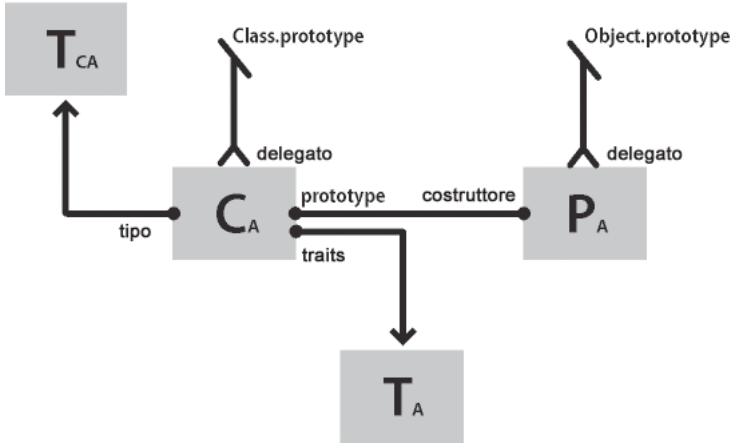
La funzione di costruzione viene ora dichiarata come parte della definizione della classe e le proprietà di classe `id` e `radius` devono anch'esse essere dichiarate esplicitamente.

In ActionScript 2.0 è stato inoltre introdotto il supporto della definizione di interfacce, che consente di rendere ancora più specifici i programmi orientati agli oggetti mediante protocolli formalmente definiti per la comunicazione tra oggetti.

Oggetto classe di ActionScript 3.0

Un comune paradigma di programmazione orientato agli oggetti, generalmente associato a Java e C++, impiega le classi per definire i tipi di oggetti. I linguaggi di programmazione che adottano questo paradigma tendono anch'essi a utilizzare le classi per costruire istanze del tipo di dati definito dalla classe stessa. ActionScript impiega le classi per entrambi questi scopi, tuttavia, le sue origini di linguaggio basato sui prototipi gli attribuisce un'interessante caratteristica. Per ogni definizione di classe in ActionScript viene creato uno speciale oggetto di classe che consente la condivisione di comportamento e stato della classe. Per numerosi programmatori ActionScript tuttavia, tale distinzione potrebbe non avere alcuna implicazione pratica in termini di codifica. ActionScript 3.0 è stato progettato in modo da consentire la creazione di sofisticate applicazioni ActionScript orientate agli oggetti senza la necessità di usare, o addirittura comprendere, tali particolari oggetti di classe.

Il diagramma seguente riporta la struttura di un oggetto di classe che rappresenta una semplice classe denominata A, definita con l'istruzione `class A {}`:



Ogni rettangolo del diagramma rappresenta un oggetto. Ogni oggetto del diagramma è seguito da una lettera in carattere pedice a indicare che appartiene alla classe A. L'oggetto di classe (CA) contiene riferimenti a vari altri oggetti importanti. L'oggetto traits dell'istanza (TA) conserva in memoria le proprietà dell'istanza definite nella definizione della classe. L'oggetto traits della classe (TCA) rappresenta il tipo interno della classe e conserva in memoria le proprietà statiche definite dalla classe (la lettera C in carattere pedice sta per "classe"). L'oggetto prototype (PA) si riferisce sempre all'oggetto di classe al quale era originariamente associato mediante la proprietà `constructor`.

Oggetto traits

L'oggetto traits, una novità di ActionScript 3.0, è stato implementato ai fini delle prestazioni. Nelle versioni precedenti di ActionScript, il processo di ricerca dei nomi poteva risultare lento e laborioso, in quanto Flash Player doveva risalire l'intera catena di prototipi. In ActionScript 3.0, le operazioni di ricerca dei nomi sono molto più efficienti e veloci, poiché le proprietà ereditate vengono copiate dalle superclassi negli oggetti traits delle sottoclassi.

