Aprendizaje de
ACTIONSCRIPT® 3.0

© 2010 Adobe Systems Incorporated. All rights reserved.

Aprendizaje de ActionScript® 3.0

This guide is protected under copyright law, furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

This guide is licensed for use under the terms of the Creative Commons Attribution Non-Commercial 3.0 License. This License allows users to copy, distribute, and transmit the guide for noncommercial purposes only so long as (1) proper attribution to Adobe is given as the owner of the guide; and (2) any reuse or distribution of the guide contains a notice that use of the guide is governed by these terms. The best way to provide notice is to include the following link. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/

Adobe, the Adobe logo, Adobe AIR, ActionScript, AIR, Flash, Flash Builder, Flash Lite, Flex, MXML, and Pixel Bender are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

ActiveX and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/other countries. Macintosh is a trademark of Apple Inc., registered in the United States and other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

Updated Information/Additional Third Party Code Information available at www.adobe.com/go/thirdparty.

Portions include software under the following terms:

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes software developed by Fourthought, Inc. (http://www.fourthought.com).

MPEG Layer-3 audio compression technology licensed by Fraunhofer IIS and Thomson Multimedia (http://www.iis.fhg.de/amm/).

This software is based in part on the work of the Independent JPEG Group.

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com).

Video in Flash Player is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. http://www.on2.com.

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.

Sorenson Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Notice to U.S. Government End Users: The Software and Documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contenido

Capítulo 1: Introducción a ActionScript 3.0
ActionScript . . . 1
Ventajas de ActionScript 3.0 . . . 1
Novedades de ActionScript 3.0 . . . 2

Capítulo 2: Introducción a ActionScript
Fundamentos de programación . . . 5
Trabajo con objetos . . . 7
Elementos comunes de los programas . . . 15
Ejemplo: Sitio de muestras de animación (Flash Professional) . . . 17
Creación de aplicaciones con ActionScript . . . 20
Creación de clases personalizadas . . . 24
Ejemplo: Creación de una aplicación básica . . . 26

Capítulo 3: El lenguaje ActionScript y su sintaxis
Información general sobre el lenguaje . . . 34
Objetos y clases . . . 35
Paquetes y espacios de nombres . . . 35
Variables . . . 45
Tipos de datos . . . 49
Sintaxis . . . 61
Operadores . . . 66
Condicionales . . . 72
Reproducir indefinidamente . . . 74
Funciones . . . 77

Capítulo 4: Programación orientada a objetos con ActionScript
Introducción a la programación orientada a objetos . . . 89
Clases . . . 89
Interfaces . . . 104
Herencia . . . 106
Temas avanzados . . . 114
Ejemplo: GeometricShapes . . . 121

Capítulo 1: Introducción a ActionScript 3.0

ActionScript

ActionScript es el lenguaje de programación para los entornos de tiempo de ejecución de Adobe® Flash® Player y Adobe® AIR™. Entre otras muchas cosas, activa la interactividad y la gestión de datos en el contenido y las aplicaciones de Flash, Flex y AIR.

ActionScript se ejecuta mediante la máquina virtual ActionScript (AVM), que forma parte de Flash Player y AIR. El código ActionScript suele transformarse en formato de código de bytes mediante el compilador. (*Bytecode* un tipo de lenguaje que los ordenadores pueden escribir y comprender.) Entre los ejemplos de compiladores se incluyen el incorporado en Adobe® Flash® Professional, en Adobe® Flash® Builder™ y el SDK de Adobe® Flex™. El código de bytes está incorporado en los archivos SWF ejecutados por Flash Player y AIR.

ActionScript 3.0 ofrece un modelo de programación robusto que resultará familiar a los desarrolladores con conocimientos básicos sobre programación orientada a objetos. Algunas de las principales funciones de ActionScript 3.0 que mejoran las versiones anteriores son:

- Una nueva máquina virtual de ActionScript, denominada AVM2, que utiliza un nuevo conjunto de instrucciones de código de bytes y proporciona importantes mejoras de rendimiento.
- Una base de código de compilador más moderna que realiza mejores optimizaciones que las versiones anteriores del compilador.
- Una interfaz de programación de aplicaciones (API) ampliada y mejorada, con un control de bajo nivel de los objetos y un auténtico modelo orientado a objetos.
- Una API XML basada en la especificación de ECMAScript para XML (E4X) (ECMA-357 edición 2). E4X es una extensión del lenguaje ECMAScript que añade XML como un tipo de datos nativo del lenguaje.
- Un modelo de eventos basado en la especificación de eventos DOM (modelo de objetos de documento) de nivel 3.

Ventajas de ActionScript 3.0

ActionScript 3.0 aumenta las posibilidades de creación de scripts de las versiones anteriores de ActionScript. Se ha diseñado para facilitar la creación de aplicaciones muy complejas con conjuntos de datos voluminosos y bases de código reutilizables y orientadas a objetos. ActionScript 3.0 no se requiere para el contenido que se ejecuta en Adobe Flash Player. Sin embargo, permite introducir unas mejoras de rendimiento que sólo están disponibles con AVM2, (la nueva máquina virtual de ActionScript 3.0). El código ActionScript 3.0 puede ejecutarse con una velocidad diez veces mayor que el código ActionScript heredado.

La versión anterior de la máquina virtual ActionScript (AVM1) ejecuta código ActionScript 1.0 y ActionScript 2.0. Flash Player 9 y 10 admiten AVM1 por compatibilidad con versiones anteriores.

Novedades de ActionScript 3.0

Aunque ActionScript 2.0 contiene muchas clases y funciones que resultan familiares a los programadores de ActionScript 1.0 y 2.0, la arquitectura y los conceptos de ActionScript 3.0 difieren de las versiones anteriores de ActionScript. ActionScript 3.0 incluye algunas mejoras como, por ejemplo, nuevas funciones del núcleo del lenguaje y una API de mejorada que proporciona un mayor control de objetos de bajo nivel.

Funciones del núcleo del lenguaje

El núcleo del lenguaje está formado por los bloques básicos del lenguaje de programación, como sentencias, expresiones, condiciones, bucles y tipos. ActionScript 3.0 contiene diversas funciones que agilizan el proceso de desarrollo.

Excepciones de tiempo de ejecución

ActionScript 3.0 notifica más situaciones de error que las versiones anteriores de ActionScript. Las excepciones de tiempo de ejecución se utilizan en situaciones de error frecuentes y permiten mejorar la depuración y desarrollar aplicaciones para gestionar errores de forma robusta. Los errores de tiempo de ejecución pueden proporcionar trazas de pila con la información del archivo de código fuente y el número de línea. Esto permite identificar rápidamente los errores.

Tipos de tiempo de ejecución

En ActionScript 3.0, la información de tipos se conserva en tiempo de ejecución. Esta información se utiliza para realizar una verificación de tipos en tiempo de ejecución, mejorando la seguridad de los tipos del sistema. La información de tipos también se utiliza para especificar variables en representaciones nativas de la máquina, lo que mejora el rendimiento y reduce el uso de memoria. Como comparación, en ActionScript 2.0 las anotaciones de tipos eran principalmente una ayuda para el desarrollador y todos los valores se escribían dinámicamente en tiempo de ejecución.

Clases cerradas

ActionScript 3.0 incluye el concepto de clases cerradas. Una clase cerrada posee únicamente el conjunto fijo de propiedades y métodos definidos durante la compilación; no es posible añadir propiedades y métodos adicionales. La incapacidad para cambiar una clase en tiempo de ejecución permite realizar una comprobación más estricta en tiempo de compilación, lo que aporta una mayor solidez a los programas.También mejora el uso de memoria, pues no requiere una tabla hash interna para cada instancia de objeto. Además, es posible utilizar clases dinámicas mediante la palabra clave `dynamic`. Todas las clases de ActionScript 3.0 están cerradas de forma predeterminada, pero pueden declararse como dinámicas con la palabra clave `dynamic`.

Cierres de métodos

ActionScript 3.0 permite que un cierre de método recuerde automáticamente su instancia de objeto original. Esta función resulta útil en la gestión de eventos. En ActionScript 2.0, los cierres de métodos no recordaban la instancia de objeto de la que se habían extraído, lo que provocaba comportamientos inesperados cuando se llamaba al cierre de método.

ECMAScript for XML (E4X)

ActionScript 3.0 implementa ECMAScript for XML (E4X), recientemente estandarizado como ECMA-357. E4X ofrece un conjunto fluido y natural de construcciones del lenguaje para manipular XML. Al contrario que las API de análisis de XML tradicionales, XML con E4X se comporta como un tipo de datos nativo del lenguaje. E4X optimiza el desarrollo de aplicaciones que manipulan XML, pues reduce drásticamente la cantidad de código necesario.

Para ver la especificación de E4X publicada por ECMA, visite www.ecma-international.org.

Expresiones regulares

ActionScript 3.0 ofrece compatibilidad nativa con expresiones regulares, que permiten encontrar y manipular cadenas rápidamente. ActionScript 3.0 implementa la compatibilidad con expresiones regulares tal y como se definen en la especificación del lenguaje ECMAScript (ECMA-262) edición 3.

Espacios de nombres

Los espacios de nombres son similares a los especificadores de acceso tradicionales que se utilizan para controlar la visibilidad de las declaraciones (`public`, `private`, `protected`). Funcionan como especificadores de acceso personalizados, con nombres elegidos por el usuario. Los espacios de nombres incluyen un identificador de recursos universal (URI) para evitar colisiones y también se utilizan para representar espacios de nombres XML cuando se trabaja con E4X.

Nuevos tipos simples

ActionScript 3.0 contiene tres tipos numéricos: Number, int y uint. El número representa un número de coma flotante y doble precisión. El tipo int es un entero de 32 bits con signo que permite al código ActionScript aprovechar las capacidades matemáticas de manipulación rápida de enteros de la CPU. Este tipo es útil para contadores de bucle y variables en las que se usan enteros. El tipo uint es un tipo entero de 32 bits sin signo que resulta útil para valores de colores RGB y recuentos de bytes, entre otras cosas. Por el contrario, ActionScript 2.0 únicamente cuenta con un solo tipo numérico, Number.

Funciones de la API

Las API en ActionScript 3.0 contienen muchas de las clases que permiten controlar objetos a bajo nivel. La arquitectura del lenguaje está diseñada para ser mucho más intuitiva que en versiones anteriores. Aunque existen demasiadas clases para analizar detalladamente, algunas diferencias significativas no tienen relevancia.

Modelo de eventos DOM3

El modelo de eventos del modelo de objetos de documento de nivel 3 (DOM3) ofrece una forma estándar para generar y gestionar mensajes de eventos. Este modelo eventos está diseñado para que los objetos de las aplicaciones puedan interactuar y comunicarse, mantener su estado y responder a los cambios. El modelo de eventos de ActionScript 3.0 está diseñado a partir de la especificación de eventos DOM de nivel 3 del World Wide Web Consortium. Este modelo proporciona un mecanismo más claro y eficaz que los sistemas de eventos disponibles en versiones anteriores de ActionScript.

Los eventos y los eventos de error se encuentran en el paquete flash.events. Los componentes de Flash Professional y la arquitectura Flex utilizan el mismo modelo de eventos, de modo que el sistema de eventos está unificado en toda la plataforma Flash.

API de la lista de visualización

La API de acceso a la lista de visualización (el árbol que contiene todos los elementos visuales de una aplicación Flash) se compone de clases para trabajar con elementos visuales simples.

La nueva clase Sprite es un bloque básico ligero, diseñado para ser una clase base para elementos visuales como, por ejemplo, componentes de interfaz de usuario. La clase Shape representa formas vectoriales sin procesar. Es posible crear instancias de estas clases de forma natural con el operador `new` y se puede cambiar el elemento principal en cualquier momento, de forma dinámica.

La administración de profundidad es automática. Se proporcionan métodos para especificar y administrar el orden de apilación de los objetos.

Gestión de contenido y datos dinámicos

ActionScript 3.0 contiene mecanismos para cargar y gestionar elementos y datos en la aplicación, que son intuitivos y coherentes en toda la API. La clase Loader ofrece un solo mecanismo para cargar archivos SWF y elementos de imagen, y proporciona una forma de acceso a información detallada sobre el contenido cargado. La clase URLLoader proporciona un mecanismo independiente para cargar texto y datos binarios en aplicaciones basadas en datos. La clase Socket proporciona una forma de leer y escribir datos binarios en sockets de servidor en cualquier formato.

Acceso a datos de bajo nivel

Distintas API proporcionan un acceso de bajo nivel a los datos. La clase URLStream proporciona acceso a los datos como datos binarios sin formato mientras se descargan. La clase ByteArray permite optimizar la lectura, escritura y utilización de datos binarios. La API de sonido proporciona control detallado del sonido a través de las clases SoundChannel y SoundMixer. La API de seguridad proporcionan información sobre los privilegios de seguridad de un archivo SWF o contenido cargado, lo que permite gestionar los errores de seguridad.

Trabajo con texto

ActionScript 3.0 contiene un paquete flash.text para todas las API relacionadas con texto. La clase TextLineMetrics proporciona medidas detalladas para una línea de texto en un campo de texto; sustituye al método `TextFormat.getTextExtent()` en ActionScript 2.0. La clase TextField contiene nuevos métodos interesantes de bajo nivel que pueden ofrecer información específica sobre una línea de texto o un solo carácter en un campo de texto. Por ejemplo, el método `getCharBoundaries()` devuelve un rectángulo que representa el recuadro de delimitación de un carácter. El método `getCharIndexAtPoint()` devuelve el índice del carácter en un punto especificado. El método `getFirstCharInParagraph()` devuelve el índice el primer carácter de un párrafo. Los métodos de nivel de línea son: `getLineLength()`, que devuelve el número de caracteres en una línea de texto especificada, y `getLineText()`, que devuelve el texto de la línea especificada. La clase Font proporciona un medio para administrar las fuentes incorporadas en archivos SWF.

Para un control incluso de más bajo nivel sobre el texto, la clases del paquete flash.text.engine conforman Flash Text Engine. Este conjunto de clases proporcionan un control de bajo nivel sobre el texto y están diseñadas para crear componentes y marcos de texto.

Capítulo 2: Introducción a ActionScript

Fundamentos de programación

ActionScript es un lenguaje de programación, por lo que será de gran ayuda comprender primero algunos conceptos generales de programación.

Para qué sirven los programas informáticos

En primer lugar, resulta útil entender qué es un programa informático y para qué sirve. Un programa informático se caracteriza por dos aspectos principales:

- Un programa es una serie de instrucciones o pasos que debe llevar a cabo el equipo.
- Cada paso implica en última instancia la manipulación de información o datos.

En general, un programa informático es simplemente una lista de instrucciones paso a paso que se dan al equipo para que las lleve a cabo una a una. Cada una de las instrucciones se denomina *sentencia*. En ActionScript, todas las sentencias se escriben con un punto y coma al final.

Lo que realiza básicamente una instrucción dada en un programa es manipular algún bit de datos almacenado en la memoria del equipo. Un ejemplo sencillo consiste en ordenar al equipo que añada dos números y almacene el resultado en su memoria. En un caso más complejo, se podría tener un rectángulo dibujado en la pantalla y escribir un programa para moverlo a algún otro lugar de la pantalla. El equipo recuerda determinada información relativa al rectángulo: las coordenadas x e y que indican su ubicación, la anchura y altura, el color, etc. Cada uno de estos bits de información se almacena en algún lugar de la memoria del equipo. Un programa para mover el rectángulo a una ubicación diferente podría incluir pasos como “cambiar la coordenada x a 200; cambiar la coordenada y a 150”. Es decir, especificando nuevos valores para las coordenadas x e y. En segundo plano, el equipo procesa estos datos de algún modo para convertir realmente estos números en la imagen que aparece en pantalla. Sin embargo, en el nivel básico de detalle que interesa, basta con saber que el proceso de “mover un rectángulo en pantalla” únicamente implica un cambio de bits de datos en la memoria del equipo.

Variables y constantes

Principalmente la programación implica el cambio de partes de información en la memoria del equipo. Por lo tanto, resulta importante disponer de algún modo de representar una sola parte de información en un programa. Una *variable* es un nombre que representa un valor en la memoria del equipo. Cuando se escriben sentencias para manipular valores, se escribe el nombre de la variable en lugar del valor; cuando el equipo ve el nombre de la variable en el programa, busca en su memoria y utiliza el valor que allí encuentra. Por ejemplo, si hay dos variables denominadas `value1` y `value2`, cada una de las cuales contiene un número, para sumar esos dos números se puede escribir la siguiente sentencia:

```
value1 + value2
```

Cuando lleve a cabo los pasos indicados, el equipo buscará los valores de cada variable y los sumará.

En ActionScript 3.0, una variable se compone realmente de tres partes distintas:

- El nombre de la variable
- El tipo de datos que puede almacenarse en la variable
- El valor real almacenado en la memoria del equipo

Se acaba de explicar el modo en que el equipo utiliza el nombre como marcador de posición del valor. El tipo de datos también es importante. Cuando se crea una variable en ActionScript, se determina el tipo específico de datos que va a incluir. A partir de aquí, las instrucciones del programa sólo pueden almacenar ese tipo de datos en la variable. El valor se puede manipular con las características particulares asociadas a su tipo de datos. En ActionScript, para crear una variable (se conoce como *declarar* la variable), se utiliza la sentencia `var`:

```
var value1:Number;
```

Con este ejemplo se indica al equipo que cree una variable denominada `value1`, que sólo puede incluir datos Number. (“Number” es un tipo de datos específico definido en ActionScript.) También es posible almacenar un valor directamente en la variable:

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

Adobe Flash Professional

En Flash Professional existe otra forma posible de declarar una variable. Cuando se coloca un símbolo de clip de película, un símbolo de botón o un campo de texto en el escenario, se le puede asignar un nombre de instancia en el inspector de propiedades. En segundo plano, Flash Professional crea una variable con el mismo nombre que la instancia. Este nombre se puede utilizar en el código ActionScript para representar a ese elemento del escenario. Por ejemplo, supongamos que se dispone de un símbolo de clip de pelí-cula en el escenario y se le asigna el nombre de instancia `rocketShip`. Siempre que se utilice la variable `rocketShip` en el código ActionScript, en realidad se está manipulando ese clip de película.

Una *constante* es similar a una variable. Se trata de un nombre que representa un valor en la memoria del equipo con un tipo especifico de datos. La diferencia es que a una constante sólo se le puede asignar un valor cada vez en el curso de una aplicación ActionScript. Tras asignar un valor a una constante, éste permanecerá invariable en toda la aplicación. La sintaxis para declarar una constante es prácticamente la misma que para la declaración de una variable. La única diferencia radica en que se utiliza la palabra clave `const` en lugar de `var`:

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

Una constante resulta útil para definir un valor que se utilizará en varios puntos de un proyecto y que no cambiará en circunstancias normales. Cuando se utiliza una constante en lugar de un valor literal el código resulta más legible. Por ejemplo, dos versiones del mismo código. En uno se multiplica un precio por `SALES_TAX_RATE`. En el otro el precio se multiplica por `0.07`. La versión que utiliza la constante `SALES_TAX_RATE` resulta más fácil de entender. Asimismo, supongamos que cambia el valor definido por la constante. Si se utiliza una constante para representar el valor en el proyecto, el valor se puede cambiar en un lugar (la declaración de la constante). Por el contrario, tendría que modificarse en distintos lugares cuando se utilizan valores literales especificados en el código.

Tipos de datos

En ActionScript, hay muchos tipos de datos que pueden utilizarse como el tipo de datos de las variables que se crean. Algunos de estos tipos de datos se pueden considerar “sencillos” o “fundamentales”:

- String: un valor de texto como, por ejemplo, un nombre o el texto de un capítulo de un libro
- Numeric: ActionScript 3.0 incluye tres tipos de datos específicos para datos numéricos:
    - Number: cualquier valor numérico, incluidos los valores fraccionarios o no fraccionarios
    - int: un entero (un número no fraccionario)
    - uint: un entero sin signo, es decir, que no puede ser negativo

- Boolean: un valor true (verdadero) o false (falso), por ejemplo, si un conmutador está activado o si dos valores son iguales

El tipo de datos sencillo representa a un solo elemento de información: por ejemplo, un único número o una sola secuencia de texto. Sin embargo, la mayoría de los tipos de datos definidos en ActionScript son tipos de datos complejos. Representan un conjunto de valores en un solo contenedor. Por ejemplo, una variable con el tipo de datos Date representa un solo valor (un momento temporal). No obstante, ese valor de fecha se representa en forma de diferentes valores: el día, el mes, el año, las horas, los minutos, los segundos, etc., los cuales son números individuales. Generalmente una fecha se suele percibir como un solo valor que se puede tratar como tal creando una variable Date. Sin embargo, internamente el equipo lo considera un grupo de varios valores que conjuntamente definen una sola fecha.