L'oggetto traits non è direttamente accessibile dal codice di programmazione, tuttavia, la sua presenza è riscontrabile in termini di miglioramenti delle prestazioni e dell'uso della memoria. L'oggetto traits fornisce ad AVM2 informazioni dettagliate sul layout e il contenuto delle classi. Grazie a tali informazioni, AVM2 è in grado di ridurre sensibilmente i tempi di esecuzione, in quanto può generare spesso istruzioni dirette alla macchina per l'accesso immediato a proprietà o la chiamata di metodi, senza che siano necessarie lente e laboriose ricerche di nomi.

L'oggetto traits consente inoltre di ridurre sensibilmente l'occupazione della memoria di un oggetto rispetto a quanto poteva occupare un oggetto simile nelle versioni precedenti di ActionScript. Ad esempio, se una classe è chiusa (vale a dire, non è dichiarata dinamica), un'istanza di tale classe non richiede una tabella hash per le proprietà aggiunte dinamicamente e può contenere qualcosa in più che un semplice puntatore agli oggetti traits e alcuni slot per le proprietà fisse definite nella classe. Di conseguenza, a un oggetto che richiedeva 100 byte di memoria in ActionScript 2.0 potrebbero bastare 20 byte in ActionScript 3.0.

Nota: *l'oggetto traits è un dettaglio interno di implementazione e potrebbe essere modificato o addirittura eliminato nelle versioni future di ActionScript.*

Oggetto prototype

Ogni oggetto della classe di ActionScript presenta una proprietà chiamata `prototype` che funge da riferimento all'oggetto prototype della classe. L'oggetto prototype è un retaggio delle origini di ActionScript come linguaggio basato sui prototipi. Per ulteriori informazioni, vedete Storia del supporto per la programmazione a oggetti in ActionScript.

La proprietà `prototype` è una proprietà di sola lettura, vale a dire che non può essere modificata per puntare a oggetti differenti. Ciò la differenzia dalla proprietà di classe `prototype` delle versioni precedenti di ActionScript, dove era possibile riassegnare il prototipo in modo che puntasse a una classe diversa. Nonostante la proprietà `prototype` sia di sola lettura, l'oggetto prototype a cui fa riferimento non lo è. In altre parole, è possibile aggiungere nuove proprietà all'oggetto prototype. Le proprietà aggiunte all'oggetto prototype vengono condivise tra tutte le istanze della classe.

La catena di prototipi, che era il solo meccanismo di ereditarietà delle versioni precedenti di ActionScript, ha soltanto un ruolo secondario in ActionScript 3.0. Il sistema di ereditarietà primario, ovvero l'ereditarietà delle proprietà fisse, viene gestito internamente dall'oggetto traits. Una proprietà fissa è una variabile o metodo definito in una definizione di classe. L'ereditarietà delle proprietà fisse è chiamata anche ereditarietà di classe, in quanto il meccanismo di ereditarietà è associato a parole chiave quali `class`, `extends` e `override`.

La catena di prototipi offre un meccanismo di ereditarietà alternativo molto più dinamico dell'ereditarietà di proprietà fisse. Potete aggiungere proprietà all'oggetto prototype di una classe non solo come parte della definizione della classe, ma anche in fase di runtime mediante la proprietà `prototype` dell'oggetto di classe. Tenete presente, tuttavia, che se impostate il compilatore in modalità rigorosa, potreste non essere in grado di accedere a proprietà aggiunte a un oggetto prototype a meno che non dichiariate una classe con la parola chiave `dynamic`.

Un buon esempio di classe con numerose proprietà associate all'oggetto prototype è costituito dalla classe Object. I metodi `toString()` e `valueOf()` della classe Object sono in realtà funzioni assegnate a proprietà dell'oggetto prototype della classe Object. L'esempio seguente illustra come la dichiarazione di tali metodi potrebbe, teoricamente, apparire (l'effettiva implementazione è leggermente differente a causa di dettagli di implementazione):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Come accennato in precedenza, potete associare una proprietà all'oggetto prototype di una classe al di fuori della definizione della classe. Ad esempio, anche il metodo `toString()` può essere definito al di fuori della definizione della classe Object, come segue:

```
Object.prototype.toString = function()
{
    // statements
};
```

A differenza dell'ereditarietà di proprietà fisse tuttavia, l'ereditarietà di prototipi non richiede la parola chiave `override` per la ridefinizione di un metodo in una sottoclasse. Ad esempio, se desiderate ridefinire il metodo `valueOf()` in una sottoclasse della classe Object, avete a disposizione tre diverse opzioni. In primo luogo, potete definire un metodo `valueOf()` sull'oggetto prototype della sottoclasse, all'interno della definizione della classe. Il codice seguente consente di creare una sottoclasse di Object chiamata Foo e di ridefinire il metodo `valueOf()` sull'oggetto prototype di Foo, nell'ambito della definizione della classe. Poiché ogni classe eredita da Object, non è necessario usare la parola chiave `extends`.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

In secondo luogo, potete definire un metodo `valueOf()` sull'oggetto prototype di Foo al di fuori della definizione della classe, come illustrato nel codice seguente:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Infine, potete definire una proprietà fissa denominata `valueOf()` come parte della classe Foo. Questa tecnica si differenzia dalle altre in quanto mescola il sistema di ereditarietà di proprietà fisse con il sistema di ereditarietà di prototipi. Tutte le sottoclassi di Foo che desiderano ridefinire `valueOf()` devono utilizzare la parola chiave `override`. Nel codice seguente è illustrato `valueOf()` definito come proprietà fissa di Foo:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

Spazio dei nomi di AS3

L'esistenza di due meccanismi di ereditarietà distinti, l'ereditarietà di prototipi e di proprietà fisse, pone un'interessante problema di compatibilità nell'ambito delle proprietà e dei metodi delle classi principali. La compatibilità con la specifica del linguaggio ECMAScript su cui è basato ActionScript prevede l'uso dell'ereditarietà di prototipi, ovvero le proprietà e i metodi di una classe principale devono essere definiti sull'oggetto prototype della classe. D'altro canto, la compatibilità con ActionScript 3.0 richiede l'uso dell'ereditarietà di proprietà fisse, ovvero che le proprietà e i metodi di una classe principale vengano definite nell'ambito della definizione della classe mediante le parole chiave `const`, `var` e `function`. Inoltre, l'uso di proprietà fisse in luogo dei prototipi può portare a significativi incrementi delle prestazioni in fase di runtime.

ActionScript 3.0 risolve il problema utilizzando sia l'ereditarietà di prototipi che di proprietà fisse per le classi principali. Ogni classe principale contiene due serie di proprietà e metodi. Una serie viene definita sull'oggetto prototype per la compatibilità con la specifica ECMAScript, mentre l'altra serie viene definita con proprietà fisse e lo spazio dei nomi di AS3 per la compatibilità con ActionScript 3.0.

Lo spazio dei nomi di AS3 fornisce un pratico meccanismo che consente di scegliere tra le due serie di metodi e proprietà. Se non utilizzate lo spazio dei nomi di AS3, un'istanza di classe principale eredita le proprietà e i metodi definiti nell'oggetto prototype della classe principale. Se invece decidete di utilizzare lo spazio dei nomi di AS3, un'istanza di classe principale eredita le versioni di AS3, in quanto le proprietà fisse hanno sempre la precedenza rispetto alle proprietà prototype. In altre parole, se è disponibile, la proprietà fissa viene sempre utilizzata al posto di una proprietà prototype con nome identico.