La mayoría de los tipos de datos incorporados y los tipos de datos definidos por los programadores son complejos. Algunos de los tipos de datos complejos que podrían reconocerse son:

- MovieClip: un símbolo de clip de película
- TextField: un campo de texto dinámico o de texto de entrada
- SimpleButton: un símbolo de botón
- Date: información sobre un solo momento temporal (una fecha y hora)

Para referirse a los tipos de datos, a menudo se emplean como sinónimos las palabras clase y objeto. Una *clase* es simplemente una definición de un tipo de datos. Se trata de una especie de plantilla para todos los objetos del tipo de datos, como afirmar que "todas las variables del tipo de datos Example tienen estas características: A, B y C". Por otra parte, un *objeto* es una instancia real de una clase. Por ejemplo, una variable cuyo tipo de datos sea MovieClip se podrá describir como un objeto MovieClip. Se puede decir lo mismo con distintos enunciados:

- El tipo de datos de la variable `myVariable` es Number.
- La variable `myVariable` es una instancia de Number.
- La variable `myVariable` es un objeto Number.
- La variable `myVariable` es una instancia de la clase Number.

Trabajo con objetos

ActionScript es lo que se denomina un lenguaje de programación orientado a objetos. La programación orientada a objetos es simplemente un enfoque de la programación. Se trata de una forma de organizar el código en un programa mediante objetos.

Anteriormente el término "programa informático" se ha definido como una serie de pasos o instrucciones que lleva a cabo el equipo. Por lo tanto, conceptualmente un programa informático se puede imaginar simplemente como una larga lista de instrucciones. Sin embargo, en la programación orientada a objetos, las instrucciones del programa se dividen entre distintos objetos. El código se agrupa en segmentos de funcionalidad, de modo que los tipos de funcionalidad relacionados o los elementos de información relacionados se agrupan en un contenedor.

Adobe Flash Professional

Si ha trabajado con símbolos en Flash Professional, estará acostumbrado a trabajar con objetos. Supongamos que se ha definido un símbolo de clip de película (por ejemplo, el dibujo de un rectángulo) y se ha colocado una copia del mismo en el escenario. Dicho símbolo de clip de película también es (literalmente) un objeto en ActionScript; es una instancia de la clase MovieClip.

Es posible modificar algunas de las características del clip de película. Cuando se selecciona, se pueden cambiar valores en el inspector de propiedades como su coordenada x o su anchura. También se puede realizar distintos ajustes de color como, por ejemplo, cambiar su valor alfa (transparencia) o aplicarle un filtro de sombra. Otras herramientas de Flash Professional permiten realizar más cambios, como utilizar la herramienta Transformación libre para girar el rectángulo. Todas estas formas de modificación de un símbolo de clip de película en Flash Professional también están disponibles en ActionScript. En ActionScript el clip de película se modifica cambiando los elementos de datos que se agrupan en un único paquete denominado objeto MovieClip.

En la programación orientada a objetos de ActionScript, hay tres tipos de características que puede contener cualquier clase:

- Propiedades
- Métodos
- Eventos

Estos elementos se utilizan para administrar los elementos de datos que utiliza el programa y para decidir qué acciones deben llevarse a cabo y en qué orden.

Propiedades

Una propiedad representa uno de los elementos de datos que se empaquetan en un objeto. Por ejemplo, un objeto Song (canción) puede tener propiedades denominadas `artist` (artista) y `title` (título); la clase MovieClip tiene propiedades como `rotation` (rotación), `x`, `width` (anchura) y `alpha` (alfa). Con las propiedades se trabaja del mismo modo que con las variables individuales. De hecho, las propiedades se puede considerar simplemente como las variables “secundarias” contenidas en un objeto.

A continuación se muestran algunos ejemplos de código ActionScript que utiliza propiedades. Esta línea de código mueve el objeto MovieClip denominado `square` a la coordenada x = 100 píxeles:

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

Este código utiliza la propiedad rotation para que el MovieClip `square` gire de forma correspondiente a la rotación del MovieClip `triangle`:

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

Este código altera la escala horizontal del MovieClip `square` para hacerlo 1,5 veces más ancho:

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

Fíjese en la estructura común: se utiliza una variable (`square`, `triangle`) como nombre del objeto, seguida de un punto (`.`) y, a continuación, el nombre de la propiedad (`x`, `rotation`, `scaleX`). El punto, denominado *operador de punto*, se utiliza para indicar el acceso a uno de los elementos secundarios de un objeto. El conjunto de la estructura (nombre de variable-punto-nombre de propiedad) se utiliza como una sola variable, como un nombre de un solo valor en la memoria del equipo.

Métodos

Un *método* es una acción que puede realizar un objeto. Por ejemplo, supongamos que se ha creado un símbolo de clip de pelí-cula en Flash con varios fotogramas clave y animación en la línea de tiempo. El clip de pelí-cula puede reproducirse, detenerse o recibir instrucciones para mover la cabeza lectora a un determinado fotograma.

Este código indica al objeto MovieClip denominado `shortFilm` que inicie su reproducción:

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

Esta línea hace que el MovieClip denominado `shortFilm` deje de reproducirse (la cabeza lectora se detiene como si se hiciera una pausa en un vídeo):

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

Este código hace que un MovieClip denominado `shortFilm` mueva su cabeza lectora al fotograma 1 y deje de reproducirse (como si se rebobinara un vídeo):

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

Para acceder a los métodos se debe escribir el nombre del objeto (una variable), un punto y el nombre del método seguido de un paréntesis, siguiendo la misma estructura que para las propiedades. El paréntesis es una forma de indicar que se está *llamando* al método, es decir, indicando al objeto que realice esa acción. Algunos valores (o variables) se incluyen dentro del paréntesis para pasar información adicional necesaria para llevar a cabo la acción. Estos valores se denominan *parámetros* del método. Por ejemplo, el método `gotoAndStop()` necesita saber cuál es el fotograma al que debe dirigirse, de modo que requiere un solo parámetro en el paréntesis. Otros métodos como `play()` y `stop()` no requieren información adicional porque son descriptivos por sí mismos. Sin embargo, también se escriben con paréntesis.

A diferencia de las propiedades (y las variables), los métodos no se usan como identificadores de valores. No obstante, algunos métodos pueden realizar cálculos y devolver un resultado que puede usarse como una variable. Por ejemplo, el método `toString()` de la clase Number convierte el valor numérico en su representación de texto:

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

Por ejemplo, se usaría el método `toString()` para mostrar el valor de una variable Number en un campo de texto de la pantalla. La propiedad `text` de la clase TextField se define como String, por lo que sólo puede contener valores de texto. (La `propiedad` de texto representa el contenido de texto real que se muestra en pantalla). Esta línea de código convierte el valor numérico de la variable `numericData` a texto. Por lo tanto, hace que el valor se muestre en pantalla en el objeto TextField denominado `calculatorDisplay`:

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

Eventos

Un programa informático consta de una serie de instrucciones que el ordenador lleva a cabo paso a paso. Algunos programas informáticos sencillos no son más que eso: unos cuantos pasos que el ordenador ejecuta, tras los cuales finaliza el programa. Sin embargo, los programas de ActionScript se han diseñado para continuar ejecutándose, esperando los datos introducidos por el usuario u otras acciones. Los eventos son los mecanismos que determinan qué instrucciones lleva a cabo el ordenador y cuándo las realiza.

Básicamente, los *eventos* son acciones que ActionScript conoce y a las que puede responder. Muchos eventos se relacionan con la interacción del usuario como, por ejemplo, hacer clic en un botón o presionar una tecla del teclado. También existen otros tipos de eventos. Por ejemplo, si se usa ActionScript para cargar una imagen externa, existe un evento que puede indicar al usuario cuándo finaliza la carga de la imagen. Cuando se está ejecutando un programa de ActionScript, conceptualmente éste espera a que ocurran determinadas acciones. Cuando suceden se ejecuta el código ActionScript que se haya especificado para tales eventos.

Gestión básica de eventos

La técnica para especificar determinadas acciones que deben realizarse como respuesta a eventos concretos se denomina *gestión de eventos*. Cuando se escribe código ActionScript para llevar a cabo la gestión de eventos, se deben identificar tres elementos importantes:

- El origen del evento: ¿en qué objeto va a repercutir el evento? Por ejemplo, ¿en qué botón se hará clic o qué objeto Loader está cargando la imagen? El origen del evento también se denomina *objetivo del evento*. Este nombre se debe a que es el objeto al que el ordenador destina el evento (es decir, el lugar donde el evento tiene lugar realmente).
- El evento: ¿qué va a suceder, a qué se va a responder? Es importante identificar el evento específico, ya que muchos objetos activan varios eventos.
- La respuesta: ¿qué pasos hay que llevar a cabo cuando ocurra el evento?

Siempre que se escriba código ActionScript para gestionar eventos, el código debe incluir estos tres elementos. El código debe seguir esta estructura básica (los elementos en negrita son marcadores de posición que se deben completar en cada caso concreto).

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

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

Este código realiza dos operaciones. En primer lugar, define una función, que es la forma de especificar las acciones que desean realizarse como respuesta al evento. Posteriormente, llama al método `addEventListener()` del objeto origen. Al llamar a `addEventListener()` básicamente se "suscribe" la función al evento especificado. Cuando ocurra el evento, se llevan a cabo las acciones de la función. Cada una de estas partes se tratará con mayor detalle.

Una *función* proporciona un modo de agrupar acciones con un único nombre que viene a ser un nombre de método abreviado para llevar a cabo las acciones. Una función es idéntica a un método, excepto en que no está necesariamente asociada con una clase específica. (De hecho, el término "método" se puede definir como una función asociada a una clase determinada). Cuando se crea una función para la gestión de eventos, se debe elegir el nombre de la función (denominada `eventResponse` en este caso). También se especifica un parámetro (llamado `eventObject` en este ejemplo). Especificar un parámetro de una función equivale a declarar una variable, de modo que también hay que indicar el tipo de datos del parámetro. (En este ejemplo, el tipo de datos del parámetro es `EventType`.)

Cada tipo de evento que se desee detectar tiene asociada una clase de ActionScript. El tipo de datos especificado para el parámetro de función es siempre la clase asociada del evento concreto al que se desea responder. Por ejemplo, un evento `click` (el cual se activa al hacer clic en un elemento con el ratón) se asocia a la clase MouseEvent. Cuando se vaya a escribir una función de detector para un evento `click`, ésta se debe definir con un parámetro con el tipo de datos MouseEvent. Por último, entre los paréntesis de apertura y cierre (`{` ... `}`), se escriben las instrucciones que debe llevar a cabo el equipo cuando ocurra el evento.

Se escribe la función de gestión de eventos. Posteriormente se indica al objeto de origen del evento (el objeto en el que se produce el evento, por ejemplo, el botón) que se desea llamar a la función cuando ocurra el evento. La función se registra con el objeto de origen del evento llamando al método `addEventListener()` de ese objeto (todos los objetos que tienen eventos también disponen de un método `addEventListener()`). El método `addEventListener()` utiliza dos parámetros:

- En primer lugar, el nombre del evento específico al que se desea responder. Cada evento se asocia a una clase específica. Todas las clases de evento tienen, a su vez, un valor especial (como un nombre exclusivo) definido para cada uno de sus eventos.Este valor se utiliza para el primer parámetro.

- En segundo lugar, el nombre de la función de respuesta al evento. Hay que tener en cuenta que el nombre de una función debe escribirse sin paréntesis cuando se pasa como un parámetro.

El proceso de control de eventos

A continuación se ofrece una descripción paso a paso del proceso que tiene lugar cuando se crea un detector de eventos. En este caso, es un ejemplo de creación de función de detector a la que se llama cuando se hace clic en un objeto denominado `myButton`.

El código escrito por el programador es el siguiente:

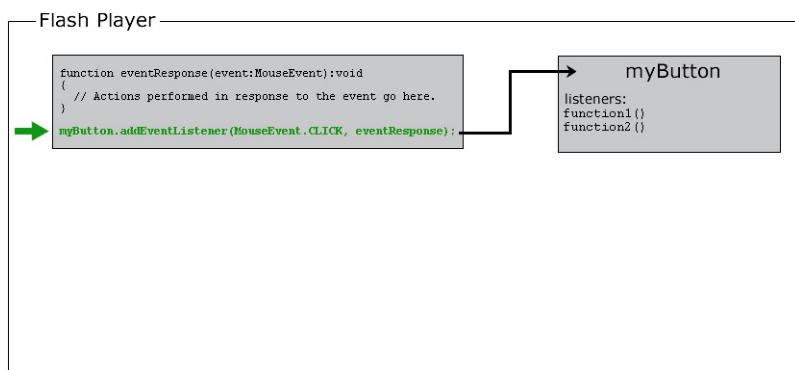
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

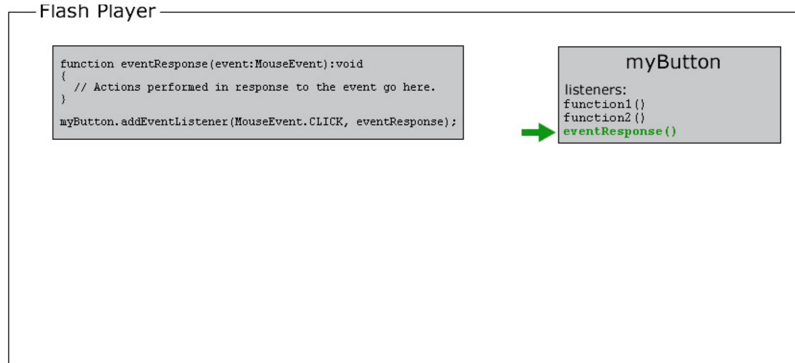
Al ejecutarse, el código funcionaría de la manera siguiente:

1 Cuando se carga el archivo SWF, el equipo detecta que existe una función denominada `eventResponse()`.

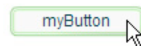
2 A continuación, el equipo ejecuta el código (concretamente las líneas de código que no están en una función). En este caso se trata de una sola línea de código: para llamar al método `addEventListener()` en el objeto de origen del evento (denominado `myButton`) y pasar la función `eventResponse` como parámetro.



Internamente, `myButton` conserva una lista de funciones que detectan cada uno de sus eventos. Cuando se llama a su método `addEventListener()`, `myButton` almacena la función `eventResponse()` en su lista de detectores de eventos.

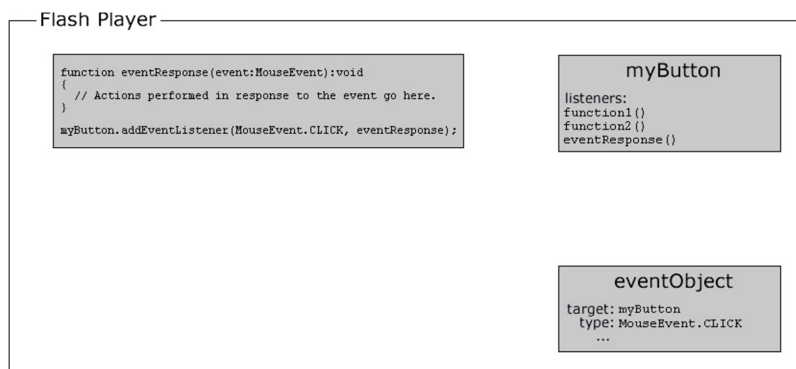


3 Cuando el usuario hace clic en el objeto `myButton`, se activa el evento `click` (identificado como `MouseEvent.CLICK` en el código).

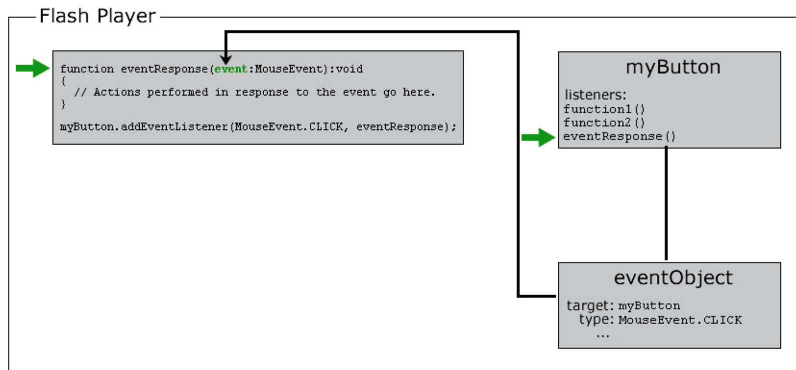


En este punto ocurre lo siguiente:

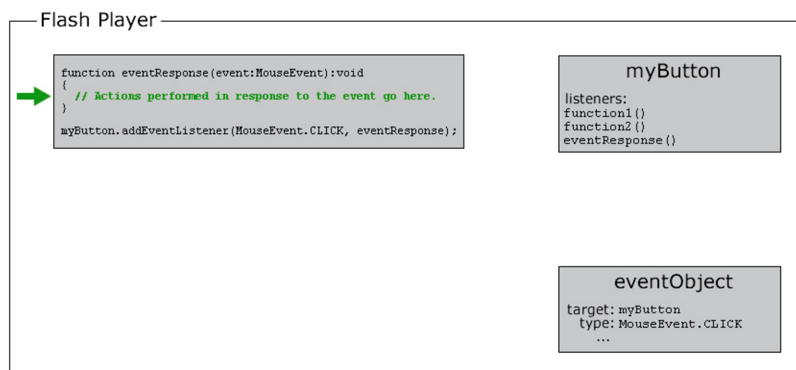
a Se crea un objeto que es una instancia de la clase asociada con el evento en cuestión (MouseEvent en este ejemplo). Para diversos eventos, este objeto es una instancia de la clase Event. Para los eventos de ratón es una instancia de MouseEvent. Para otros eventos, se trata de una instancia de la clase que está asociada con ese evento. Este objeto creado se conoce como el *objeto de evento* y contiene información específica sobre el evento que se ha producido: el tipo de evento, el momento en que ha ocurrido y otros datos relacionados con el evento, si procede.



b A continuación, el equipo busca en la lista de detectores de eventos almacenada en `myButton`. Recorre estas funciones de una en una, llamando a cada función y pasando el objeto de evento a la función como parámetro. Como la función `eventResponse()` es uno de los detectores de `myButton`, como parte de este proceso el equipo llama a la función `eventResponse()`.



c Cuando se llama a la función `eventResponse()`, se ejecuta el código de la función para realizar las acciones especificadas.