Per scegliere di usare la versione dello spazio dei nomi di AS3 di una proprietà o di un metodo, è necessario qualificare la proprietà o il metodo con lo spazio dei nomi di AS3. Ad esempio, il codice seguente impiega la versione AS3 del metodo `Array.pop()`:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Oppure, potete utilizzare la direttiva `use namespace` per aprire lo spazio dei nomi AS3 di tutte le definizioni racchiuse in un blocco di codice. Ad esempio, il codice seguente impiega la direttiva `use namespace` per aprire lo spazio dei nomi di AS3 dei metodi `pop()` e `push()`:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 presenta inoltre opzioni del compilatore per ciascuna serie di proprietà, per consentire l'applicazione dello spazio dei nomi di AS3 all'intero programma. L'opzione del compilatore `-as3` rappresenta lo spazio dei nomi di AS3, mentre l'opzione del compilatore `-es` rappresenta l'opzione di ereditarietà dei prototipi (`es` sta per ECMAScript). Per aprire lo spazio dei nomi di AS3 per l'intero programma, impostate l'opzione del compilatore `-as3` su `true` e l'opzione del compilatore `-es` su `false`. Per utilizzare le versioni dei prototipi, impostate le opzioni del compilatore sui valori opposti. Le impostazioni predefinite del compilatore per Flash Builder sono `-as3 = true` e `-es = false`.

Se avete intenzione di estendere una delle classi principali e di sostituire uno o più metodi, dovete sapere come lo spazio dei nomi di AS3 può influenzare la modalità di dichiarazione di un metodo sostituito. Se utilizzate lo spazio dei nomi AS3, anche in ogni sostituzione di metodo di una classe principale dovete utilizzare lo spazio dei nomi AS3 insieme all'attributo `override`. Se non usate lo spazio dei nomi AS3 e desiderate ridefinire un metodo di classe principale all'interno di una sottoclasse, non dovete utilizzare lo spazio dei nomi AS3 né la parola chiave `override`.

Esempio: GeometricShapes

L'applicazione di esempio GeometricShapes illustra come è possibile applicare una serie di concetti e funzioni orientati agli oggetti utilizzando ActionScript 3.0, ad esempio:

- Definizione di classi
- Estensione di classi
- Polimorfismo e parola chiave `override`
- Definizione, estensione e implementazione delle interfacce

L'esempio include inoltre un "metodo factory" per la creazione di istanze di classi che illustra come dichiarare un valore restituito come un'istanza di un'interfaccia e utilizzare tale oggetto restituito in maniera generica.

Per ottenere i file dell'applicazione per questo esempio, vedete www.adobe.com/go/learn_programmingAS3samples_flash_it. I file dell'applicazione GeometricShapes si trovano nella cartella Samples/GeometricShapes. L'applicazione è composta dai seguenti file:

File	Descrizione
GeometricShapes.mxml oppure GeometricShapes.fla	Il file principale dell'applicazione in Flash (FLA) o Flex (MXML)
com/example/programmingas3/geometricshapes/IGeometricShape.as	Metodi di definizione interfaccia di base da implementare in tutte le classi dell'applicazione GeometricShapes.
com/example/programmingas3/geometricshapes/IPolygon.as	Metodo di definizione di interfaccia da implementare nelle classi dell'applicazione GeometricShapes che presentano più lati.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Tipo di figura geometrica che presenta lati di uguale lunghezza posizionati simmetricamente attorno al centro della figura.
com/example/programmingas3/geometricshapes/Circle.as	Tipo di figura geometrica che definisce un cerchio.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Sottoclasse di RegularPolygon che definisce un triangolo con lati di pari lunghezza.
com/example/programmingas3/geometricshapes/IPolygon.as	Sottoclasse di RegularPolygon che definisce un rettangolo con quattro lati di pari lunghezza.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Classe contenente un metodo factory per la creazione di figure geometriche a partire da una forma e da una dimensione specificate.

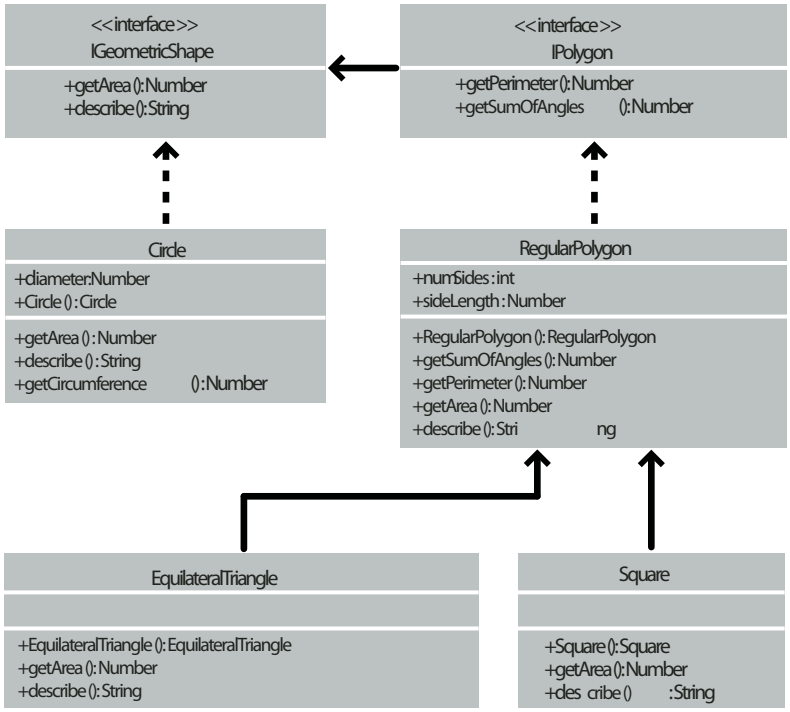
Definizione delle classi GeometricShapes

L'applicazione GeometricShapes consente di specificare un tipo di figura geometrica e una dimensione. Viene quindi restituita una descrizione della figura, la relativa area e il perimetro.

L'interfaccia utente dell'applicazione è molto semplice; essa include alcuni comandi per la selezione del tipo di figura geometrica, l'impostazione delle dimensione e la visualizzazione della descrizione. La parte più interessante di questa applicazione sta sotto la superficie, nella struttura delle classi e delle interfacce stesse.

Si tratta di un'applicazione che si occupa di figure geometriche pur senza visualizzarle graficamente.

Le classi e le interfacce che definiscono le figure geometriche di questo esempio sono visualizzate nel diagramma seguente utilizzando la notazione del linguaggio UML (Unified Modeling Language):



Classi di esempio GeometricShapes

Definizione di comportamenti comuni alle interfacce

L'applicazione GeometricShapes tratta tre diversi tipi di figure geometriche: cerchio, quadrato e triangolo equilatero. La struttura della classe GeometricShapes inizia con un'interfaccia molto semplice, IGeometricShape, che elenca una serie di metodi comuni a tutte e tre le figure geometriche:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

L'interfaccia definisce due metodi: il metodo `getArea()`, che calcola e restituisce l'area della figura geometrica, e il metodo `describe()`, che assembla una descrizione in formato testo delle proprietà della figura.

È opportuno inoltre conoscere il perimetro di ciascuna figura. Tuttavia, il perimetro del cerchio viene definito circonferenza e viene calcolato in modo univoco, di conseguenza, in questo caso, il comportamento differisce da quello per calcolare il perimetro di un triangolo o un quadrato. Vi sono comunque sufficienti analogie tra triangoli, quadrati e altri poligoni da consentire la definizione di una nuova classe di interfaccia esclusiva per tali figure: IPolygon. Anche l'interfaccia IPolygon è piuttosto semplice, come illustrato di seguito:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Questa interfaccia definisce due metodi comuni a tutti i poligoni: il metodo `getPerimeter()` che misura la distanza combinata di tutti i lati e il metodo `getSumOfAngles()` che somma tutti gli angoli interni.

L'interfaccia IPolygon estende l'interfaccia IGeometricShape, di conseguenza tutte le classi che implementano l'interfaccia IPolygon devono dichiarare tutti e quattro i metodi, i due dell'interfaccia IGeometricShape e i due dell'interfaccia IPolygon.

Definizione delle classi delle figure geometriche

Una volta a conoscenza dei metodi comuni a ogni tipo di figura, potete passare alla definizione delle classi delle figure geometriche. In termini di quantità di metodi che è necessario implementare, la figura geometrica più semplice corrisponde alla classe Circle visualizzata di seguito:

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

La classe Circle implementa l'interfaccia IGeometricShape, di conseguenza essa deve fornire codice sia per il metodo `getArea()` che per il metodo `describe()`. Inoltre, essa definisce il metodo `getCircumference()`, univoco per la classe Circle. Nella classe Circle viene infine dichiarata una proprietà, `diameter`, che non si trova nelle altre classi dei poligoni.