Ejemplos de gestión de eventos

A continuación se incluyen algunos ejemplos más concretos de código de gestión de eventos. Con estos ejemplos se pretende proporcionar una idea de algunos de los elementos comunes de los eventos y de las posibles variaciones disponibles cuando se escribe código de gestión de eventos:

- Hacer clic en un botón para iniciar la reproducción del clip de película actual. En el siguiente ejemplo, `playButton` es el nombre de instancia del botón y `this` es el nombre especial, que significa “el objeto actual”:

```
this.stop();

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

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

- Detectar si se ha escrito algo en un campo de texto. En este ejemplo, `entryText` es un campo de introducción de texto y `outputText` es un campo de texto dinámico:

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

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

- Hacer clic en un botón para navegar a un URL. En este caso, `linkButton` es el nombre de instancia del botón:

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

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

Creación de instancias de objetos

Antes de poder utilizar un objeto en ActionScript, éste debe existir. Una parte de la creación de un objeto es la declaración de una variable; sin embargo, declarar una variable sólo crea un espacio vacío en la memoria del equipo. Es necesario asignar siempre un valor real a la variable, es decir, crear un objeto y almacenarlo en la variable, antes de intentar usarla o manipularla. El proceso de creación de un objeto se denomina *creación de una instancia* del objeto. Es decir, se crea una instancia de una clase concreta.

Hay una forma sencilla de crear una instancia de objeto en la que no se utiliza ActionScript en absoluto. En Flash Professional sitúe un símbolo de clip de película, símbolo de botón o campo de texto en el escenario y asígnele un nombre de instancia. Flash Professional declara automáticamente una variable con ese nombre de instancia, crea una instancia de objeto y almacena ese objeto en la variable. Del mismo modo, en Flex se crea un componente en MXML, codificando una etiqueta MXML o situando el componente en el editor en modo de diseño de Flash Builder. Cuando se asigna un ID a ese componente, este ID se convierte en el nombre de una variable de ActionScript que contiene la instancia de ese componente.

Sin embargo, no siempre se desea crear un objeto visualmente y para los objetos no visuales no se puede realizar esta operación. Existen varias formas adicionales de crear instancias de objetos utilizando exclusivamente ActionScript.

Con varios tipos de datos de ActionScript es posible crear una instancia con una *expresión literal*, es decir, un valor escrito directamente en el código ActionScript. A continuación se muestran algunos ejemplos:

- Valor numérico literal (introducir el número directamente):

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

- Valor de cadena literal (poner el texto entre comillas dobles):

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

- Valor booleano literal (usar los valores literales `true` o `false`):

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

- Valor de conjunto literal (cerrar entre corchetes una lista de valores separados por coma):

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

- Valor XML literal (introducir los datos XML directamente):

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

ActionScript también define expresiones literales para los tipos de datos Array, RegExp, Object y Function.

La forma más común de crear una instancia para cualquier tipo de datos consiste en utilizar el operador `new` con el nombre de clase, tal y como se muestra a continuación:

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

La creación de un objeto con el operador `new` se suele describir como “llamar al constructor de la clase”. Un *constructor* es un método especial al que se llama como parte del proceso de creación de una instancia de una clase. Se debe tener en cuenta que, cuando se crea una instancia de este modo, se incluyen paréntesis después del nombre de la clase. En ocasiones se especifican los valores del parámetro en los paréntesis. Existen dos aspectos que también se llevan a cabo al llamar a un método.

Incluso en los tipos de datos que permiten crear instancias con una expresión literal, se puede utilizar el operador `new` para crear una instancia de objeto. Por ejemplo, las siguientes dos líneas de código realizan lo mismo:

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

Es importante familiarizarse con la creación de objetos mediante `new` *ClassName*`()`. Muchos tipos de datos de ActionScript no disponen de una representación visual. Por lo tanto, no se pueden crear situando un elemento en el escenario de Flash Professional o en el modo de diseño del editor MXML de Flash Builder. Únicamente se puede crear una instancia de cualquiera de estos tipos de datos en ActionScript utilizando el operador `new`.

Adobe Flash Professional
En Flash Professional, el operador `new` también se puede usar para crear una instancia de un símbolo de clip de película que esté definido en la biblioteca pero no esté colocado en el escenario.

Más temas de ayuda

Trabajo con conjuntos

Uso de expresiones regulares

Creación de objetos MovieClip con ActionScript

Elementos comunes de los programas

Existen algunos componentes esenciales adicionales que se utilizan para crear un programa de ActionScript.

Operadores

Los *operadores* son símbolos especiales (o, en ocasiones, palabras) que se utilizan para realizar cálculos. Se utilizan principalmente en las operaciones matemáticas, pero también en la comparación entre valores. Por lo general, un operador utiliza uno o varios valores y calcula un solo resultado. Por ejemplo:

- El operador de suma (+) suma dos valores y obtiene como resultado una sola cifra:

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

- El operador de multiplicación (`*`) multiplica un valor por otro y obtiene como resultado una sola cifra:

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

- El operador de igualdad (`==`) compara dos valores para ver si son iguales y obtiene como resultado un solo valor booleano (true o false):

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

  Tal y como se muestra aquí, el operador de igualdad y los otros operadores de comparación se suelen utilizar con la sentencia `if` para determinar si determinadas instrucciones deben llevarse a cabo o no.

Comentarios

Mientras se escribe código ActionScript, a menudo el propio programador desea realizar anotaciones. Por ejemplo, para explicar el funcionamiento de algunas lí-neas de código o el motivo de una determinada elección. Los *comentarios del código* son una herramienta que permite escribir en el código texto que el ordenador debe ignorar. ActionScript incluye dos tipos de comentarios:

- Comentario de una sola línea: un comentario de una sola línea se indica mediante dos barras diagonales en cualquier lugar de una línea. El ordenador ignora todo lo que se incluya tras las barras inclinadas hasta el final de esa línea:

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

- Comentario multilínea: un comentario multilínea contiene un marcador de inicio del comentario (`/*`), el contenido del comentario y un marcador de fin del comentario (`*/`). Todo el texto incluido entre los marcadores de inicio y de fin será omitido por el ordenador, independientemente del número de lí-neas que ocupe el comentario:

```
/*
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.
*/
```

Los comentarios también se utilizan para “desactivar” temporalmente una o varias lí-neas de código. Por ejemplo, los comentarios se pueden usar si se está probando una forma distinta de llevar a cabo alguna operación. También se pueden emplear para intentar saber por qué determinado código ActionScript no funciona del modo previsto.

Control de flujo

En un programa, muchas veces se desea repetir determinadas acciones, realizar sólo algunas acciones y no otras, o realizar acciones alternativas en función de determinadas condiciones, etc. El *control de flujo* es el control sobre el cual se llevan a cabo las funciones. Hay varios tipos de elementos de control de flujo disponibles en ActionScript.

- Funciones: las funciones son como los métodos abreviados. Proporcionan un modo de agrupar una serie de acciones bajo un solo nombre y pueden utilizarse para realizar cálculos. Las funciones resultan necesarias en la gestión de eventos, pero también se utilizan como una herramienta general para agrupar una serie de instrucciones.
- Bucles: las estructuras de bucle permiten designar una serie de instrucciones que el equipo realizará un número definido de veces o hasta que cambie alguna condición. A menudo los bucles se utilizan para manipular varios elementos relacionados, mediante una variable cuyo valor cambia cada vez que el ordenador recorre el bucle.
- Sentencias condicionales: las sentencias condicionales proporcionan un modo de designar determinadas instrucciones que sólo se llevan a cabo en circunstancias concretas. También se utilizan para ofrecer conjuntos alternativos de instrucciones para condiciones distintas. El tipo más común de sentencia condicional es la sentencia `if`. La sentencia `if` comprueba un valor o una expresión escrita entre paréntesis. Si el valor es `true,` se ejecutan las líneas de código entre llaves. De lo contrario, se omiten. Por ejemplo:

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

  La pareja de la sentencia `if`, la sentencia `else`, permite designar instrucciones alternativas que lleva a cabo el equipo si la condición no es `true`:

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

Ejemplo: Sitio de muestras de animación (Flash Professional)

Este ejemplo se ha diseñado para ofrecer una primera oportunidad de consultar cómo se pueden juntar fragmentos de ActionScript para crear una aplicación completa. El sitio de muestras de animación es un ejemplo de cómo se puede partir de una animación lineal existente y añadir algunos elementos interactivos secundarios. Por ejemplo, se puede incorporar una animación creada para un cliente en un sitio de muestras en línea. El comportamiento interactivo que añadiremos a la animación incluirá dos botones en los que el espectador podrá hacer clic: uno para iniciar la animación y otro para navegar a un URL diferente (como el menú del sitio o la página principal del autor).

El proceso de crear este trabajo puede dividirse en las siguientes partes principales:

1. Preparar el archivo FLA para añadir código ActionScript y elementos interactivos.
2. Crear y añadir los botones.
3. Escribir el código ActionScript.
4. Probar la aplicación.

Preparación de la animación para añadirle interactividad

Para poder añadir elementos interactivos a la animación, es útil configurar el archivo FLA creando algunos lugares para añadir el contenido nuevo. Esta tarea incluye la creación del espacio real en el escenario en el que se colocarán los botones. También implica la creación de "espacio" en el archivo FLA para mantener separados los distintos elementos.

Para preparar el archivo FLA para añadir elementos interactivos:

1. Cree un archivo FLA con una animación sencilla, como una sola interpolación de movimiento o de forma. Si ya se dispone de un archivo FLA que contiene la animación que se va a exhibir en el proyecto, abra este archivo y guárdelo con un nuevo nombre.
2. Decida el lugar en pantalla donde desea que aparezcan los dos botones: uno para iniciar la animación y otro vinculado al sitio de muestras o a la página principal del autor. Si es necesario, borre o añada espacio en el escenario para el nuevo contenido. Si la animación no dispone de ninguna, puede crear una pantalla de inicio en el primer fotograma. Probablemente tenga que desplazar la animación de forma que empiece en el fotograma 2 o en un fotograma posterior.
3. Añada una nueva capa sobre las demás capas de la línea de tiempo y asígnele el nombre **buttons**. En esta capa se añadirán los botones.
4. Añada otra capa sobre la capa buttons y asígnele el nombre **actions**. Ésta será la capa en la que añadirá el código ActionScript para la aplicación.

Creación y adición de botones

A continuación se deben crear y colocar los botones que forman el centro de la aplicación interactiva.

Para crear y añadir botones al archivo FLA:

1. Utilice las herramientas de dibujo para crear el aspecto visual del primer botón (el botón "play") en la capa buttons. Por ejemplo, puede dibujar un óvalo horizontal con texto encima.
2. Con la herramienta Selección, seleccione todos los elementos gráficos del botón individual.
3. En el menú principal, elija Modificar > Convertir en símbolo.
4. En el cuadro de diálogo, elija Botón como tipo de símbolo, asigne un nombre al símbolo y haga clic en Aceptar.
5. Con el botón seleccionado, asigne al botón el nombre de instancia **playButton** en el inspector de propiedades.
6. Repita los pasos 1 a 5 para crear el botón que llevará al usuario a la página principal del autor. Asigne a este botón el nombre **homeButton**.

Escritura del código

El código ActionScript para esta aplicación puede dividirse en tres grupos de funcionalidad, aunque se escribirá todo en el mismo lugar. El código realiza tres operaciones:

- Detener la cabeza lectora en cuanto se cargue el archivo SWF (cuando la cabeza lectora llegue al Fotograma 1).
- Detectar un evento para iniciar la reproducción del archivo SWF cuando el usuario haga clic en el botón de reproducción.
- Detectar un evento para enviar el navegador al URL apropiado cuando el usuario haga clic en el botón vinculado a la página de inicio del autor.

Para crear el código necesario para detener la cabeza lectora cuando llegue al Fotograma 1:

1. Seleccione el fotograma clave en el Fotograma 1 de la capa actions.

2 Para abrir el panel Acciones, en el menú principal, elija Ventana > Acciones.

3 En el panel Script, escriba el código siguiente:

```
stop();
```

Para escribir código para iniciar la animación cuando se haga clic en el botón play:

1 Al final del código escrito en los pasos anteriores, añada dos líneas vacías.

2 Escriba el código siguiente al final del script:

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

Este código define una función denominada `startMovie()`. Cuando se llama a `startMovie()`, hace que se inicie la reproducción de la línea de tiempo principal.

3 En la línea que sigue al código añadido en el paso anterior, escriba esta línea de código:

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

Esta línea de código registra la función `startMovie()` como un detector del evento `click` de `playButton`. Es decir, hace que siempre que se haga clic en el botón `playButton`, se llame a la función `startMovie()`.

Para escribir código que envíe el navegador a una dirección URL cuando se haga clic en el botón vinculado a la página principal:

1 Al final del código escrito en los pasos anteriores, añada dos líneas vacías.

2 Escriba el código siguiente al final del script:

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

Este código define una función denominada `gotoAuthorPage()`. En primer lugar esta función crea una instancia de URLRequest que representa la URL http://example.com/. A continuación pasa la URL a la función `navigateToURL()`, lo que hace que el navegador del usuario abra dicha URL.

3 En la línea que sigue al código añadido en el paso anterior, escriba esta línea de código:

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

Esta línea de código registra la función `gotoAuthorPage()` como un detector del evento `click` de `homeButton`. Es decir, hace que siempre que se haga clic en el botón `homeButton`, se llame a la función `gotoAuthorPage()`.

Probar la aplicación

En este momento la aplicación es completamente funcional. Pruébela.

Para probar la aplicación:

1 En el menú principal, elija Control > Probar película. Flash Professional crea el archivo SWF y lo abre en una ventana de Flash Player.

2 Pruebe ambos botones para asegurarse de que funcionan correctamente.

3 Si los botones no funcionan, puede comprobar lo siguiente:

- ¿Tienen los botones nombres de instancia distintos?
- ¿Las llamadas al método `addEventListener()` utilizan los mismos nombres que los nombres de instancia de los botones?
- ¿Se utilizan los nombres de evento correctos en las llamadas al método `addEventListener()`?
- ¿Se ha especificado el parámetro correcto para cada una de las funciones? (Ambos métodos deben tener un solo parámetro con el tipo de datos MouseEvent.)

Todos estos errores y otros posibles generan un mensaje de error. Este mensaje puede aparecer al seleccionar el comando Probar película o al hacer clic en el botón mientras se prueba el proyecto. Vea si hay errores de compilador en el panel Errores del compilador (los que se producen al seleccionar Probar pelí-cula). Compruebe el panel Salida para ver los errores en tiempo de ejecución que se producen durante la reproducción de contenido, como al hacer clic en un botón.

Creación de aplicaciones con ActionScript

El proceso de escritura de ActionScript para crear una aplicación implica algo más que el simple conocimiento de la sintaxis y los nombres de las clases que se van a utilizar. La mayor parte de la documentación de la plataforma Flash trata estos dos temas (sintaxis y uso de las clases de ActionScript). No obstante, para crear una aplicación ActionScript también se debe conocer la siguiente información:

- ¿Qué programas se pueden utilizar para escribir código ActionScript?
- ¿Cómo se organiza el código ActionScript?
- ¿Cómo se incluye el código ActionScript en una aplicación?
- ¿Qué pasos deben seguirse a la hora de desarrollar una aplicación ActionScript?

Opciones para organizar el código

Se puede utilizar código ActionScript 3.0 para crear desde sencillas animaciones gráficas hasta complejos sistemas de procesamiento de transacciones cliente-servidor. Dependiendo del tipo de aplicación que se cree, utilice una o varias de las siguientes formas posibles de incluir código ActionScript en un proyecto.

Almacenamiento de código en fotogramas de una línea de tiempo de Flash Professional

En el entorno de edición de Flash Professional, es posible añadir código ActionScript a cualquier fotograma de una línea de tiempo. Este código se ejecutará mientras se reproduce la película, cuando la cabeza lectora alcance dicho fotograma.

La colocación del código ActionScript en fotogramas es una forma sencilla de añadir comportamientos a las aplicaciones incorporadas en la herramienta de edición de Flash Professional. Se puede añadir código a cualquier fotograma de la línea de tiempo principal o a cualquier fotograma de la línea de tiempo de cualquier símbolo MovieClip. No obstante, esta flexibilidad tiene un coste. Cuando se crean aplicaciones de mayor tamaño, es fácil perder el rastro de los scripts contenidos en cada fotograma. Con el tiempo, esta complicada estructura puede dificultar el mantenimiento de la aplicación.

Muchos desarrolladores, para simplificar la organización de su código ActionScript en el entorno de edición de Flash Professional, incluyen código únicamente en el primer fotograma de una línea de tiempo o en una capa específica del documento de Flash. La separación del código facilita la localización y el mantenimiento del código en los archivos FLA de Flash. Sin embargo, el mismo código no puede utilizarse en otro proyecto de Flash Professional sin copiar y pegar el código en el nuevo archivo.

Para facilitar el uso del código ActionScript en otros proyectos de Flash Professional en el futuro, almacene el código en archivos de ActionScript externos (archivos de texto con la extensión .as).

Incorporación de código en archivos MXML de Flex

En un entorno de desarrollo de Flex como Flash Builder, el código ActionScript se puede incluir dentro de una etiqueta `<fx:Script>` en un archivo MXML de Flex. No obstante, esta técnica puede añadir complejidad a proyectos de gran tamaño y dificultar el uso del mismo código en otro proyecto de Flex. Para facilitar la utilización del código ActionScript en otros proyectos de Flex en el futuro, almacene el código en archivos de ActionScript externos.

Nota: *se puede especificar un parámetro de origen para una etiqueta* `<fx:Script>`*. Con el uso de un parámetro de origen se podrá "importar" código ActionScript de un archivo externo como si se estuviera escribiendo directamente en la etiqueta* `<fx:Script>`*. Sin embargo, el archivo de código fuente que se utiliza no puede definir su propia clase, lo cual limita su reutilización.*

Almacenamiento de código en archivos de ActionScript

Si el proyecto contiene una cantidad importante de código ActionScript, la mejor forma de organizar el código es en archivos de código fuente ActionScript independientes (archivos de texto con la extensión .as). Un archivo de ActionScript puede estructurarse de una o dos formas, dependiendo del uso que se le quiera dar en la aplicación.

- Código ActionScript no estructurado: líneas de código ActionScript, incluidas sentencias o definiciones de funciones, escritas como si se introdujeran directamente en un script de la línea de tiempo o un archivo MXML.

  Para acceder al código ActionScript escrito de este modo, es preciso utilizar la sentencia `include` en ActionScript o la etiqueta `<fx:Script>` en MXML de Flex. La sentencia `include` de ActionScript indica al equipo que incluya el contenido de un archivo de ActionScript externo en una ubicación específica y en un ámbito concreto de un script. El resultado es el mismo que si el código se incluyese allí directamente. En el lenguaje MXML, el uso de una etiqueta `<fx:Script>`con un atributo de origen identifica un archivo de ActionScript externo que carga el compilador en ese punto de la aplicación. Por ejemplo, la siguiente etiqueta carga un archivo de ActionScript externo denominado Box.as:

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

- Definición de clase de ActionScript: definición de una clase de ActionScript, incluidas sus definiciones de métodos y propiedades.

  Cuando se define una clase, se puede acceder a su código ActionScript creando una instancia de la clase y usando sus propiedades, métodos y eventos. La utilización de clases propias es un proceso idéntico al uso de cualquier clase de ActionScript incorporada y requiere dos partes:

  - Utilizar la sentencia `import` para especificar el nombre completo de la clase, de modo que el compilador de ActionScript sepa dónde encontrarlo. Por ejemplo, para utilizar la clase MovieClip en ActionScript, impórtela utilizando su nombre concreto, incluyendo paquete y clase:

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

    Como alternativa, se puede importar el paquete que contiene la clase MovieClip, que equivale a escribir sentencias `import` independientes para cada clase del paquete:

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

Las clases de nivel superior son la única excepción de la regla que indica que una clase se debe importar para utilizarse en el código. Estas clases no se definen en un paquete.

- Escriba código que utilice específicamente el nombre de la clase. Por ejemplo, declare una variable con esa clase como su tipo de datos y cree una instancia de la clase para almacenar en la variable. Al utilizar una clase en el código ActionScript, se indica al compilador que cargue la definición de dicha clase. Si se toma como ejemplo una clase externa denominada Box, esta sentencia crea una instancia de la clase Box:

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

La primera vez que el compilador se encuentra con la referencia a la clase Box, busca el código fuente disponible para localizar la definición de la clase Box.

Selección de la herramienta adecuada

Puede utilizar una de las distintas herramientas (o varias herramientas de forma conjunta) para escribir y editar código ActionScript.

Flash Builder

Adobe Flash Builder es la principal herramienta para crear proyectos con la arquitectura Flex o proyectos que constan principalmente de código ActionScript. Flash Builder también incluye un editor de ActionScript completo, así como herramientas de edición de MXML y diseño visual. Se puede emplear para crear proyectos de Flex o sólo de ActionScript. Flex ofrece varias ventajas, como un amplio conjunto de controles de interfaz de usuario predefinidos, controles de diseño dinámicos y flexibles y mecanismos incorporados para trabajar con datos remotos y vincular datos externos a elementos de interfaz de usuario. Sin embargo, debido al código adicional necesario para proporcionar estas funciones, los proyectos que utilizan Flex pueden tener un tamaño de archivo SWF mayor que sus equivalentes que no pertenecen a Flex.

El uso de Flash Builder se recomienda para crear aplicaciones de Internet completas basadas en datos con Flex. Utilice la aplicación para editar código ActionScript, editar código MXML y diseñar aplicaciones visualmente con una sola herramienta.

Muchos usuarios de Flash Professional que crean proyectos completos con ActionScript utilizan Flash Professional para crear activos visuales y Flash Builder como editor para código ActionScript.

Flash Professional

Además de las capacidades de creación de animación y gráficos, Flash Professional incluye herramientas para trabajar con código ActionScript. El código puede asociarse a los elementos de un archivo FLA o de archivos externos que sólo contienen código ActionScript. Flash Professional resulta ideal para los proyectos que contienen una cantidad importante de animación o ví-deo. Es idóneo cuando el propio usuario desea crear la mayor parte de los activos gráficos. Otro motivo para el uso de Flash Professional en el desarrollo de proyectos de ActionScript es el de poder crear activos visuales y escribir código en la misma aplicación. Flash Professional también incluye componentes de interfaz de usuario creados previamente. Estos componentes se pueden emplear para reducir el tamaño de los archivos SWF y se pueden utilizar herramientas visuales para adaptarlas al proyecto.

Flash Professional incluye dos herramientas para escribir código ActionScript:

- Panel Acciones: este panel, disponible cuando se trabaja en un archivo FLA, permite escribir código ActionScript asociado a los fotogramas de una línea de tiempo.
- Ventana Script: la ventana Script es un editor de texto dedicado para trabajar con archivos de código ActionScript (.as).

Editor de ActionScript de terceros

Debido a que los archivos ActionScript (.as) se almacenan como archivos de texto sencillo, cualquier programa que sea capaz de editar archivos de texto simple se puede usar para escribir archivos ActionScript. Además de los productos ActionScript de Adobe, se han creado varios programas de edición de texto de terceros con funciones específicas de ActionScript. Se pueden escribir archivos MXML o clases de ActionScript con cualquier programa editor de texto. Es posible crear una aplicación a partir de estos archivos utilizando el SDK de Flex. El proyecto puede utilizar Flex o ser una aplicación de sólo ActionScript. De forma alternativa, algunos desarrolladores utilizan Flash Builder o un editor de ActionScript de terceros para escribir clases de ActionScript, junto con Flash Professional para crear contenido gráfico.

Entre los motivos para seleccionar un editor de ActionScript de terceros se incluyen:

- Se prefiere escribir código ActionScript en un programa independiente y diseñar elementos visuales en Flash Professional.
- Utiliza una aplicación para programar con un lenguaje distinto de ActionScript (por ejemplo, para crear páginas HTML o aplicaciones en otro lenguaje de programación) y desea usar la misma aplicación para el código ActionScript.
- Desea crear proyectos de Flex o sólo de ActionScript con el SDK de Flex, sin tener que utilizar Flash Professional o Flash Builder.

Entre los editores de código que proporcionan funciones específicas de ActionScript, cabe destacar:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (con paquetes de ActionScript y Flex)

Proceso de desarrollo de ActionScript

Independientemente del tamaño del proyecto de ActionScript, la utilización de un proceso para diseñar y desarrollar la aplicación permitirá trabajar con mayor eficacia. En los siguientes pasos se describe un proceso de desarrollo básico para crear una aplicación con ActionScript 3.0:

1. Diseñe la aplicación.

   Debe describir la aplicación de alguna forma antes de empezar a crearla.

2. Escriba el código ActionScript 3.0.

   Puede crear código ActionScript con Flash Professional, Flash Builder, Dreamweaver o un editor de texto.

3. Cree un proyecto de Flash o Flex para ejecutar el código.

   En Flash Professional, cree un archivo FLA, establezca la configuración de publicación, añada componentes de interfaz de usuario a la aplicación y haga referencia al código ActionScript. En Flex, defina la aplicación, añada componentes de interfaz de usuario utilizando MXML y haga referencia al código de ActionScript.

4. Publique y pruebe la aplicación ActionScript.

   La prueba de la aplicación implica su ejecución desde el entorno de desarrollo y la comprobación de que todo funciona según lo previsto.

Estos pasos no tienen por qué seguir este orden necesariamente y que tampoco es necesario finalizar completamente uno de los pasos antes de poder trabajar en otro. Por ejemplo, se puede diseñar una pantalla de la aplicación (paso 1) y luego crear los gráficos, botones y otros elementos (paso 3) antes de escribir el código ActionScript (paso 2) y probarlo (paso 4). O bien, se puede realizar parte del diseño y luego añadir un botón o elemento de interfaz en un momento dado, escribir código ActionScript para cada uno de estos elementos y probarlos. Resulta útil recordar estas cuatro fases del proceso de desarrollo. Sin embargo, en una situación real suele ser más eficaz ir pasando de una fase a otra según convenga.

Creación de clases personalizadas

El proceso de creación de clases para usarlas en los proyectos puede parecer desalentador. Sin embargo, la tarea más difícil de la creación de una clase es su diseño, es decir, la identificación de los métodos, propiedades y eventos que va a incluir.

Estrategias de diseño de una clase

El tema del diseño orientado a objetos es complejo; algunas personas han dedicado toda su carrera al estudio académico y la práctica profesional de esta disciplina. Sin embargo, a continuación se sugieren algunos enfoques que pueden ayudarle a comenzar.

1 Considere la función que desempeñarán las instancias de esta clase en la aplicación. Normalmente, los objetos desempeñan una de estas tres funciones:
   - Objeto de valor: estos objetos actúan principalmente como contenedores de datos. Es probable que tengan varias propiedades y pocos métodos (o algunas veces ninguno). Normalmente son representaciones de código de elementos claramente definidos. Por ejemplo, una clase Song (que representa una sola canción real) o una clase Playlist (que representa un grupo conceptual de canciones) en una aplicación de reproductor de música.
   - Objetos de visualización: son objetos que aparecen realmente en la pantalla. Por ejemplo, elementos de interfaz de usuario como una lista desplegable o una lectura de estado, o elementos gráficos como las criaturas de un videojuego, etc.
   - Estructura de aplicación: estos objetos desempeñan una amplia gama de funciones auxiliares en la lógica o el procesamiento llevado a cabo por las aplicaciones. Por ejemplo, puede hacer que un objeto realice determinados cálculos en una simulación biológica. Puede hacer un objeto responsable de sincronizar valores entre un control de dial y una lectura de volumen en una aplicación de reproductor de música. Otro puede administrar las reglas de un videojuego. O bien, puede crear una clase que cargue una imagen guardada en una aplicación de dibujo.

2 Decida la funcionalidad específica que necesitará la clase. Los distintos tipos de funcionalidad suelen convertirse en los métodos de la clase.

3 Si se prevé que la clase actúe como un objeto de valor, decida los datos que incluirán las instancias. Estos elementos son buenos candidatos para las propiedades.

4 Dado que la clase se diseña específicamente para el proyecto, lo más importante es proporcionar la funcionalidad que necesita la aplicación. Quizá le sirva de ayuda formularse estas preguntas:
   - ¿Qué elementos de información almacenará, rastreará y manipulará la aplicación? Esta decisión le ayudará a identificar los posibles objetos de valor y las propiedades.
   - ¿Qué conjunto de acciones realiza la aplicación? Por ejemplo, ¿qué sucede cuando la aplicación se carga por primera vez, cuando se hace clic en un botón determinado, cuando una película deja de reproducirse, etc? Éstos son buenos candidatos para los métodos. También puede ser propiedades, si las “acciones” implican cambiar valores individuales.

- Para cualquier acción, ¿qué información es necesaria para realizar esa acción? Estos elementos de información se convierten en los parámetros del método.
- Mientras la aplicación lleva a cabo su trabajo, ¿qué cambiará en la clase que deba ser conocido por las demás partes de la aplicación? Éstos son buenos candidatos para los eventos.

5 ¿Existe algún objeto similar al que se necesita, excepto que carece de cierta funcionalidad adicional que desea añadir? Considere la creación de una subclase. (Una *subclase* es una clase que se basa en la funcionalidad de otra clase existente, en lugar de definir su propia funcionalidad.) Por ejemplo, para crear una clase que sea un objeto visual en pantalla, utilice el comportamiento de un objeto de visualización existente como base para la clase. En este caso, el objeto de visualización (por ejemplo, MovieClip o Sprite) sería la *clase base* y su clase sería una ampliación de dicha clase.

Escritura del código de una clase

Cuando ya tenga un diseño para la clase, o al menos alguna idea de la información que almacena y de las acciones que necesitará realizar, la sintaxis real de escritura de una clase es bastante sencilla.

A continuación se describen los pasos básicos para crear su propia clase de ActionScript:

1 Abra un documento de texto nuevo en el programa editor de texto de ActionScript.

2 Introduzca una sentencia `class` para definir el nombre de la clase. Para añadir una sentencia `class`, indique las palabras `public class` y a continuación el nombre de la clase. Añada llaves de cierre y apertura que contengan el contenido de la clase (las definiciones de la propiedad y el método). Por ejemplo:

```
public class MyClass
{
}
```

La palabra `public` indica que es posible acceder a la clase desde cualquier otro código. Para conocer otras alternativas, consulte Atributos del espacio de nombres de control de acceso.

3 Escriba una sentencia `package` para indicar el nombre del paquete que contiene la clase. La sintaxis consta de la palabra `package`, seguida del nombre completo del paquete, más las llaves de apertura y cierre alrededor del bloque de sentencias `class`). Por ejemplo, cambie el código en el paso anterior a lo siguiente:

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

4 Defina cada propiedad de la clase con la sentencia `var` en el cuerpo de la clase. La sintaxis es la misma que se usa para declarar cualquier variable (añadiendo el modificador `public`). Por ejemplo, si se añaden estas líneas entre las llaves de apertura y cierre de la definición de clase, se crearán las propiedades `textProperty`, `numericProperty` y `dateProperty`:

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

5 Defina cada método de la clase con la misma sintaxis empleada para definir una función. Por ejemplo:

- Para crear un método `myMethod()`, introduzca:

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

- Para crear un constructor (el método especial al que se llama como parte del proceso de creación de una instancia de una clase), cree un método cuyo nombre coincida exactamente con el nombre de la clase:

```
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);
}
```

  Si no incluye un método constructor en la clase, el compilador creará automáticamente un constructor vacío en la clase. (Es decir, un constructor sin parámetros ni sentencias.)

Existen algunos otros elementos de la clase que pueden definirse. Estos elementos presentan mayor complejidad.

- Los *descriptores de acceso* son un cruce especial entre un método y una propiedad. Cuando se escribe el código para definir la clase, el descriptor de acceso se escribe como un método, de forma que es posible realizar varias acciones, en lugar de simplemente leer o asignar un valor, que es todo lo que puede hacerse cuando se define una propiedad. Sin embargo, cuando se crea una instancia de la clase, se trata al descriptor de acceso como una propiedad y sólo se utiliza el nombre para leer o asignar el valor.
- En ActionScript, los eventos no se definen con una sintaxis específica. Los eventos se definen en la clase utilizando la funcionalidad de la clase EventDispatcher.

Más temas de ayuda

Gestión de eventos

Ejemplo: Creación de una aplicación básica

ActionScript 3.0 puede utilizarse en varios entornos de desarrollo de aplicaciones, como las herramientas Flash Professional y Flash Builder y cualquier editor de texto.

En este ejemplo se indican los pasos necesarios para crear y mejorar una sencilla aplicación ActionScript 3.0 utilizando Flash Professional o Flash Builder. La aplicación que se va a crear presenta un patrón sencillo de utilización de archivos de clases de ActionScript 3.0 externos en Flash Professional y Flex.

Diseño de una aplicación ActionScript

Esta aplicación de ejemplo ActionScript es una aplicación “Hello World” estándar, por lo que su diseño resulta sencillo:

- La aplicación se denomina HelloWorld.
- Mostrará un solo campo de texto con las palabras “Hello World!”.
- La aplicación utiliza una sola clase orientada a objetos llamada Greeter. Este diseño permite a la clase utilizarse en un proyecto de Flash Professional o Flex.

- En este ejemplo, en primer lugar se crea una versión básica de la aplicación. Posteriormente se añade funcionalidad para que el usuario introduzca un nombre de usuario y que la aplicación compruebe el nombre en una lista de usuarios conocidos.

Teniendo en cuenta esta definición concisa, ya puede empezar a crear la aplicación.

Creación del proyecto HelloWorld y de la clase Greeter

Según el propósito del diseño de la aplicación Hello World, el código debería poder reutilizarse fácilmente. Para lograr este objetivo, la aplicación emplea una sola clase orientada a objetos denominada Greeter. Esta clase se utiliza desde una aplicación que se cree en Flash Builder o Flash Professional.

Para crear el proyecto HelloWorld y la clase Greeter en Flex:

1 En Flash Builder, seleccione File (Archivo) > New (Nuevo)> Flex Project (Proyecto de Flex).

2 Escriba HelloWorld como nombre del proyecto. Asegúrese de que el tipo de aplicación se establece como “Web (runs in Adobe Flash Player)” (Web (se ejecuta en Adobe Flash Player)), a continuación, haga clic en Finish (Finalizar).

   Flash Builder crea el proyecto y lo muestra en el explorador de paquetes. De forma predeterminada, el proyecto debe contener un archivo con el nombre HelloWorld.mxml, el cual se debe encontrar abierto en el editor.

3 A continuación, para crear una clase personalizada de ActionScript en la herramienta Flash Builder, seleccione File (Archivo) > New (Nuevo) >ActionScript Class (Clase de ActionScript).

4 En el cuadro de diálogo New ActionScript Class (Nueva clase de ActionScript), en el campo Name (Nombre), escriba **Greeter** como nombre de la clase y haga clic en Finish (Terminar).

   Aparecerá una nueva ventana de edición de ActionScript.

   Para continuar, consulte la sección Añadir código a la clase Greeter.

Para crear la clase Greeter en Flash Professional:

1 En Flash Professional, seleccione Archivo > Nuevo.

2 En el cuadro de diálogo Nuevo documento, seleccione Archivo ActionScript y haga clic en Aceptar.

   Aparecerá una nueva ventana de edición de ActionScript.

3 Seleccione Archivo > Guardar. Seleccione la carpeta en la que desea almacenar la aplicación, asigne el nombre **Greeter.as** al archivo ActionScript y haga clic en Aceptar.

   Para continuar, consulte la sección Añadir código a la clase Greeter.

Añadir código a la clase Greeter

La clase Greeter define un objeto, `Greeter`, que podrá utilizar en la aplicación HelloWorld.

Para añadir código a la clase Greeter:

1 Escriba el código siguiente en el nuevo archivo (parte del código se puede añadir por usted):

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

La clase Greeter incluye un único método `sayHello()`, el cual devuelve una cadena con el texto “Hello World!”.

2 Seleccione Archivo > Guardar para guardar este archivo de ActionScript.

Ya se puede utilizar la clase Greeter en la aplicación.

Creación de una aplicación que utilice el código ActionScript

La clase Greeter que ha creado define un conjunto de funciones de software con contenido propio, pero no representa una aplicación completa. Para utilizar la clase, se crea un documento de Flash Professional o un proyecto de Flex.

El código necesita una instancia de la clase Greeter. A continuación se explica cómo utilizar la clase Greeter en la aplicación.

Para crear una aplicación ActionScript con Flash Professional:

1 Seleccione Archivo > Nuevo.

2 En el cuadro de diálogo Nuevo documento, seleccione Archivo de Flash (ActionScript 3.0) y haga clic en Aceptar.

   Aparece una nueva ventana de documento.

3 Seleccione Archivo > Guardar. Seleccione la misma carpeta que contiene el archivo de clase Greeter.as, asigne al documento de Flash el nombre **HelloWorld.fla** y haga clic en Aceptar.

4 En la paleta de herramientas de Flash Professional, seleccione la herramienta Texto. Arrastre el cursor por el escenario para definir un nuevo campo de texto, con una anchura de aproximadamente 300 pí-xeles y una altura de unos 100 píxeles.

5 En el panel Propiedades, con el campo de texto todavía seleccionado en el escenario, establezca “Texto dinámico” como tipo de texto. Escriba **mainText** como nombre de instancia del campo de texto.

6 Haga clic en el primer fotograma de la línea de tiempo principal. Abra el panel Acciones eligiendo Ventana > Acciones.

7 En el panel Acciones, escriba el siguiente script:

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

8 Guarde el archivo.

Para continuar, consulte la sección Publicación y prueba de la aplicación ActionScript.

Para crear una aplicación ActionScript con Flash Builder:

1 Abra el archivo HelloWorld.mxml y añada código para que coincida con lo siguiente:

```
<?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>
```

Este proyecto de Flex incluye cuatro etiquetas MXML:

- Una etiqueta `<s:Application>` que define el contenedor de la aplicación
- Una etiqueta `<s:layout>` que define el estilo del diseño (diseño vertical) para la etiqueta de la aplicación
- Una etiqueta `<fx:Script>` que incluye código ActionScript
- Una etiqueta `<s:TextArea>` que define un campo para que se muestren mensajes de texto al usuario

El código de la etiqueta `<fx:Script>` define un método `initApp()` al que se llama cuando se carga la aplicación. El método `initApp()` establece el valor de texto de la etiqueta TextArea de `mainTxt` como la cadena “Hello World!” devuelta por el método `sayHello()` de la clase Greeter personalizada que se acaba de escribir.

2 Seleccione File (Archivo) > Save (Guardar) para guardar la aplicación.

Para continuar, consulte la sección Publicación y prueba de la aplicación ActionScript.

Publicación y prueba de la aplicación ActionScript

El desarrollo de software es un proceso repetitivo. Se escribe código, se intenta compilar y se edita hasta que se compile sin problemas. La aplicación compilada se ejecuta y se prueba para garantizar que cumple con el diseño previsto. Si no cumple las expectativas, el código se edita de nuevo hasta obtener el resultado esperado. Los entornos de desarrollo de Flash Professional y Flash Builder ofrecen diversas formas de publicar, probar y depurar las aplicaciones.

A continuación se explican los pasos básicos para probar la aplicación HelloWorld en cada entorno.

Para publicar y probar una aplicación ActionScript con Flash Professional:

1 Publique la aplicación y vea si aparecen errores de compilación. En la herramienta de edición de Flash Professional, seleccione Control > Probar película para compilar el código ActionScript y ejecutar la aplicación HelloWorld.

2 Si aparecen errores o advertencias en la ventana Salida al probar la aplicación, solucione estos errores en los archivos HelloWorld.fla o HelloWorld.as. Posteriormente intente probar de nuevo la aplicación.

3 Si no se producen errores de compilación, verá una ventana de Flash Player en la que se mostrará la aplicación Hello World.

Acaba de crear una aplicación orientada a objetos sencilla, pero completa, que utiliza ActionScript 3.0. Para continuar, consulte la sección Mejora de la aplicación HelloWorld.

Para publicar y probar una aplicación ActionScript con Flash Builder:

1 Seleccione Run (Ejecutar) > Run HelloWorld (Ejecutar HelloWorld).

2 Se iniciará la aplicación HelloWorld.

- Si aparecen errores o advertencias en la ventana de salida al probar la aplicación, solucione estos errores en los archivos HelloWorld.mxml o Greeter.as. Posteriormente intente probar de nuevo la aplicación.
- Si no se producen errores de compilación, se abrirá una ventana del navegador con la aplicación Hello World. Aparece el texto “Hello World!”.

Acaba de crear una aplicación orientada a objetos sencilla, pero completa, que utiliza ActionScript 3.0. Para continuar, consulte la sección Mejora de la aplicación HelloWorld.

Mejora de la aplicación HelloWorld

Para hacer la aplicación un poco más interesante, hará que pida y valide un nombre de usuario en una lista predefinida de nombres.

En primer lugar, deberá actualizar la clase Greeter para añadir funcionalidad nueva. A continuación, actualizará la aplicación para utilizar la nueva funcionalidad.

Para actualizar el archivo Greeter.as:

1 Abra el archivo Greeter.as.

2 Cambie el contenido del archivo por lo siguiente (las líneas nuevas y cambiadas se muestran en negrita):

```
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;
            }
        }
    }
}
```

La clase Greeter tiene ahora varias funciones nuevas:

- El conjunto `validNames` enumera los nombres de usuario válidos. El conjunto se inicializa como una lista de tres nombres cuando se carga la clase Greeter.
- El método `sayHello()` acepta ahora un nombre de usuario y cambia el saludo en función de algunas condiciones. Si `userName` es una cadena vacía (`""`), la propiedad `greeting` se define de forma que pida un nombre al usuario. Si el nombre de usuario es válido, el saludo se convierte en `"Hello,` *userName*`".` Finalmente, si no se cumple alguna de estas dos condiciones, la variable `greeting` se define como `"Sorry,` *userName*`, you are not on the list"`.
- El método `validName()` devuelve `true` si `inputName` se encuentra en el conjunto `validNames` y `false` si no se encuentra. La sentencia `validNames.indexOf(inputName)` comprueba cada una de las cadenas del conjunto `validNames` con respecto a la cadena `inputName`. El método `Array.indexOf()` devuelve la posición de í-ndice de la primera instancia de un objeto en un conjunto. Devuelve el valor -1 si el objeto no se encuentra en el conjunto.

A continuación se edita el archivo de la aplicación que hace referencia a esta clase de ActionScript.

Para modificar la aplicación utilizando Flash Professional:

1 Abra el archivo HelloWorld.fla.

2 Modifique el script en el Fotograma 1 de forma que se pase una cadena vacía (`""`) al método `sayHello()` de la clase Greeter:

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

3 Seleccione la herramienta Texto en la paleta Herramientas y cree dos nuevos campos de texto en el escenario. Colóquelos uno junto al otro directamente debajo del campo de texto `mainText` existente.

4 En el primer nuevo campo de texto, que es la etiqueta, escriba el texto **User Name:**.

5 Seleccione el otro campo de texto nuevo y, en el inspector de propiedades, seleccione Texto de entrada como tipo del campo de texto. Seleccione Línea única como tipo de línea. Escriba **textIn** como nombre de instancia.

6 Haga clic en el primer fotograma de la línea de tiempo principal.

7 En el panel Acciones, añada las líneas siguientes al final del script existente:

```
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);
    }
}
```

El código nuevo añade la siguiente funcionalidad:

- Las dos primeras líneas sólo definen los bordes de dos campos de texto.
- Un campo de texto de entrada, como el campo `textIn`, tiene un conjunto de eventos que puede distribuir. El método `addEventListener()` permite definir una función que se ejecuta cuando se produce un tipo de evento. En este caso, el evento es presionar una tecla del teclado.
- La función personalizada `keyPressed()` comprueba si la tecla presionada es la tecla Intro. De ser así, llama al método `sayHello()` del objeto `myGreeter` y le pasa el texto del campo de texto `textIn` como parámetro. El método devuelve una cadena de saludo basada en el valor pasado. Después se asigna la cadena devuelta a la propiedad `text` del campo de texto `mainText`.

A continuación se muestra el script completo para el Fotograma 1:

```
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 Guarde el archivo.

9 Seleccione Control > Probar película para ejecutar la aplicación.

Al ejecutar la aplicación, se solicita que indique un nombre de usuario. Si es válido (Sammy, Frank o Dean), la aplicación mostrará el mensaje de confirmación “hello”.

Para modificar la aplicación con Flash Builder:

1 Abra el archivo HelloWorld.mxml.

2 A continuación modifique la etiqueta `<mx:TextArea>` para indicar al usuario que es sólo para visualización. Cambie el color de fondo a un gris claro y establezca el atributo `editable` en `false`:

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

3 Añada ahora las siguientes líneas justo después de la etiqueta de cierre `<s:TextArea>`. Estas líneas crean un componente TextInput que permite al usuario especificar un valor de nombre de usuario:

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

El atributo `enter` define qué sucede cuando el usuario presiona la tecla Intro en el campo `userNameTxt`. En este ejemplo, el código pasa el texto en el campo al método `Greeter.sayHello()`. El saludo del campo `mainTxt` cambia como resultado.