Gli altri due tipi di due figure geometriche, quadrato e triangolo equilatero, presentano altre somiglianze: entrambe hanno lati di pari lunghezza e vi sono formule comuni utilizzabili per calcolarne il perimetro e la somma degli angoli interni. In realtà, tali formule comuni sono applicabili a qualsiasi altro poligono regolare definito in futuro.

La classe RegularPolygon è la superclasse sia della classe Square che della classe EquilateralTriangle. Una superclasse consente di definire metodi comuni in una sola posizione, quindi non sarà necessario definirli separatamente in ogni sottoclasse. Segue il codice per la classe RegularPolygon:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
```

```
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

In primo luogo, la classe RegularPolygon dichiara due proprietà comuni a tutti i poligoni regolari: la lunghezza di ciascun lato (proprietà `sideLength`) e il numero di lati (proprietà `numSides`).

La classe RegularPolygon implementa l'interfaccia IPolygon e dichiara tutti e quattro i metodi dell'interfaccia IPolygon. Essa implementa inoltre due di essi (i metodi `getPerimeter()` e `getSumOfAngles()`) utilizzando formule comuni.

Poiché la formula del metodo `getArea()` è diversa in base al tipo di figura geometrica, la versione della classe base del metodo non può includere logica comune ereditabile dai metodi delle sottoclassi. Al contrario, restituisce semplicemente un valore 0 predefinito a indicare che l'area non è stata calcolata. Per calcolare correttamente l'area di ogni figura geometrica, le sottoclassi della classe RegularPolygon devono sostituire il metodo `getArea()`.

Il codice seguente della classe EquilateralTriangle mostra come è possibile sostituire il metodo `getArea()`:

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

La parola chiave `override` indica che il metodo `EquilateralTriangle.getArea()` sostituisce intenzionalmente il metodo `getArea()` dalla superclasse RegularPolygon. Quando viene chiamato, il metodo `EquilateralTriangle.getArea()` calcola l'area utilizzando la formula del codice precedente, mentre il codice del metodo `RegularPolygon.getArea()` non viene mai eseguito.

Al contrario, la classe EquilateralTriangle non definisce una propria versione del metodo `getPerimeter()`. Quando il metodo `EquilateralTriangle.getPerimeter()` viene chiamato, la chiamata risale la catena di ereditarietà e viene eseguito il codice del metodo `getPerimeter()` della superclasse RegularPolygon.

La funzione di costruzione `EquilateralTriangle()` impiega l'istruzione `super()` per chiamare esplicitamente la funzione di costruzione `RegularPolygon()` della sua superclasse. Se entrambe le funzioni di costruzione presentano la stessa serie di parametri, anche se la funzione `EquilateralTriangle()` venisse omessa, la funzione di costruzione `RegularPolygon()` verrebbe eseguita al suo posto. Tuttavia, la funzione di costruzione `RegularPolygon()` richiede un parametro aggiuntivo, `numSides`. Quindi, la funzione di costruzione `EquilateralTriangle()` chiama `super(len, 3)`, che trasmette il parametro di input `len` e il valore 3 per indicare che il triangolo ha tre lati.

Anche il metodo `describe()` utilizza l'istruzione `super()`, ma in una maniera differente, cioè per chiamare la versione della superclasse RegularPolygon del metodo `describe()`. Il metodo `EquilateralTriangle.describe()` imposta in primo luogo la variabile di stringa `desc` in modo che indichi un'istruzione relativa al tipo di figura. Quindi, ottiene il risultato del metodo `RegularPolygon.describe()` chiamando `super.describe()` e aggiunge tale risultato alla stringa `desc`.