El archivo HelloWorld.mxml presenta el siguiente aspecto:

```
<?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 Guarde el archivo HelloWorld.mxml modificado. Seleccione Run (Ejecutar) > Run HelloWorld (Ejecutar HelloWorld) para ejecutar la aplicación.

Al ejecutar la aplicación, ésta solicita que indique un nombre de usuario. Si es válido (Sammy, Frank o Dean), la aplicación mostrará el mensaje de confirmación “ `Hello,` *userName*”.

Capítulo 3: El lenguaje ActionScript y su sintaxis

ActionScript 3.0 consta del lenguaje ActionScript y la interfaz de programación de aplicaciones (API) de Adobe Flash Platform. El lenguaje principal es la parte de ActionScript que define la sintaxis del lenguaje, así como los tipos de datos de nivel superior. ActionScript 3.0 proporciona acceso programado a los motores de ejecución de la plataforma Adobe Flash: Adobe Flash Player y Adobe AIR.

Información general sobre el lenguaje

Los objetos constituyen la base del lenguaje ActionScript 3.0. Son sus componentes esenciales. Cada variable que se declare, cada función que se escriba y cada instancia de clase que se cree es un objeto. Se puede considerar que un programa ActionScript 3.0 es un grupo de objetos que realizan tareas, responden a eventos y se comunican entre sí.

Para los programadores que están familiarizados con la programación orientada a objetos (OOP) en Java o C++ los objetos son módulos que contienen dos tipos de miembros: datos almacenados en variables o propiedades miembro, y comportamiento al que se puede acceder a través de métodos. ActionScript 3.0 define los objetos de forma similar, aunque ligeramente distinta. En ActionScript 3.0, los objetos son simplemente colecciones de propiedades. Estas propiedades son contenedores que pueden contener no sólo datos, sino también funciones u otros objetos. Si se asocia una función a un objeto de esta manera, la función se denomina método.

Aunque la definición de ActionScript 3.0 puede parecer extraña a los programadores con experiencia en programación con Java o C++, en la práctica, la definición de tipos de objetos con clases de ActionScript 3.0 es muy similar a la manera de definir clases en Java o C++. La distinción entre las dos definiciones de objeto es importante al describir el modelo de objetos de ActionScript y otros temas avanzados, pero en la mayoría de las demás situaciones, el término *propiedades* se refiere a variables miembro de clase, no a métodos. Por ejemplo, en ActionScript 3.0 Reference for the Adobe Flash Platform se utiliza el término *propiedades* para hacer referencia a variables o propiedades de captores y definidores. El término *métodos* se utiliza para designar funciones que forman parte de una clase.

Una diferencia sutil entre las clases de ActionScript y las clases de Java o C++ es que, en ActionScript, las clases no son sólo entidades abstractas. Las clases de ActionScript se representan mediante *objetos de clase* que almacenan las propiedades y los métodos de la clase. Esto permite utilizar técnicas que pueden parecer extrañas a los programadores de Java y C++, como incluir sentencias o código ejecutable en el nivel superior de una clase o un paquete.

Otra diferencia entre las clases de ActionScript y las clases de Java o C++ es que cada clase de ActionScript tiene algo denominado *objetoprototipo*. En versiones anteriores de ActionScript, los objetos prototipo, vinculados entre sí en *cadenas de prototipos*, constituían en conjunto la base de toda la jerarquía de herencia de clases. Sin embargo, en ActionScript 3.0 los objetos prototipo desempeñan una función poco importante en el sistema de herencia. Pero el objeto prototipo puede ser útil como alternativa a las propiedades y los métodos estáticos si se desea compartir una propiedad y su valor entre todas las instancias de una clase.

En el pasado, los programadores expertos de ActionScript podían manipular directamente la cadena de prototipos con elementos especiales incorporados en el lenguaje. Ahora que el lenguaje proporciona una implementación más madura de una interfaz de programación basada en clases, muchos de estos elementos del lenguaje especiales, como `__proto__` y `__resolve`, ya no forman parte del lenguaje. Asimismo, las optimizaciones realizadas en el mecanismo de herencia interno, que aportan mejoras importantes de rendimiento, impiden el acceso directo al mecanismo de herencia.

Objetos y clases

En ActionScript 3.0, cada objeto se define mediante una clase. Una clase puede considerarse como una plantilla o un modelo para un tipo de objeto. Las definiciones de clase pueden incluir variables y constantes, que contienen valores de datos y métodos, que son funciones que encapsulan el comportamiento asociado a la clase. Los valores almacenados en propiedades pueden ser *valores simples* u otros objetos. Los valores simples son números, cadenas o valores booleanos.

ActionScript contiene diversas clases incorporadas que forman parte del núcleo del lenguaje. Algunas de estas clases incorporadas, como Number, Boolean y String, representan los valores simples disponibles en ActionScript. Otras, como las clases Array, Math y XML, definen objetos más complejos.

Todas las clases, tanto las incorporadas como las definidas por el usuario, se derivan de la clase Object. Para los programadores con experiencia previa en ActionScript, es importante tener en cuenta que el tipo de datos Object ya no es el tipo de datos predeterminado, aunque todas las demás clases se deriven de él. En ActionScript 2.0, las dos líneas de código siguientes eran equivalentes, ya que la ausencia de una anotación de tipo significaba que una variable era de tipo Object:

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

En ActionScript 3.0 se introduce el concepto de variables sin tipo, que pueden designarse de las dos maneras siguientes:

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

Una variable sin tipo no es lo mismo que una variable de tipo Object. La principal diferencia es que las variables sin tipo pueden contener el valor especial `undefined`, mientras que una variable de tipo Object no puede contener ese valor.

Un programador puede definir sus propias clases mediante la palabra clave `class`. Las propiedades de clase se pueden definir de tres formas diferentes: las constantes se pueden definir con la palabra clave `const`, las variables con la palabra clave `var`, y las propiedades de captores y definidores con los atributos `get` y `set` de una declaración de método. Los métodos se declaran con la palabra clave `function`.

Para crear una instancia de una clase hay que utilizar el operador `new`. En el ejemplo siguiente se crea una instancia de la clase Date denominada `myBirthday`.

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

Paquetes y espacios de nombres

Los conceptos de paquete y espacio de nombres están relacionados. Los paquetes permiten agrupar definiciones de clase de una manera que permite compartir código fácilmente y minimiza los conflictos de nomenclatura. Los espacios de nombres permiten controlar la visibilidad de los identificadores, como los nombres de propiedades y métodos, y pueden aplicarse a código tanto dentro como fuera de un paquete. Los paquetes permiten organizar los archivos de clase y los espacios de nombres permiten administrar la visibilidad de propiedades y métodos individuales.

Paquetes

En ActionScript 3.0, los paquetes se implementan con espacios de nombres, pero son un concepto distinto. Al declarar un paquete, se crea implícitamente un tipo especial de espacio de nombres que se conoce en tiempo de compilación. Si los espacios de nombres se crean explícitamente, no se conocerán necesariamente en tiempo de compilación.

En el ejemplo siguiente se utiliza la directiva `package` para crear un paquete sencillo que contiene una clase:

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

El nombre de la clase de este ejemplo es SampleCode. Debido a que la clase se encuentra en el interior del paquete samples, el compilador califica el nombre de clase automáticamente en tiempo de compilación como su nombre completo: samples.SampleCode. El compilador también califica los nombres de propiedades y métodos, de forma que `sampleGreeting` y `sampleFunction()` se convierten en `samples.SampleCode.sampleGreeting` y `samples.SampleCode.sampleFunction()`, respectivamente.

Muchos desarrolladores, especialmente los que tienen experiencia en programación con Java, pueden elegir colocar únicamente clases en el nivel superior de un paquete. Sin embargo, ActionScript 3.0 no sólo admite clases en el nivel superior de un paquete, sino también variables, funciones e incluso sentencias. Un uso avanzado de esta característica consiste en definir un espacio de nombres en el nivel superior de un paquete de forma que esté disponible para todas las clases del paquete. Sin embargo, hay que tener en cuenta que sólo se admiten dos especificadores de acceso en el nivel superior de un paquete: `public` e `internal`. A diferencia de Java, que permite declarar clases anidadas como privadas, ActionScript 3.0 no admite clases anidadas privadas ni anidadas.

Sin embargo, en muchos otros aspectos los paquetes de ActionScript 3.0 son similares a los paquetes del lenguaje de programación Java. Como se puede ver en el ejemplo anterior, las referencias de nombre completo a paquetes se expresan con el operador punto (`.`), igual que en Java. Se pueden utilizar paquetes para organizar el código en una estructura jerárquica intuitiva que puedan usar otros programadores. Esto permite compartir código fácilmente, ya que ofrece una manera de crear un paquete para compartirlo con otros y de utilizar en el código paquetes creados por otros.

El uso de paquetes también ayuda a garantizar que los nombres de los identificadores utilizados son únicos y no entran en conflicto con otros nombres de identificador. De hecho, se podría decir que ésta es la ventaja principal de los paquetes. Por ejemplo, dos programadores que desean compartir código pueden haber creado una clase denominada SampleCode. Sin los paquetes, esto crearía un conflicto de nombres y la única resolución sería cambiar el nombre de una de las clases. Sin embargo, con los paquetes el conflicto de nombres se evita fácilmente colocando una de las clases (o las dos, preferiblemente) en paquetes con nombres únicos.

También se pueden incluir puntos incorporados en el nombre de paquete para crear paquetes anidados. Esto permite crear una organización jerárquica de paquetes. Un buen ejemplo de ello es el paquete flash.display que proporciona ActionScript 3.0. Este paquete se anida dentro del paquete flash.

La mayor parte de ActionScript 3.0 se organiza en el paquete flash. Por ejemplo, el paquete flash.display contiene la API de la lista de visualización y el paquete flash.events contiene el nuevo modelo de eventos.

Creación de paquetes

ActionScript 3.0 proporciona una gran flexibilidad para organizar los paquetes, las clases y los archivos de código fuente. Las versiones anteriores de ActionScript sólo permitían una clase por archivo de código fuente y requerían que el nombre del archivo coincidiera con el nombre de la clase. ActionScript 3.0 permite incluir varias clases en un archivo de código fuente, pero sólo puede estar disponible una clase de cada archivo para el código externo a ese archivo. Es decir, sólo se puede declarar una clase de cada archivo en una declaración de paquete. Hay que declarar las clases adicionales fuera de la definición de paquete, lo que hace que estas clases sean invisibles para el código externo a ese archivo de código fuente. El nombre de clase declarado en la definición de paquete debe coincidir con el nombre del archivo de código fuente.

ActionScript 3.0 también proporciona más flexibilidad para la declaración de paquetes. En versiones anteriores de ActionScript, los paquetes sólo representaban directorios en los que se colocaban archivos de código fuente y no se declaraban con la sentencia `package`, sino que se incluía el nombre de paquete como parte del nombre completo de clase en la declaración de clase. Aunque los paquetes siguen representando directorios en ActionScript 3.0, pueden contener algo más que clases. En ActionScript 3.0 se utiliza la sentencia `package` para declarar un paquete, lo que significa que también se pueden declarar variables, funciones y espacios de nombres en el nivel superior de un paquete. Incluso se pueden incluir sentencias ejecutables en el nivel superior de un paquete. Si se declaran variables, funciones o espacios de nombres en el nivel superior de un paquete, los únicos atributos disponibles en ese nivel son `public` e `internal`, y sólo una declaración de nivel de paquete por archivo puede utilizar el atributo `public`, independientemente de que la declaración sea una clase, una variable, una función o un espacio de nombres.

Los paquetes son útiles para organizar el código y para evitar conflictos de nombres. No se debe confundir el concepto de paquete con el concepto de herencia de clases, que no está relacionado con el anterior. Dos clases que residen en el mismo paquete tendrán un espacio de nombres común, pero no estarán necesariamente relacionadas entre sí de ninguna otra manera. Asimismo, un paquete anidado puede no tener ninguna relación semántica con su paquete principal.

Importación de paquetes

Si se desea utilizar una clase que está dentro un paquete, se debe importar el paquete o la clase específica. Esto varía con respecto a ActionScript 2.0, donde la importación de clases era opcional.

Por ejemplo, tenga en cuenta el ejemplo de la clase SampleCode presentada anteriormente. Si la clase reside en un paquete denominado samples, hay que utilizar una de las siguientes sentencias de importación antes de utilizar la clase SampleCode:

```
import samples.*;
```

o

```
import samples.SampleCode;
```

En general, la sentencias `import` deben ser lo más específicas posible. Si se pretende utilizar la clase SampleCode desde el paquete samples, hay que importar únicamente la clase SampleCode, no todo el paquete al que pertenece. La importación de paquetes completos puede producir conflictos de nombres inesperados.

También se debe colocar el código fuente que define el paquete o la clase en la *ruta de clases*. La ruta de clases es una lista de rutas de directorio locales definida por el usuario que determina dónde buscará el compilador los paquetes y las clases importados. La ruta de clases se denomina a veces *ruta de compilación* o *ruta de código fuente*.

Tras importar correctamente la clase o el paquete, se puede utilizar el nombre completo de la clase (samples.SampleCode) o simplemente el nombre de clase (SampleCode).

Los nombres completos son útiles cuando hay ambigüedad en el código a causa de clases, métodos o propiedades con nombre idénticos, pero pueden ser difíciles de administrar si se usan para todos los identificadores. Por ejemplo, la utilización del nombre completo produce código demasiado extenso al crear una instancia de la clase SampleCode:

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

A medida que aumentan los niveles de paquetes anidados, la legibilidad del código disminuye. En las situaciones en las que se esté seguro de que los identificadores ambiguos no constituirán un problema, se puede hacer el código más legible utilizando identificadores sencillos. Por ejemplo, al crear una nueva instancia de la clase SampleCode, el resultado será mucho menos extenso si se utiliza sólo el identificador de clase:

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

Si se intenta utilizar nombres de identificador sin importar primero el paquete o la clase apropiados, el compilador no podrá encontrar las definiciones de clase. Por otra parte, si se importa un paquete o una clase, cualquier intento de definir un nombre que entre en conflicto con un nombre importado generará un error.

Cuando se crea un paquete, el especificador de acceso predeterminado para todos los miembros del paquete es `internal`, lo que significa que, de manera predeterminada, los miembros del paquete sólo estarán visibles para los otros miembros del paquete. Si se desea que una clase esté disponible para código externo al paquete, se debe declarar la clase como `public`. Por ejemplo, el siguiente paquete contiene dos clases, SampleCode y CodeFormatter:

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

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

La clase SampleCode está visible fuera del paquete porque se ha declarado como una clase `public`. Sin embargo, la clase CodeFormatter sólo está visible dentro del mismo paquete samples. Si se intenta acceder la clase CodeFormatter fuera del paquete samples, se generará un error, como se indica en el siguiente ejemplo:

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

Si se desea que ambas clases estén disponibles para código externo al paquete, se deben declarar las dos como `public`. No se puede aplicar el atributo `public` a la declaración del paquete.

Los nombres completos son útiles para resolver conflictos de nombres que pueden producirse al utilizar paquetes. Este escenario puede surgir si se importan dos paquetes que definen clases con el mismo identificador. Por ejemplo, considérese el siguiente paquete, que también tiene una clase denominada SampleCode:

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

Si se importan ambas clases de la manera siguiente, se producirá un conflicto de nombres al utilizar la clase SampleCode:

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

El compilador no sabe qué clase SampleCode debe utilizar. Para resolver este conflicto, hay que utilizar el nombre completo de cada clase, de la manera siguiente:

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

***Nota:** los programadores con experiencia en C++ suelen confundir la sentencia `import` con `#include`. La directiva `#include` es necesaria en C++ porque los compiladores de C++ procesan un archivo cada vez y no buscan definiciones de clases en otros archivos, a menos que se incluya explícitamente un archivo de encabezado. ActionScript 3.0 tiene una directiva `include`, pero no está diseñada para importar clases y paquetes. Para importar clases o paquetes en ActionScript 3.0, hay que usar la sentencia `import` y colocar el archivo de código fuente que contiene el paquete en la ruta de clases.*

Espacios de nombres

Los espacios de nombres ofrecen control sobre la visibilidad de las propiedades y los métodos que se creen. Los especificadores de control de acceso `public`, `private`, `protected` e `internal` son espacios de nombres incorporados. Si estos especificadores de control de acceso predefinidos no se adaptan a las necesidades del programador, es posible crear espacios de nombres personalizados.

Para los programadores que estén familiarizados con los espacios de nombres XML, gran parte de lo que se va a explicar no resultará nuevo, aunque la sintaxis y los detalles de la implementación de ActionScript son ligeramente distintos de los de XML. Si nunca se ha trabajado con espacios de nombres, el concepto en sí es sencillo, pero hay que aprender la terminología específica de la implementación.

Para comprender mejor cómo funcionan los espacios de nombres, es necesario saber que el nombre de una propiedad o un método siempre contiene dos partes: un identificador y un espacio de nombres. El identificador es lo que generalmente se considera un nombre. Por ejemplo, los identificadores de la siguiente definición de clase son `sampleGreeting` y `sampleFunction()`:

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

Siempre que las definiciones no estén precedidas por un atributo de espacio de nombres, sus nombres se califican mediante el espacio de nombres `internal` predeterminado, lo que significa que sólo estarán visibles para los orígenes de llamada del mismo paquete. Si el compilador está configurado en modo estricto, emite una advertencia para indicar que el espacio de nombres `internal` se aplica a cualquier identificador que no tenga un atributo de espacio de nombres. Para asegurarse de que un identificador esté disponible en todas partes, hay que usar específicamente el atributo `public` como prefijo del nombre de identificador. En el ejemplo de código anterior, `sampleGreeting` y `sampleFunction()` tienen `internal` como valor de espacio de nombres.

Se deben seguir tres pasos básicos al utilizar espacios de nombres. En primer lugar, hay que definir el espacio de nombres con la palabra clave `namespace`. Por ejemplo, el código siguiente define el espacio de nombres `version1`:

```
namespace version1;
```

En segundo lugar, se aplica el espacio de nombres utilizándolo en lugar de utilizar un especificador de control de acceso en una declaración de propiedad o método. El ejemplo siguiente coloca una función denominada `myFunction()` en el espacio de nombres `version1`:

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

En tercer lugar, tras aplicar el espacio de nombres, se puede hacer referencia al mismo con la directiva `use` o calificando el nombre de un identificador con un espacio de nombres. En el ejemplo siguiente se hace referencia a la función `myFunction()` mediante la directiva `use`:

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

También se puede utilizar un nombre completo para hacer referencia a la función `myFunction()`, como se indica en el siguiente ejemplo:

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

Definición de espacios de nombres

Los espacios de nombres contienen un valor, el Identificador uniforme de recurso (URI), que a veces se denomina *nombre del espacio de nombres*. El URI permite asegurarse de que la definición del espacio de nombres es única.

Para crear un espacio de nombres se declara una definición de espacio de nombres de una de las dos maneras siguientes. Se puede definir un espacio de nombres con un URI explícito, de la misma manera que se define un espacio de nombres XML, o se puede omitir el URI. En el siguiente ejemplo se muestra la manera de definir un espacio de nombres mediante un URI:

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

El URI constituye una cadena de identificación única para ese espacio de nombres. Si se omite el URI, como en el siguiente ejemplo, el compilador creará una cadena de identificación interna única en lugar del URI. El usuario no tiene acceso a esta cadena de identificación interna.

```
namespace flash_proxy;
```

Una vez definido un espacio de nombres, con o sin URI, ese espacio de nombres no podrá definirse de nuevo en el mismo ámbito. Si se intenta definir un espacio de nombres definido previamente en el mismo ámbito, se producirá un error del compilador.

Si se define un espacio de nombres dentro de un paquete o una clase, el espacio de nombres puede no estar visible para código externo al paquete o la clase, a menos que se use el especificador de control de acceso apropiado. Por ejemplo, el código siguiente muestra el espacio de nombres `flash_proxy` definido en el paquete flash.utils. En el siguiente ejemplo, la falta de un especificador de control de acceso significa que el espacio de nombres `flash_proxy` sólo estará visible para el código del paquete flash.utils y no estará visible para el código externo al paquete:

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

El código siguiente utiliza el atributo `public` para hacer que el espacio de nombres `flash_proxy` esté visible para el código externo al paquete:

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

Aplicación de espacios de nombres

Aplicar un espacio de nombres significa colocar una definición en un espacio de nombres. Entre las definiciones que se puede colocar en espacios de nombres se incluyen funciones, variables y constantes (no se puede colocar una clase en un espacio de nombres personalizado).

Considérese, por ejemplo, una función declarada con el espacio de nombres de control de acceso `public`. Al utilizar el atributo `public` en una definición de función se coloca la función en el espacio de nombres public, lo que hace que esté disponible para todo el código. Una vez definido un espacio de nombres, se puede utilizar el espacio de nombres definido de la misma manera que se utilizaría el atributo `public` y la definición estará disponible para el código que puede hacer referencia al espacio de nombres personalizado. Por ejemplo, si se define un espacio de nombres `example1`, se puede añadir un método denominado `myFunction()` utilizando `example1` como un atributo, como se indica en el siguiente ejemplo:

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

Si se declara el método `myFunction()` con el espacio de nombres `example1` como un atributo, significa que el método pertenece al espacio de nombres `example1`.

Se debe tener en cuenta lo siguiente al aplicar espacios de nombres:

- Sólo se puede aplicar un espacio de nombres a cada declaración.
- No hay manera de aplicar un atributo de espacio de nombres a más de una definición simultáneamente. Es decir, si se desea aplicar el espacio de nombres a diez funciones distintas, se debe añadir el espacio de nombres como un atributo a cada una de las diez definiciones de función.
- Si se aplica un espacio de nombres, no se puede especificar también un especificador de control de acceso, ya que los espacios de nombres y los especificadores de control de acceso son mutuamente excluyentes. Es decir, no se puede declarar una función o propiedad como `public`, `private`, `protected` o `internal` y aplicar también el espacio de nombres.

Referencia a un espacio de nombres

No es necesario hacer referencia explícita a un espacio de nombres al usar un método o una propiedad declarados con alguno de los espacios de nombres de control de acceso, como `public`, `private`, `protected` e `internal`. Esto se debe a que el acceso a estos espacios de nombres especiales se controla por el contexto. Por ejemplo, las definiciones colocadas en el espacio de nombres `private` estarán disponibles automáticamente para el código de la misma clase. Sin embargo, para los espacios de nombres personalizados que se definan no existirá este control mediante el contexto. Para poder utilizar un método o una propiedad que se ha colocado en un espacio de nombres personalizado, se debe hacer referencia al espacio de nombres.

Se puede hacer referencia a espacios de nombres con la directiva `use namespace` o se puede calificar el nombre con el espacio de nombres mediante el signo calificador de nombre (`::`). Al hacer referencia a un espacio de nombres con la directiva `use namespace` "se abre" el espacio de nombres, de forma que se puede aplicar a cualquier identificador que no esté calificado. Por ejemplo, si se define el espacio de nombres `example1`, se puede acceder a nombres de ese espacio de nombres mediante `use namespace example1`:

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

Es posible abrir más de un espacio de nombres simultáneamente. Cuando se abre un espacio de nombres con `use namespace`, permanece abierto para todo el bloque de código en el que se abrió. No hay forma de cerrar un espacio de nombres explícitamente.

Sin embargo, si hay más de un espacio de nombres abierto, aumenta la probabilidad de que se produzcan conflictos de nombres. Si se prefiere no abrir un espacio de nombres, se puede evitar la directiva `use namespace` calificando el nombre del método o la propiedad con el espacio de nombres y el signo calificador de nombre. Por ejemplo, el código siguiente muestra la manera de calificar el nombre `myFunction()` con el espacio de nombres `example1`:

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

Uso de espacios de nombres

Puede encontrar un ejemplo real de un espacio de nombres que se utiliza para evitar conflictos de nombre en la clase flash.utils.Proxy que forma parte de ActionScript 3.0. La clase Proxy, que es la sustitución de la propiedad `Object.__resolve` de ActionScript 2.0, permite interceptar referencias a propiedades o métodos no definidos antes de que se produzca un error. Todos los métodos de la clase Proxy residen en el espacio de nombres `flash_proxy` para evitar conflictos de nombres.

Para entender mejor cómo se utiliza el espacio de nombres `flash_proxy`, hay que entender cómo se utiliza la clase Proxy. La funcionalidad de la clase Proxy sólo está disponible para las clases que heredan de ella. Es decir, si se desea utilizar los métodos de la clase Proxy en un objeto, la definición de clase del objeto debe ampliar la clase Proxy. Por ejemplo, si desea se interceptar intentos de llamar a un método no definido, se debe ampliar la clase Proxy y después reemplazar el método `callProperty()` de la clase Proxy.

La implementación de espacios de nombres es generalmente un proceso en tres pasos consistente en definir y aplicar un espacio de nombres, y después hacer referencia al mismo. Sin embargo, como nunca se llama explícitamente a ninguno de los métodos de la clase Proxy, sólo se define y aplica el espacio de nombres `flash_proxy`, pero nunca se hace referencia al mismo. ActionScript 3.0 define el espacio de nombres `flash_proxy` y lo aplica en la clase Proxy. El código sólo tiene que aplicar el espacio de nombres `flash_proxy` a las clases que amplían la clase Proxy.

El espacio de nombres `flash_proxy` se define en el paquete flash.utils de una manera similar a la siguiente:

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

El espacio de nombres se aplica a los métodos de la clase Proxy, como se indica en el siguiente fragmento de dicha clase:

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

Como se indica en el código siguiente, se debe importar primero la clase Proxy y el espacio de nombres `flash_proxy`. A continuación se debe declarar la clase de forma que amplíe la clase Proxy (también se debe añadir el atributo `dynamic` si se compila en modo estricto). Al sustituir el método `callProperty()`, se debe utilizar el espacio de nombres `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);
        }
    }
}
```

Si se crea una instancia de la clase MyProxy y se llama a un método no definido, como el método `testing()` llamado en el ejemplo siguiente, el objeto Proxy intercepta la llamada al método y ejecuta las sentencias del método `callProperty()` sustituido (en este caso, una simple sentencia `trace()`).

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

Tener los métodos de la clase Proxy dentro del espacio de nombres `flash_proxy` ofrece dos ventajas. En primer lugar, tener un espacio de nombres independiente reduce el desorden en la interfaz pública de cualquier clase que amplíe la clase Proxy. (Hay aproximadamente una docena de métodos en la clase Proxy que se pueden sustituir; todos ellos han sido diseñados para no ser llamados directamente. Colocarlos todos en el espacio de nombres public podría provocar confusiones.) En segundo lugar, el uso del espacio de nombres `flash_proxy` evita los conflictos de nombres en caso de que la subclase de Proxy contenga métodos de instancia con nombres que coincidan con los de los métodos de la clase Proxy. Por ejemplo, supongamos que un programador desea asignar a uno de sus métodos el nombre `callProperty()`. El código siguiente es aceptable porque su versión del método `callProperty()` está en un espacio de nombres distinto:

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

Los espacios de nombres también pueden ser útiles cuando se desea proporcionar acceso a métodos o propiedades de una manera que se no puede realizar con los cuatro especificadores de control de acceso (`public`, `private`, `internal` y `protected`). Por ejemplo, un programador puede tener algunos métodos de utilidad repartidos por varios paquetes. Desea que estos métodos estén disponibles para todos los paquetes, pero no quiere que sean públicos. Para ello, se puede crear un espacio de nombres y utilizarlo como un especificador de control de acceso especial.

En el ejemplo siguiente se utiliza un espacio de nombres definido por el usuario para agrupar dos funciones que residen en paquetes distintos. Al agruparlos en el mismo espacio de nombres, se puede hacer que ambas funciones estén visibles para una clase o un paquete mediante una única sentencia `use namespace`.

En este ejemplo se utilizan cuatro archivos para ilustrar la técnica. Todos los archivos deben estar en la ruta de clases. El primer archivo, myInternal.as, se utiliza para definir el espacio de nombres `myInternal`. Como el archivo está en un paquete denominado example, se debe colocar en una carpeta denominada example. El espacio de nombres se marca como `public` de forma que se pueda importar en otros paquetes.

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

Los archivos Utility.as y Helper.as definen las clases que contienen los métodos que deben estar disponibles para otros paquetes. La clase Utility está en el paquete example.alpha, lo que significa que se debe colocar el archivo dentro de una subcarpeta de la carpeta example denominada alpha. La clase Helper está en el paquete example.beta, por lo que se debe colocar el archivo dentro de otra subcarpeta de la carpeta example denominada beta. Ambos paquetes, example.alpha y example.beta, deben importar el espacio de nombres antes de utilizarlo.

```
// 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;
        }
    }
}
```

El cuarto archivo, NamespaceUseCase.as, es la clase principal de la aplicación, y debe estar en el mismo nivel que la carpeta example. En Flash Professional, esta clase se utilizaría como la clase de documento para el archivo FLA. La clase NamespaceUseCase también importa el espacio de nombres `myInternal` y lo utiliza para llamar a los dos métodos estáticos que residen en los otros paquetes. El ejemplo utiliza métodos estáticos sólo para simplificar el código. Se pueden colocar tanto métodos estáticos como métodos de instancia en el espacio de nombres `myInternal`.

```
// 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]
        }
    }
}
```

Variables

Las variables permiten almacenar los valores que se utilizan en el programa. Para declarar una variable se debe utilizar la sentencia `var` con el nombre de variable. En ActionScript 3.0, el uso de la sentencia `var` es siempre obligatorio. Por ejemplo, la línea siguiente de código ActionScript declara una variable denominada `i`:

```
var i;
```

Si se omite la sentencia `var` al declarar una variable, se producirá un error del compilador en modo estricto y un error en tiempo de ejecución en modo estándar. Por ejemplo, la siguiente línea de código producirá un error si la variable `i` no se ha definido previamente:

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

La asociación de una variable con un tipo de datos, debe realizarse al declarar la variable. Es posible declarar una variable sin designar su tipo, pero esto generará una advertencia del compilador en modo estricto. Para designar el tipo de una variable se añade el nombre de la variable con un signo de dos puntos (:), seguido del tipo de la variable. Por ejemplo, el código siguiente declara una variable `i` de tipo int:

```
var i:int;
```

Se puede asignar un valor a una variable mediante el operador de asignación (`=`). Por ejemplo, el código siguiente declara una variable `i` y le asigna el valor 20:

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

Puede ser más cómodo asignar un valor a una variable a la vez que se declara la variable, como en el siguiente ejemplo:

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

La técnica de asignar un valor a una variable en el momento de declararla se suele utilizar no sólo al asignar valores simples, como enteros y cadenas, sino también al crear un conjunto o una instancia de una clase. En el siguiente ejemplo se muestra un conjunto que se declara y recibe un valor en una línea de código.

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

Para crear una instancia de una clase hay que utilizar el operador `new`. En el ejemplo siguiente se crea una instancia de una clase denominada `CustomClass` y se asigna una referencia a la instancia de clase recién creada a la variable denominada `customItem`:

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

Si hubiera que declarar más de una variable, se pueden declarar todas en una línea de código utilizando el operador coma (`,`) para separar las variables. Por ejemplo, el código siguiente declara tres variables en una línea de código:

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

También se puede asignar valores a cada una de las variables en la misma línea de código. Por ejemplo, el código siguiente declara tres variables (`a`, `b` y `c`) y asigna un valor a cada una de ellas:

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

Aunque se puede utilizar el operador coma para agrupar declaraciones de variables en una sentencia, al hacerlo el código será menos legible.

Aspectos básicos del ámbito de variables

El *ámbito* de una variable es el área del código en la que se puede acceder a la variable mediante una referencia léxica. Una variable *global* está definida en todas las áreas del código, mientras que una variable *local* sólo está definida en una parte del código. En ActionScript 3.0, las variables siempre se asignan al ámbito de la función o la clase en la que se han declarado. Una variable global es una variable que se define fuera de una definición de función o clase. Por ejemplo, el código siguiente crea una variable global `strGlobal` declarándola fuera de las funciones. El ejemplo muestra que una variable global está disponible tanto dentro como fuera de una definición de función.

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

Las variables locales se declaran dentro de una definición de función. La parte más pequeña de código para la que se puede definir una variable local es una definición de función. Una variable local declarada en una función sólo existirá en esa función. Por ejemplo, si se declara una variable denominada `str2` dentro de una función denominada `localScope()`, dicha variable no estará disponible fuera de la función.

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

Si el nombre de la variable que utiliza para la variable local ya se ha declarado como variable global, la definición local oculta (o reemplaza) la definición global mientras la variable local se encuentre dentro del ámbito. La variable global continuará existiendo fuera de la función. Por ejemplo, el siguiente fragmento de código crea una variable de cadena global denominada `str1` y a continuación crea una variable local con el mismo nombre dentro de la función `scopeTest()`. La sentencia `trace` de la función devuelve el valor local de la variable, pero la sentencia `trace` situada fuera de la función devuelve el valor global de la variable.

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

Las variables de ActionScript, a diferencia de las variables de C++ y Java, no tienen ámbito a nivel de bloque. Un bloque de código es un grupo de sentencias entre una llave inicial (`{`) y una llave final (`}`). En algunos lenguajes de programación, como C++ y Java, las variables declaradas dentro de un bloque de código no están disponibles fuera de ese bloque de código. Esta restricción de ámbito se denomina ámbito a nivel de bloque y no existe en ActionScript. Si se declara una variable dentro de un bloque de código, dicha variable no sólo estará disponible en ese bloque de código, sino también en cualquier otra parte de la función a la que pertenece el bloque de código. Por ejemplo, la siguiente función contiene variables definidas en varios ámbitos de bloque. Todas las variables están disponibles en toda la función.

```
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"]);
```

Una consecuencia interesante de la falta de ámbito a nivel de bloque es que se puede leer el valor de una variable o escribir en ella antes de que se declare, con tal de que se declare antes del final de la función. Esto se debe a una técnica denominada *hoisting*, que consiste en que el compilador mueve todas las declaraciones de variables al principio de la función. Por ejemplo, el código siguiente se compila aunque la función `trace()` inicial para la variable `num` está antes de la declaración de la variable `num`:

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

Sin embargo, el compilador no mueve ninguna sentencia de asignación al principio. Esto explica por qué la función `trace()` inicial de `num` devuelve `NaN` (no es un número), que es el valor predeterminado para variables con tipo de datos Number. Así, es posible asignar valores a variables incluso antes de declararlas, como se muestra en el siguiente ejemplo:

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

Valores predeterminados

Un *valor predeterminado* es el valor que contiene una variable antes de que se establezca su valor. Una variable se *inicializa* al establecer su valor por primera vez. Si se declara una variable pero no establece su valor, dicha variable estará *sin inicializar*. El valor de una variable no inicializada depende del tipo de datos que tenga. En la tabla siguiente se describen los valores predeterminados de las variables, clasificados por tipo de datos:

Tipo de datos	**Valor predeterminado**
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`
uint	0
No declarada (equivalente a anotación de tipo `*`)	`undefined`
Todas las demás clases, incluidas las clases definidas por el usuario.	`null`

Para variables de tipo Number, el valor predeterminado es `NaN` (no es un número), que es un valor especial definido por la norma IEEE-754 para designar un valor que no representa un número.

Si se declara una variable pero no se declara su tipo de datos, se aplica el tipo de datos predeterminado, `*`, que en realidad indica que la variable no tiene tipo. Si tampoco se inicializa una variable sin tipo con un valor, su valor predeterminado será `undefined`.

Para tipos de datos distintos de Boolean, Number, int y uint, el valor predeterminado de cualquier variable no inicializada es `null`. Esto se aplica a todas las clases definidas mediante ActionScript 3.0, así como a las clases personalizadas que se creen.

El valor `null` no es válido para variables de tipo Boolean, Number, int o uint. Si se intenta asignar el valor `null` a una variable de este tipo, dicho valor se convierte en el valor predeterminado para ese tipo de datos. A las variables de tipo Object se les puede asignar un valor `null`. Si se intenta asignar el valor `undefined` a una variable de tipo Object, dicho valor se convierte en `null`.

Para variables de tipo Number, hay una función especial de nivel superior denominada `isNaN()` que devuelve el valor booleano `true` si la variable no es un número, y `false` en caso contrario.

Tipos de datos

Un *tipo de datos* define un conjunto de valores. Por ejemplo, el tipo de datos Boolean es el conjunto de dos valores bien definidos: `true` y `false`. Además del tipo de datos Boolean, ActionScript 3.0 define varios tipos de datos utilizados con frecuencia, como String, Number y Array. Un programador puede definir sus propios tipos de datos utilizando clases o interfaces para definir un conjunto de valores personalizado. En ActionScript 3.0 todos los valores, tanto simples como complejos, son objetos.

Un *valor simple* es un valor que pertenece a uno de los tipos de datos siguientes: Boolean, int, Number, String y uint. Trabajar con valores simples suele ser más rápido que trabajar con valores complejos, ya que ActionScript almacena los valores simples de una manera especial que permite optimizar la velocidad y el uso de la memoria.

***Nota:** para los interesados en los detalles técnicos, ActionScript almacena internamente los valores simples como objetos inmutables. El hecho de que se almacenen como objetos inmutables significa que pasar por referencia es en realidad lo mismo que pasar por valor. Esto reduce el uso de memoria y aumenta la velocidad de ejecución, ya que las referencias ocupan normalmente bastante menos que los valores en sí.*

Un *valor complejo* es un valor que no es un valor simple. Entre los tipos de datos que definen conjuntos de valores complejos se encuentran Array, Date, Error, Function, RegExp, XML y XMLList.

Muchos lenguajes de programación distinguen entre los valores simples y los objetos que los contienen. Java, por ejemplo, tiene un valor simple int y la clase java.lang.Integer que lo contiene. Los valores simples de Java no son objetos, pero los objetos que los contienen sí, lo que hace que los valores simples sean útiles para algunas operaciones y los objetos contenedores sean más adecuados para otras operaciones. En ActionScript 3.0, los valores simples y sus objetos contenedores son, para fines prácticos, indistinguibles. Todos los valores, incluso los valores simples, son objetos. El motor de ejecución trata estos tipos simples como casos especiales que se comportan como objetos pero que no requieren la sobrecarga normal asociada a la creación de objetos. Esto significa que las dos líneas de código siguientes son equivalentes:

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

Todos los tipos de datos simples y complejos antes descritos se definen en las clases principales de ActionScript 3.0. Las clases principales permiten crear objetos utilizando valores literales en lugar de utilizar el operador `new`. Por ejemplo, se puede crear un conjunto utilizando un valor literal o el constructor de la clase Array, de la manera siguiente:

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

Verificación de tipos

La verificación de tipos puede tener lugar al compilar o en tiempo de ejecución. Los lenguajes con tipos estáticos, como C++ y Java, realizan la verificación de tipos en tiempo de compilación. Los lenguajes con tipos dinámicos, como Smalltalk y Python, realizan la verificación de tipos en tiempo de ejecución. Al ser un lenguaje con tipos dinámicos, ActionScript 3.0 ofrece verificación de tipos en tiempo de ejecución, pero también admite verificación de tipos en tiempo de compilación con un modo de compilador especial denominado *modo estricto*. En modo estricto, la verificación de tipos se realiza en tiempo de compilación y en tiempo de ejecución; en cambio, en modo estándar la verificación de tipos sólo se realiza en tiempo de ejecución.

Los lenguajes con tipos dinámicos ofrecen una gran flexibilidad para estructurar el código, pero a costa de permitir que se produzcan errores de tipo en tiempo de ejecución. Los lenguajes con tipos estáticos notifican los errores de tipo en tiempo de compilación, pero a cambio deben conocer la información de tipos en tiempo de compilación.

Verificación de tipos en tiempo de compilación

La verificación de tipos en tiempo de compilación se suele favorecer en los proyectos grandes ya que, a medida que el tamaño de un proyecto crece, la flexibilidad de los tipos de datos suele ser menos importante que detectar los errores de tipo lo antes posible. Ésta es la razón por la que, de manera predeterminada, el compilador de ActionScript de Flash Professional y Flash Builder está configurado para ejecutarse en modo estricto.

Adobe Flash Builder

Para desactivar el modo estricto en Flash Builder, vaya a la configuración del compilador de ActionScript en el cuadro de diálogo Propiedades del proyecto.

Para poder proporcionar verificación de tipos en tiempo de compilación, el compilador necesita saber cuáles son los tipos de datos de las variables o las expresiones del código. Para declarar explícitamente un tipo de datos para una variable, se debe añadir el operador dos puntos (`:`) seguido del tipo de datos como sufijo del nombre de la variable. Para asociar un tipo de datos a un parámetro, se debe utilizar el operador dos puntos seguido del tipo de datos. Por ejemplo, el código siguiente añade la información de tipo de datos al parámetro `xParam` y declara una variable `myParam` con un tipo de datos explícito:

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

En modo estricto, el compilador de ActionScript notifica los tipos no coincidentes como errores de compilación. Por ejemplo, el código siguiente declara un parámetro de función `xParam`, de tipo Object, pero después intenta asignar valores de tipo String y Number a ese parámetro. Esto produce un error del compilador en modo estricto.

```
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);
    }
}
```

Sin embargo, incluso en modo estricto se puede evitar selectivamente la verificación de tipos en tiempo de compilación dejando sin tipo el lado derecho de una sentencia de asignación. También se puede marcar una variable o expresión como variable o expresión sin tipo, omitiendo una anotación de tipo o utilizando la anotación de tipo asterisco (`*`). Por ejemplo, si se modifica el parámetro `xParam` del ejemplo anterior de forma que ya no tenga una anotación de tipo, el código se compilará de modo estricto:

```
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");
```

Verificación de tipos en tiempo de ejecución

ActionScript 3.0 realiza la verificación de tipos en tiempo de ejecución tanto si se compila en modo estricto como si se compila en modo estándar. Considérese una situación en la que se pasa el valor 3 como argumento a una función que espera un conjunto. En modo estricto, el compilador generará un error, ya que el valor 3 no es compatible con el tipo de datos Array. Al desactivar el modo estricto y entrar en modo estándar, el compilador no notificará acerca de los tipos no coincidentes, pero la verificación de tipos en tiempo de ejecución realizada por el motor de ejecución dará lugar a un error en tiempo de ejecución.