La classe Square non viene descritta dettagliatamente in questa sezione, tuttavia essa è simile alla classe EquilateralTriangle, in quanto fornisce una funzione di costruzione e modalità proprie di implementazione dei metodi `getArea()` e `describe()`.

Polimorfismo e metodo factory

Una serie di classi che impiega in modo proficuo interfacce ed ereditarietà può essere utilizzata in tanti modi interessanti. Ad esempio, tutte le classi delle figure geometriche descritte in precedenza o implementano l'interfaccia IGeometricShape o estendono una superclasse che lo fa. Di conseguenza, se definite una variabile come istanza di IGeometricShape, non dovete sapere se si tratta di un'istanza della classe Circle o della classe Square per chiamare il suo metodo `describe()`.

Il codice seguente illustra come viene applicato questo principio:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Quando `myShape.describe()` viene chiamata, essa esegue il metodo `Circle.describe()` perché anche se la variabile è definita come un'istanza dell'interfaccia IGeometricShape, Circle è la sua classe sottostante.

L'esempio seguente mostra il principio del polimorfismo in azione: l'esatta chiamata di uno stesso metodo porta all'esecuzione di codici differenti, a seconda della classe dell'oggetto il cui metodo viene chiamato.

L'applicazione GeometricShapes applica questo tipo di polimorfismo basato sulle interfacce impiegando una versione semplificata di un modello di progettazione conosciuto come metodo factory. Il termine *metodo factory* indica una funzione che restituisce un oggetto il cui tipo di dati o contenuto sottostante può differire in base al contesto.

La classe GeometricShapeFactory illustrata qui definisce un metodo factory denominato `createShape()`:

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

Il metodo factory `createShape()` consente alle funzioni di costruzione delle sottoclassi delle figure geometriche di definire i dettagli delle istanze da esse create, ma di restituire i nuovi oggetti come istanze di IGeometricShape, in modo che possano essere gestiti dall'applicazione in maniera più generica.

Il metodo `describeShape()` dell'esempio precedente mostra come un'applicazione può utilizzare il metodo factory per ottenere un riferimento generico a un oggetto più specifico. L'applicazione è in grado di ottenere la descrizione di un oggetto Circle appena creato come segue:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

Il metodo `describeShape()` chiama quindi il metodo factory `createShape()` con gli stessi parametri, memorizzando il nuovo oggetto Circle in una variabile statica denominata `currentShape`, originariamente inserita come un oggetto di IGeometricShape. Quindi, viene chiamato il metodo `describe()` sull'oggetto `currentShape` e la chiamata viene automaticamente risolta per eseguire il metodo `Circle.describe()`, che restituisce una descrizione dettagliata del cerchio.

Ottimizzazione dell'applicazione di esempio

La reale portata di interfacce ed ereditarietà diventa evidente quando un'applicazione viene ottimizzata o modificata.

Supponete di dover aggiungere una nuova figura geometrica, un pentagono, all'applicazione di esempio. Verrebbe creata una classe Pentagon che estende la classe RegularPolygon e definisce versioni proprie dei metodi `getArea()` e `describe()`. Quindi verrebbe aggiunta una nuova opzione Pentagon alla casella combinata dell'interfaccia utente dell'applicazione. Niente altro. La classe Pentagon otterrebbe automaticamente le funzionalità del metodo `getPerimeter()` e del metodo `getSumOfAngles()` ereditandole dalla classe RegularPolygon. Poiché eredita da una classe che implementa l'interfaccia IGeometricShape, un'istanza di Pentagon può essere considerata anche un'istanza di IGeometricShape. In pratica ogni volta che desiderate aggiungere un nuovo tipo di figura geometrica, non sarà necessario modificare la firma di nessuno dei metodi della classe GeometricShapeFactory (e quindi non sarà necessario modificare nemmeno i codici che utilizzano la classe GeometricShapeFactory).

Potete aggiungere una classe Pentagon all'esempio Geometric Shapes come esercizio, per vedere come le interfacce e l'ereditarietà possono facilitare le operazioni di aggiunta di nuove funzioni a un'applicazione.