En el siguiente ejemplo se muestra una función denominada `typeTest()` que espera un argumento de tipo Array pero recibe el valor 3. Esto provoca un error en tiempo de ejecución en modo estándar, ya que el valor 3 no es un miembro del tipo de datos (Array) declarado para el parámetro.

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

También puede haber situaciones en las que se produzca un error de tipo en tiempo de ejecución aunque se opere en modo estricto. Esto puede suceder si se usa el modo estricto pero se elige no verificar tipos en tiempo de compilación utilizando una variable sin tipo. Si se utiliza una variable sin tipo, no se elimina la verificación de tipos, sino que se aplaza hasta el tiempo de ejecución. Por ejemplo, si la variable `myNum` del ejemplo anterior no tiene un tipo de datos declarado, el compilador no podrá detectar los tipos no coincidentes, pero el código generará un error en tiempo de ejecución al comparar el valor en tiempo de ejecución de `myNum`, que está definido en 3 como resultado de la sentencia de asignación, con el tipo de `xParam`, que está definido como el tipo de datos Array.

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

La verificación de tipos en tiempo de ejecución también permite un uso más flexible de la herencia que la verificación en tiempo de compilación. Al aplazar la verificación de tipos al tiempo de ejecución, el modo estándar permite hacer referencia a propiedades de una subclase aunque se realice una *conversión hacia arriba*. Una conversión hacia arriba se produce si se utiliza una clase base para declarar el tipo de una instancia de la clase y una subclase para crear la instancia. Por ejemplo, se puede crear una clase denominada ClassBase ampliable (las clases con el atributo `final` no se pueden ampliar):

```
class ClassBase
{
}
```

Posteriormente se puede crear una subclase de ClassBase denominada ClassExtender, con una propiedad denominada `someString`, como se indica a continuación:

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

Se pueden usar ambas clases para crear una instancia de clase que se declara con el tipo de datos de ClassBase, pero se crea con el constructor de ClassExtender. Una conversión hacia arriba se considera una operación segura porque la clase base no contiene ninguna propiedad o método que no esté en la subclase.

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

No obstante, una subclase contiene propiedades o métodos que la clase base no contiene. Por ejemplo, la clase ClassExtender contiene la propiedad `someString`, que no existe en la clase ClassBase. En el modo estándar de ActionScript 3.0 se puede hacer referencia a esta propiedad mediante la instancia de `myClass` sin generar un error de tiempo de compilación, como se muestra en el siguiente ejemplo:

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

Operador is

El operador `is` permite comprobar si una variable o expresión es un miembro de un tipo de datos determinado. En versiones anteriores de ActionScript el operador `instanceof` proporcionaba esta funcionalidad, pero en ActionScript 3.0 no se debe utilizar el operador `instanceof` para comprobar la pertenencia a un tipo de datos. Para la verificación manual de tipos se debe utilizar el operador `is` en lugar del operador `instanceof`, ya que la expresión `x instanceof y` simplemente comprueba en la cadena de prototipos de `x` si existe `y` (y en ActionScript 3.0, la cadena de prototipos no proporciona una imagen completa de la jerarquía de herencia).

El operador `is` examina la jerarquía de herencia adecuada y se puede utilizar no sólo para verificar si un objeto es una instancia de una clase específica, sino también en el caso de que un objeto sea una instancia de una clase que implementa una interfaz determinada. En el siguiente ejemplo se crea una instancia de la clase Sprite denominada `mySprite` y se utiliza el operador `is` para comprobar si `mySprite` es una instancia de las clases Sprite y DisplayObject, y si implementa la interfaz IEventDispatcher.

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

El operador `is` comprueba la jerarquía de herencia y notifica que `mySprite` es compatible con las clases Sprite y DisplayObject (la clase Sprite es una subclase de la clase DisplayObject). El operador `is` también comprueba si `mySprite` hereda de alguna clase que implementa la interfaz IEventDispatcher. Como la clase Sprite hereda de la clase EventDispatcher, que implementa la interfaz IEventDispatcher, el operador `is` notifica correctamente que `mySprite` implementa la misma interfaz.

En el siguiente ejemplo se muestran las mismas pruebas del ejemplo anterior, pero con `instanceof` en lugar del operador `is`. El operador `instanceof` identifica correctamente que `mySprite` es una instancia de Sprite o DisplayObject, pero devuelve `false` cuando se usa para comprobar si `mySprite` implementa la interfaz IEventDispatcher.

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

Operador as

El operador `as` también permite comprobar si una expresión es un miembro de un tipo de datos determinado. Sin embargo, a diferencia del operador `is`, el operador `as` no devuelve un valor booleano. El operador `as` devuelve el valor de la expresión en lugar de `true` y `null` en lugar de `false`. En el siguiente ejemplo se muestran los resultados de utilizar el operador `as` en lugar del operador `is` en el caso sencillo de comprobar si una instancia de Sprite es un miembro de los tipos de datos DisplayObject, IEventDispatcher y 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
```

Al utilizar el operador `as`, el operando de la derecha debe ser un tipo de datos. Si se intenta utilizar una expresión que no sea un tipo de datos como operando de la derecha se producirá un error.

Clases dinámicas

Una clase *dinámica* define un objeto que se puede modificar en tiempo de ejecución añadiendo o modificando propiedades y métodos. Una clase que no es dinámica, como la clase String, es una clase *cerrada*. No es posible añadir propiedades o métodos a una clase cerrada en tiempo de ejecución.

Para crear clases dinámicas se utiliza el atributo `dynamic` al declarar una clase. Por ejemplo, el código siguiente crea una clase dinámica denominada `Protean`:

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

Si posteriormente se crea una instancia de la clase `Protean`, se pueden añadir propiedades o métodos fuera de la definición de clase. Por ejemplo, el código siguiente crea una instancia de la clase `Protean` y añade una propiedad denominada `aString` y otra propiedad denominada `aNumber` a la instancia:

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

Las propiedades que se añaden a una instancia de una clase dinámica son entidades de tiempo de ejecución, por lo que no se realiza ninguna verificación de tipos en tiempo de ejecución. No se puede añadir una anotación de tipo a una propiedad añadida de esta manera.

También se puede añadir un método a la instancia de `myProtean` definiendo una función y asociando la función a una propiedad de la instancia de `myProtean`. El código siguiente mueve la sentencia trace a un método denominado `traceProtean()`:

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

Sin embargo, los métodos creados de esta manera no tienen acceso a ninguna propiedad o método privado de la clase Protean. Además, incluso las referencias a las propiedades o métodos públicos de la clase `Protean` deben calificarse con la palabra clave `this` o el nombre de la clase. En el siguiente ejemplo se muestra el intento de acceso del método `traceProtean()` a las variables privadas y las variables públicas de la clase `Protean`.

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

Descripción de los tipos de datos

Los tipos de datos simples son Boolean, int, Null, Number, String, uint y void. Las clases principales de ActionScript también definen los tipos de datos complejos siguientes: Object, Array, Date, Error, Function, RegExp, XML y XMLList.

Tipo de datos Boolean

El tipo de datos Boolean incluye dos valores: `true` y `false`. Ningún otro valor es válido para variables de tipo Boolean. El valor predeterminado de una variable booleana declarada pero no inicializada es `false`.

Tipo de datos int

El tipo de datos int se almacena internamente como un entero de 32 bits y consta del conjunto de enteros entre

-2.147.483.648 (-2^{31}) a 2.147.483,647 ($2^{31} - 1$), ambos incluidos. Las versiones anteriores de ActionScript sólo ofrecían el tipo de datos Number, que se usaba tanto para enteros como para números de coma flotante. En ActionScript 3.0 se tiene acceso a tipos de bajo nivel para enteros de 32 bits con o sin signo. Si la variable no va a usar números de coma flotante, es más rápido y eficaz utilizar el tipo de datos int en lugar del tipo de datos Number.

Para valores enteros que estén fuera del rango de los valores enteros mínimo y máximo, hay que utilizar el tipo de datos Number, que admite valores entre los valores positivo y negativo de 9.007.199.254.740.992 (valores enteros de 53 bits). El valor predeterminado para variables con tipo de datos int es 0.

Tipo de datos Null

El tipo de datos Null (nulo) tiene un único valor: `null`. Éste es el valor predeterminado para el tipo de datos String y para todas las clases que definen tipos de datos complejos, incluida la clase Object. Ninguno de los demás tipos de datos simples, como Boolean, Number, int y uint, contienen el valor `null`. En tiempo de ejecución, el valor `null` se convierte en el valor predeterminado adecuado si se intenta asignar `null` a las variables de tipo Boolean, Number, int o uint. Este tipo de datos no se puede utilizar como una anotación de tipo.

Tipo de datos Number

En ActionScript 3.0, el tipo de datos Number representa enteros, enteros sin signo y números de coma flotante. Sin embargo, para maximizar el rendimiento se recomienda utilizar el tipo de datos Number únicamente para valores enteros que ocupen más que los 32 bits que pueden almacenar los tipos de datos `int` y `uint` o para números de coma flotante. Para almacenar un número de coma flotante se debe incluir una coma decimal en el número. Si se omite un separador decimal, el número se almacenará como un entero.

El tipo de datos Number utiliza el formato de doble precisión de 64 bits especificado en la norma IEEE para aritmética binaria de coma flotante (IEEE-754). Esta norma especifica cómo se almacenan los números de coma flotante utilizando los 64 bits disponibles. Se utiliza un bit para designar si el número es positivo o negativo. El exponente, que se almacena como un número de base 2, utiliza once bits. Los 52 bits restantes se utilizan para almacenar la *cifra significativa* (también denominada *mantisa*), que es el número que se eleva a la potencia indicada por el exponente.

Al utilizar parte de los bits para almacenar un exponente, el tipo de datos Number puede almacenar números de coma flotante considerablemente más grandes que si se utilizaran todos los bits para la mantisa. Por ejemplo, si el tipo de datos Number utilizara los 64 bits para almacenar la mantisa, podría almacenar números hasta 2^{65} -1. Al utilizar 11 bits para almacenar un exponente, el tipo de datos Number puede elevar la mantisa a una potencia de 2^{1023}.

Los valores máximo y mínimo que el tipo Number puede representar se almacenan en propiedades estáticas de la clase Number denominadas `Number.MAX_VALUE` y `Number.MIN_VALUE`.

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

Aunque este rango de números es enorme, se pierde en precisión. El tipo de datos Number utiliza 52 bits para almacenar la mantisa y, como consecuencia, los números que requieren más de 52 bits para una representación precisa, como la fracción 1/3, son sólo aproximaciones. Si la aplicación requiere precisión absoluta con números decimales, hay que utilizar software que implemente aritmética decimal de coma flotante en lugar de aritmética binaria de coma flotante.

Al almacenar valores enteros con el tipo de datos Number, sólo se utilizarán los 52 bits de la mantisa. El tipo de datos Number utiliza estos 52 bits y un bit oculto especial para representar enteros entre -9.007.199.254.740.992 (-2^{53}) y 9.007.199.254.740.992 (2^{53}).

El valor `NaN` no sólo se utiliza como valor predeterminado para las variables de tipo `Number`, sino también como resultado de cualquier operación que debiera devolver un número y no lo haga. Por ejemplo, si se intenta calcular la raíz cuadrada de un número negativo, el resultado será `NaN`. Otros valores especiales de Number son *infinito positivo* e *infinito negativo*.

Nota: *el resultado de la división por* `0` *sólo es* `NaN` *si el divisor también es* `0`*. La división por* `0` *produce* `infinity` *cuando el dividendo es positivo o* `-infinity` *cuando el dividendo es negativo.*

Tipo de datos String

El tipo de datos String representa una secuencia de caracteres de 16 bits. Las cadenas se almacenan internamente como caracteres Unicode empleando el formato UTF-16. Las cadenas son valores inmutables, igual que en el lenguaje de programación Java. Una operación sobre un valor de cadena (String) devuelve una nueva instancia de la cadena. El valor predeterminado de una variable declarada con el tipo de datos String es `null`. El valor `null` no es lo mismo que la cadena vacía (`""`). El valor `null` indica que la variable no tiene ningún valor almacenado, mientras que la cadena vacía significa que la variable tiene un valor que es una cadena sin caracteres.

Tipo de datos uint

El tipo de datos uint se almacena internamente como un entero sin signo de 32 bits y consta del conjunto de enteros entre 0 y 4.294.967.295 (2^{32}- 1), ambos incluidos. El tipo de datos uint debe utilizarse en circunstancias especiales que requieran enteros no negativos. Por ejemplo, se debe utilizar el tipo de datos uint para representar valores de colores de píxeles, ya que el tipo de datos int tiene un bit de signo interno que no es apropiado para procesar valores de colores. Para valores enteros más grandes que el valor uint máximo, se debe utilizar el tipo de datos Number, que puede procesar valores enteros de 53 bits. El valor predeterminado para variables con tipo de datos uint es 0.

Tipo de datos Void

El tipo de datos void tiene un único valor: `undefined`. En las versiones anteriores de ActionScript, `undefined` era el valor predeterminado de las instancias de la clase Object. En ActionScript 3.0, el valor predeterminado de las instancias de Object es `null`. Si se intenta asignar el valor `undefined` a una instancia del objeto Object, dicho valor se convierte en `null`. Sólo se puede asignar un valor `undefined` a variables que no tienen tipo. Las variables sin tipo son variables que no tienen anotación de tipo o utilizan el símbolo de asterisco (`*`) como anotación de tipo. Sólo se puede usar `void` como anotación de tipo devuelto.

Tipo de datos Object

El tipo de datos Object (objeto) se define mediante la clase Object. La clase Object constituye la clase base para todas las definiciones de clase en ActionScript. La versión del tipo de datos Object en ActionScript 3.0 difiere de la de versiones anteriores en tres aspectos. En primer lugar, el tipo de datos Object ya no es el tipo de datos predeterminado que se asigna a las variables sin anotación de tipo. En segundo lugar, el tipo de datos Object ya no incluye el valor `undefined` que se utilizaba como valor predeterminado de las instancias de Object. Por último, en ActionScript 3.0, el valor predeterminado de las instancias de la clase Object es `null`.

En versiones anteriores de ActionScript, a una variable sin anotación de tipo se le asignaba automáticamente el tipo de datos Object. Esto ya no es así en ActionScript 3.0, que incluye el concepto de variable sin tipo. Ahora se considera que las variables sin anotación de tipo no tienen tipo. Si se prefiere dejar claro a los lectores del código que la intención es dejar una variable sin tipo, se puede utilizar el símbolo de asterisco (`*`) para la anotación de tipo, que equivale a omitir una anotación de tipo. En el siguiente ejemplo se muestran dos sentencias equivalentes que declaran una variable sin tipo `x`:

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

Sólo las variables sin tipo pueden contener el valor `undefined`. Si se intenta asignar el valor `undefined` a una variable que tiene un tipo de datos, el motor de ejecución convertirá el valor `undefined` en el valor predeterminado de dicho tipo de datos. Para las instancias del tipo de datos Object, el valor predeterminado es `null`, lo que significa que si se intenta asignar `undefined` a una instancia de Object, el valor se convierte en `null`.

Conversiones de tipos

Se dice que se produce una conversión de tipo cuando se transforma un valor en otro valor con un tipo de datos distinto. Las conversiones de tipo pueden ser *implícitas* o *explícitas*. La conversión implícita, también denominada *coerción*, se realiza en ocasiones en tiempo de ejecución. Por ejemplo, si a una variable del tipo de datos Boolean se le asigna el valor 2, este valor se convierte en el valor booleano `true` antes de asignar el valor a la variable. La conversión explícita, también denominada *conversión*, se produce cuando el código ordena al compilador que trate una variable de un tipo de datos como si perteneciera a un tipo de datos distinto. Si se usan valores simples, la conversión convierte realmente los valores de un tipo de datos a otro. Para convertir un objeto a otro tipo, hay que incluir el nombre del objeto entre paréntesis y anteponerle el nombre del nuevo tipo. Por ejemplo, el siguiente código toma un valor booleano y lo convierte en un entero:

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

Conversiones implícitas

Las conversiones implícitas se realizan en tiempo de ejecución en algunos contextos:

- En sentencias de asignación
- Cuando se pasan valores como argumentos de función
- Cuando se devuelven valores desde funciones
- En expresiones que utilizan determinados operadores, como el operador suma (+)

  Para tipos definidos por el usuario, las conversiones implícitas se realizan correctamente cuando el valor que se va a convertir es una instancia de la clase de destino o una clase derivada de la clase de destino. Si una conversión implícita no se realiza correctamente, se producirá un error. Por ejemplo, el código siguiente contiene una conversión implícita correcta y otra incorrecta:

  ```
  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.
  ```

  Para tipos simples, las conversiones implícitas se realizan llamando a los mismos algoritmos internos de conversión que utilizan las funciones de conversión explícita.

Conversiones explícitas

Resulta útil usar conversiones explícitas cuando se compila en modo estricto, ya que a veces no se desea que una discordancia de tipos genere un error en tiempo de compilación. Esto puede ocurrir, por ejemplo, cuando se sabe que la coerción convertirá los valores correctamente en tiempo de ejecución. Por ejemplo, al trabajar con datos recibidos desde un formulario, puede ser interesante basarse en la coerción para convertir determinados valores de cadena en valores numéricos. El código siguiente genera un error de tiempo de compilación aunque se ejecuta correctamente en modo estándar:

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

Si se desea seguir utilizando el modo estricto pero se quiere convertir la cadena en un entero, se puede utilizar la conversión explícita de la manera siguiente:

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

Conversión a int, uint y Number

Cualquier tipo de datos se puede convertir en uno de los tres tipos numéricos siguientes: int, uint y Number. Si el número no puede convertirse por algún motivo, el valor predeterminado de 0 se asigna para los tipos de datos int y uint y el valor predeterminado de `NaN` se asigna para el tipo de datos Number. Si se convierte un valor booleano a un número, `true` se convierte en el valor 1 y `false` se convierte en el valor 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
```

Los valores de cadena que sólo contienen dígitos pueden convertirse correctamente en uno de los tipos numéricos. Los tipos numéricos también pueden convertir cadenas que parecen números negativos o cadenas que representan un valor hexadecimal (por ejemplo, `0x1A`). El proceso de conversión omite los caracteres de espacio en blanco iniciales y finales del valor de cadena. También se puede convertir cadenas que parecen números de coma flotante mediante `Number()`. La inclusión de un separador decimal hace que `uint()` e `int()` devuelvan un entero, truncando el separador decimal y los caracteres que siguen. Por ejemplo, los siguientes valores de cadena pueden convertirse en números:

```
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
```

Los valores de cadena que contienen caracteres no numéricos devuelven 0 cuando se convierten con `int()` o `uint()`, y `NaN` cuando se convierten con`Number()`. El proceso de conversión omite el espacio en blanco inicial y final, pero devuelve 0 o `NaN` si una cadena contiene espacio en blanco separando dos números.

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

En ActionScript 3.0 la función `Number()` ya no admite números octales (de base 8). Si se suministra una cadena con un cero inicial a la función `Number()` de ActionScript 2.0, el número se interpreta como un número octal y se convierte en su equivalente decimal. Esto no es así con la función `Number()` de ActionScript 3.0, que omite el cero inicial. Por ejemplo, el código siguiente genera resultados distintos cuando se compila con versiones distintas de ActionScript:

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

La conversión no es necesaria cuando se asigna un valor de un tipo numérico a una variable de un tipo numérico distinto. Incluso en modo estricto, los tipos numéricos se convierten implícitamente a los otros tipos numéricos. Esto significa que en algunos casos pueden producirse valores inesperados cuando se supera el rango de un tipo. Todos los ejemplos siguientes se compilan en modo estricto, aunque algunos generarán valores inesperados:

```
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
```

En la tabla siguiente se resumen los resultados de convertir a los tipos de datos Number, int o uint desde otros tipos de datos.

Tipo de datos o valor	Resultado de la conversión a Number, int o uint
Boolean	Si el valor es `true`, 1; de lo contrario, 0.
Date	Representación interna del objeto Date, que es el número de milisegundos transcurridos desde la medianoche del 1 de enero de 1970, hora universal.
`null`	0
Object	Si la instancia es `null` y se convierte a Number, `NaN`; de lo contrario, 0.
String	Un número si la cadena se puede convertir en uno; de lo contrario, `NaN` si se convierte en Number o 0 si se convierte en int o uint.
`undefined`	Si se convierte a Number, `NaN`; si se convierte a int o uint, 0.

Conversión a Boolean

La conversión a Boolean desde cualquiera de los tipos de datos numéricos (uint, int y Number) produce `false` si el valor numérico es 0 y `true` en caso contrario. Para el tipo de datos Number, el valor `NaN` también produce `false`. En el siguiente ejemplo se muestran los resultados de convertir los números -1, 0 y 1:

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

El resultado del ejemplo muestra que de los tres números, sólo 0 devuelve un valor `false`:

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

La conversión a Boolean desde un valor String devuelve `false` si la cadena es `null` o una cadena vacía (`""`). De lo contrario, devuelve `true`.

```
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
```

La conversión a Boolean desde una instancia de la clase Object devuelve `false` si la instancia es `null` y `true` en caso contrario:

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

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

En modo estricto, las variables Boolean reciben un tratamiento especial en el sentido de que se puede asignar valores de cualquier tipo de datos a una variable Boolean sin realizar una conversión. La coerción implícita desde todos los tipos de datos al tipo de datos Boolean se produce incluso en modo estricto. Es decir, a diferencia de lo que ocurre para casi todos los demás tipos de datos, la conversión a Boolean no es necesaria para evitar errores en modo estricto. Todos los ejemplos siguientes se compilan en modo estricto y se comportan de la manera esperada en tiempo de ejecución:

```
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
```

En la tabla siguiente se resumen los resultados de convertir al tipo de datos Boolean desde otros tipos de datos:

Tipo de datos o valor	Resultado de la conversión a Boolean
String	`false` si el valor es `null` o la cadena vacía (`""`); `true` en caso contrario.
`null`	`false`
Number, int o uint	`false` si el valor es `NaN` o 0; `true` en caso contrario.
Object	`false` si la instancia es `null`; `true` en caso contrario.

Conversión a String

La conversión al tipo de datos String desde cualquiera de los tipos de datos numéricos devuelve una representación del número como una cadena. La conversión al tipo de datos String desde un valor booleano devuelve la cadena `"true"` si el valor es `true` y devuelve la cadena `"false"` si el valor es `false`.

La conversión al tipo de datos String desde una instancia de la clase Object devuelve la cadena `"null"` si la instancia es `null`. De lo contrario, la conversión al tipo String de la clase Object devuelve la cadena `"[object Object]"`.

La conversión a String desde una instancia de la clase Array devuelve una cadena que consta de una lista delimitada por comas de todos los elementos del conjunto. Por ejemplo, la siguiente conversión al tipo de datos String devuelve una cadena que contiene los tres elementos del conjunto:

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

La conversión a String desde una instancia de la clase Date devuelve una representación de cadena de la fecha que contiene la instancia. Por ejemplo, el ejemplo siguiente devuelve una representación de cadena de la instancia de la clase Date (la salida muestra el resultado para el horario de verano de la costa del Pacífico de EE.UU.):

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

En la tabla siguiente se resumen los resultados de convertir al tipo de datos String desde otros tipos de datos.

Tipo de datos o valor	Resultado de la conversión a String
Array	Cadena que consta de todos los elementos de conjunto.
Boolean	`"true"` o `"false"`
Date	Una representación de cadena del objeto Date.
`null`	`"null"`
Number, int o uint	Una representación de cadena del número.
Object	Si la instancia es null, `"null"`; de lo contrario, `"[object Object]"`.

Sintaxis

La sintaxis de un lenguaje define un conjunto de reglas que deben cumplirse al escribir código ejecutable.

Distinción entre mayúsculas y minúsculas

El lenguaje ActionScript 3.0 distingue mayúsculas de minúsculas. Los identificadores que sólo se diferencien en mayúsculas o minúsculas se considerarán identificadores distintos. Por ejemplo, el código siguiente crea dos variables distintas:

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

Sintaxis con punto

El operador de punto (`.`) permite acceder a las propiedades y los métodos de un objeto. La sintaxis con punto permite hacer referencia a una propiedad o un método de clase mediante un nombre de instancia, seguido del operador punto y el nombre de la propiedad o el método. Por ejemplo, considere la siguiente definición de clase:

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

La sintaxis con punto permite acceder a la propiedad `prop1` y al método `method1()` utilizando el nombre de la instancia creada en el código siguiente:

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

Se puede utilizar la sintaxis con punto al definir paquetes. El operador punto se utiliza para hacer referencia a paquetes anidados. Por ejemplo, la clase EventDispatcher reside en un paquete denominado events que está anidado dentro del paquete denominado flash. Se puede hacer referencia al paquete events mediante la siguiente expresión:

```
flash.events
```

También se puede hacer referencia a la clase EventDispatcher mediante esta expresión:

```
flash.events.EventDispatcher
```

Sintaxis con barras diagonales

ActionScript 3.0 no admite la sintaxis con barras diagonales. Esta sintaxis se utilizaba en versiones anteriores de ActionScript para indicar la ruta a un clip de película o una variable.

Literales

Un *literal* es un valor que aparece directamente en el código. Todos los ejemplos siguientes son literales:

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

Los literales también pueden agruparse para formar literales compuestos. Los literales de conjunto se escriben entre corchetes (`[]`) y utilizan la coma para separar los elementos de conjunto.

Un literal de conjunto puede utilizarse para inicializar un conjunto. En los siguientes ejemplos se muestran dos conjuntos que se inicializan mediante literales de conjunto. Se puede utilizar la sentencia `new` y pasar el literal compuesto como parámetro al constructor de la clase Array, pero también se pueden asignar valores literales directamente al crear instancias de las siguientes clases principales de ActionScript: Object, Array, String, Number, int, uint, XML, XMLList y 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];
```

Los literales también se pueden utilizar para inicializar un objeto genérico. Un objeto genérico es una instancia de la clase Object. Los literales de objetos se escriben entre llaves (`{}`) y utilizan la coma para separar las propiedades del objeto. Cada propiedad se declara mediante el signo de dos puntos (`:`), que separa el nombre de la propiedad del valor de la propiedad.

Se puede crear un objeto genérico utilizando la sentencia `new` y pasar el literal de objeto como parámetro al constructor de la clase Object, o bien asignar el literal de objeto directamente a la instancia que se está declarando. En el siguiente ejemplo se muestran dos formas de crear un nuevo objeto genérico y se inicializa el objeto con tres propiedades (`propA`, `propB` y `propC`) establecidas en los valores 1, 2 y 3 respectivamente:

```
// 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};
```

Más temas de ayuda

Trabajo con cadenas

Uso de expresiones regulares

Inicialización de variables XML

Signos de punto y coma

Se puede utilizar el signo de punto y coma (`;`) para finalizar una sentencia. Como alternativa, si se omite el signo de punto y coma, el compilador dará por hecho que cada línea de código representa a una sentencia independiente. Como muchos programadores están acostumbrados a utilizar el signo de punto y coma para indicar el final de una sentencia, el código puede ser más legible si se usan siempre signos de punto y coma para finalizar las sentencias.

El uso del punto y coma para terminar una sentencia permite colocar más de una sentencia en una misma línea, pero esto hará que el código resulte más difícil de leer.

Paréntesis

Los paréntesis (`()`) se pueden utilizar de tres modos diferentes en ActionScript 3.0. En primer lugar, se pueden utilizar para cambiar el orden de las operaciones de una expresión. Las operaciones agrupadas entre paréntesis siempre se ejecutan primero. Por ejemplo, se utilizan paréntesis para modificar el orden de las operaciones en el código siguiente:

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

En segundo lugar, se pueden utilizar paréntesis con el operador coma (`,`) para evaluar una serie de expresiones y devolver el resultado de la expresión final, como se indica en el siguiente ejemplo:

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

Por último, se pueden utilizar paréntesis para pasar uno o más parámetros a funciones o métodos, como se indica en el siguiente ejemplo, que pasa un valor String a la función `trace()`:

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

Comentarios

El código de ActionScript 3.0 admite dos tipos de comentarios: comentarios de una sola línea y comentarios multilínea. Estos mecanismos para escribir comentarios son similares a los equivalentes de C++ y Java. El compilador omitirá el texto marcado como un comentario.

Los comentarios de una sola línea empiezan por dos caracteres de barra diagonal (`//`) y continúan hasta el final de la línea. Por ejemplo, el código siguiente contiene un comentario de una sola línea:

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

Los comentarios multilínea empiezan con una barra diagonal y un asterisco (`/*`) y terminan con un asterisco y una barra diagonal (`*/`).

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

Palabras clave y palabras reservadas

Las palabras reservadas son aquellas que no se pueden utilizar como identificadores en el código porque su uso está reservado para ActionScript. Incluyen las *palabras clave léxicas*, que son eliminadas del espacio de nombres del programa por el compilador. El compilador notificará un error si se utiliza una palabra clave léxica como un identificador. En la tabla siguiente se muestran las palabras clave léxicas de 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
with			

Hay un pequeño conjunto de palabras clave, denominadas *palabras clave sintácticas*, que se pueden utilizar como identificadores, pero que tienen un significado especial en determinados contextos. En la tabla siguiente se muestran las palabras clave sintácticas de ActionScript 3.0.

each	get	set	namespace
include	dynamic	final	native
override	static		

También hay varios identificadores que a veces se llaman *futuras palabras reservadas*. ActionScript 3.0 no reserva estos identificadores, aunque el software que incorpore ActionScript 3.0 podrá tratar algunos de ellos como palabras clave. Es posible que pueda utilizar un gran número de estos identificadores en su código, pero Adobe recomienda que no se utilicen ya que pueden aparecer como palabras clave en una versión posterior de dicho lenguaje.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic

long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

Constantes

ActionScript 3.0 admite la sentencia `const`, que se puede utilizar para crear constantes. Las constantes son propiedades con un valor fijo que no se puede modificar. Se puede asignar un valor a una constante una sola vez y la asignación debe realizarse cerca de la declaración de la constante. Por ejemplo, si se declara una constante como un miembro de una clase, se puede asignar un valor a esa constante únicamente como parte de la declaración o dentro del constructor de la clase.

El código siguiente declara dos constantes. Se asigna un valor a la primera constante, `MINIMUM`, como parte de la sentencia de declaración. A la segunda constante, `MAXIMUM`, se le asigna un valor en el constructor. Es necesario tener en cuenta que este ejemplo sólo compila en modo estándar, ya que el modo estricto sólo permite asignar un valor de constante en tiempo de inicialización.

```
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 producirá un error si se intenta asignar de otra manera un valor inicial a una constante. Por ejemplo, si se intenta establecer el valor inicial de `MAXIMUM` fuera de la clase, se producirá un error en tiempo de ejecución.

```
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 define una amplia gama de constantes para su uso. Por convención, en ActionScript las constantes se escriben en mayúsculas y las palabras que las forman se separan mediante el carácter de subrayado (_). Por ejemplo, la definición de la clase MouseEvent utiliza esta convención de nomenclatura para sus constantes, cada una de las cuales representa un evento relacionado con una entrada del ratón:

```
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";
    ...
    }
}
```

Operadores

Los operadores son funciones especiales que se aplican a uno o más operandos y devuelven un valor. Un *operando* es un valor (generalmente un literal, una variable o una expresión) que se usa como entrada de un operador. Por ejemplo, en el código siguiente, los operadores de suma (`+`) y multiplicación (`*`) se usan con tres operandos literales (2, 3 y 4) para devolver un valor. A continuación, el operador de asignación (`=`) usa este valor para asignar el valor devuelto, 14, a la variable `sumNumber`.

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

Los operadores pueden ser unarios, binarios o ternarios. Un operador *unario* se aplica a un operando. Por ejemplo, el operador de incremento (`++`) es un operador unario porque se aplica a un solo operando. Un operador *binario* se aplica a dos operandos. Por ejemplo, el operador división (`/`) se aplica a dos operandos. Un operador *ternario* se aplica a tres operandos. Por ejemplo, el operador condicional (`?:`) se aplica a tres operandos.

Algunos operadores están *sobrecargados*, lo que significa que se comportan de distinta manera en función del tipo o la cantidad de operandos que se les pase. El operador suma (`+`) es un ejemplo de un operador sobrecargado que se comporta de distinta manera en función del tipo de datos de los operandos. Si ambos operandos son números, el operador suma devuelve la suma de los valores. Si ambos operandos son cadenas, el operador suma devuelve la concatenación de los dos operandos. En el siguiente ejemplo de código se muestra cómo cambia el comportamiento del operador en función de los operandos:

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

Los operadores también pueden comportarse de distintas maneras en función del número de operandos suministrados. El operador resta (`-`) es la vez un operador unario y un operador binario. Si se le suministra un solo operando, el operador resta devuelve como resultado la negación del operando. Si se le suministran dos operandos, el operador resta devuelve la diferencia de los operandos. En el siguiente ejemplo se muestra el operador resta usado primero como un operador unario y después como un operador binario.

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

Precedencia y asociatividad de operadores

La precedencia y asociatividad de los operadores determina el orden en que se procesan los operadores. Aunque para aquellos usuarios familiarizados con la programación aritmética puede parecer natural que el compilador procese el operador de multiplicación (`*`) antes que el operador de suma (`+`), el compilador necesita instrucciones explícitas sobre qué operadores debe procesar primero. Dichas instrucciones se conocen colectivamente como *precedencia de operadores*. ActionScript establece una precedencia de operadores predeterminada que se puede modificar utilizando el operador paréntesis (`()`). Por ejemplo, el código siguiente modifica la precedencia predeterminada del ejemplo anterior para forzar al compilador a procesar el operador suma antes que el operador producto:

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

Pueden darse situaciones en las que dos o más operadores con la misma precedencia aparezcan en la misma expresión. En estos casos, el compilador utiliza las reglas de *asociatividad* para determinar qué operador se procesa primero. Todos los operadores binarios, salvo los operadores de asignación, tienen *asociatividad desde la izquierda*, lo que significa que los operadores de la izquierda se procesan antes que los operadores de la derecha. Los operadores de asignación y el operador condicional (`?:`) tienen *asociatividad desde la derecha*, lo que significa que los operadores de la derecha se procesan antes que los operadores de la izquierda.

Consideremos, por ejemplo, los operadores menor que (`<`) y mayor que (`>`), que tienen la misma precedencia. Si ambos operadores se utilizan en la misma expresión, el operador de la izquierda se procesará en primer lugar porque ambos operadores tienen asociatividad desde la izquierda. Esto significa que las dos sentencias siguientes generan el mismo resultado:

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

El operador mayor que se procesa primero, lo que devuelve un valor `true`, ya que el operando 3 es mayor que el operando 2. A continuación, se pasa el valor `true` al operador menor que, junto con el operando 1. El código siguiente representa este estado intermedio:

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

El operador menor que convierte el valor `true` en el valor numérico 1 y compara dicho valor numérico con el segundo operando 1 para devolver el valor `false` (el valor 1 no es menor que 1).

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

Se puede modificar la asociatividad predeterminada desde la izquierda con el operador paréntesis. Se puede ordenar al compilador que procese primero el operador menor que escribiendo dicho operador y sus operandos entre paréntesis. En el ejemplo siguiente se utiliza el operador paréntesis para producir un resultado diferente utilizando los mismos números que en el ejemplo anterior:

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

El operador menor que se procesa primero, lo que devuelve un valor `false`, ya que el operando 2 no es menor que el operando 1. A continuación, se pasa el valor `false` al operador mayor que, junto con el operando 3. El código siguiente representa este estado intermedio:

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

El operador mayor que convierte el valor `false` en el valor numérico 0 y compara dicho valor numérico con el otro operando 3 para devolver el valor `true` (el valor 3 es mayor que 0).

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

En la tabla siguiente se muestran los operadores de ActionScript 3.0 por orden decreciente de precedencia. Cada fila de la tabla contiene operadores de la misma precedencia. Cada fila de operadores tiene precedencia superior a la fila que aparece debajo de ella en la tabla.

Grupo	Operadores
Primario	[] {x:y} () f(x) new x.y x[y] <></> @ :: ..
Sufijo	x++ x--
Unario	++x --x + - ~ ! delete typeof void
Multiplicativo	* / %
Aditivo	+ -
Desplazamiento en modo bit	<< >> >>>
Relacional	< > <= >= como en instanceof es
Igualdad	== != === !==
AND en modo bit	&
XOR en modo bit	^
OR en modo bit	|
AND lógico	&&
OR lógico	||
Condicional	?:
Asignación	= *= /= %= += -= <<= >>= >>>= &= ^= |=
Coma	,

Operadores principales

Los operadores principales incluyen los que se utilizan para crear literales Array y Object, agrupar expresiones, llamar a funciones, crear instancias de clase y acceder a propiedades.

Todos los operadores principales, indicados en la tabla siguiente, tienen la misma precedencia. Los operadores que forman parte de la especificación E4X se indican mediante la notación (E4X).

Operador	Operación realizada
[]	Inicializa un conjunto
{x:y}	Inicializa un objeto
()	Agrupa expresiones
f(x)	Llama a una función
new	Llama a un constructor
x.y x[y]	Accede a una propiedad
<></>	Inicializa un objeto XMLList (E4X)
@	Accede a un atributo (E4X)
::	Califica un nombre (E4X)
..	Accede a un elemento XML descendiente (E4X)

Operadores de sufijo

Los operadores de sufijo se aplican a un operador para aumentar o reducir el valor. Aunque estos operadores son unarios, se clasifican por separado del resto de los operadores unarios debido a su mayor precedencia y a su comportamiento especial. Al utilizar un operador de sufijo como parte de una expresión mayor, el valor de la expresión se devuelve antes de que se procese el operador de sufijo. Por ejemplo, el siguiente código muestra cómo se devuelve el valor de la expresión `xNum++` antes de que se incremente el valor:

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

Todos los operadores de sufijo, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	**Operación realizada**
`++`	Incremento (sufijo)
`--`	Decremento (sufijo)

Operadores unarios

Los operadores unarios se aplican a un operando. Los operadores de incremento (`++`) y decremento (`--`) de este grupo son *operadores* de *prefijo*, lo que significa que aparecen delante del operando en una expresión. Los operadores de prefijo difieren de los correspondientes operadores de sufijo en que la operación de incremento o decremento se realiza antes de que se devuelva el valor de la expresión global. Por ejemplo, el siguiente código muestra cómo se devuelve el valor de la expresión `++xNum` después de que se incremente el valor:

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

Todos los operadores unarios, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	**Operación realizada**
`++`	Incremento (prefijo)
`--`	Decremento (prefijo)
`+`	Unario +
`-`	Unario - (negación)
`!`	NOT lógico
`~`	NOT en modo bit
`delete`	Elimina una propiedad
`typeof`	Devuelve información de tipo
`void`	Devuelve un valor no definido

Operadores multiplicativos

Los operadores multiplicativos toman dos operandos y realizan cálculos de multiplicación, división o módulo.

Todos los operadores multiplicativos, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	Operación realizada
*	Multiplicación
/	División
%	Módulo

Operadores aditivos

Los operadores aditivos se aplican a dos operandos y realizan cálculos de suma y resta. Todos los operadores aditivos, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	Operación realizada
+	Suma
-	Resta

Operadores de desplazamiento en modo bit

Los operadores de desplazamiento en modo bit se aplican a dos operandos y desplazan los bits del primer operando según lo especificado por el segundo operando. Todos los operadores de desplazamiento en modo de bit, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	Operación realizada
<<	Desplazamiento a la izquierda en modo bit
>>	Desplazamiento a la derecha en modo bit
>>>	Desplazamiento a la derecha en modo bit sin signo

Operadores relacionales

Los operadores relacionales se aplican a dos operandos, comparan sus valores y devuelven un valor booleano. Todos los operadores relacionales, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	Operación realizada
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
as	Comprueba el tipo de datos
in	Comprueba las propiedades de objetos
instanceof	Comprueba una cadena de prototipos
is	Comprueba el tipo de datos

Operadores de igualdad

Los operadores de igualdad se aplican a dos operandos, comparan sus valores y devuelven un valor booleano. Todos los operadores de igualdad, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	Operación realizada
`==`	Igualdad
`!=`	Desigualdad
`===`	Igualdad estricta
`!==`	Desigualdad estricta

Operadores lógicos en modo bit

Los operadores lógicos en modo bit se aplican a dos operandos y realizan operaciones lógicas a nivel de bits. Estos operadores, que tienen una precedencia diferente, se enumeran en la tabla siguiente por orden decreciente de precedencia:

Operador	Operación realizada
`&`	AND en modo bit
`^`	XOR en modo bit
`|`	OR en modo bit

Operadores lógicos

Los operadores lógicos se aplican a dos operandos y devuelven un resultado booleano. Estos operadores, que tienen distintas precedencias, se enumeran en la tabla siguiente por orden decreciente de precedencia:

Operador	Operación realizada
`&&`	AND lógico
`||`	OR lógico

Operador condicional

El operador condicional es un operador ternario, lo que significa que se aplica a tres operandos. El operador condicional es un método abreviado para aplicar la sentencia condicional `if..else`.

Operador	Operación realizada
`?:`	Condicional

Operadores de asignación

Los operadores de asignación se aplican a dos operandos y asignan un valor a un operando en función del valor del otro operando. Todos los operadores de asignación, indicados en la tabla siguiente, tienen la misma precedencia:

Operador	Operación realizada
`=`	Asignación
`*=`	Asignación de multiplicación
`/=`	Asignación de división
`%=`	Asignación de módulo
`+=`	Asignación de suma
`-=`	Asignación de resta
`<<=`	Asignación de desplazamiento a la izquierda en modo bit
`>>=`	Asignación de desplazamiento a la derecha en modo bit
`>>>=`	Asignación de desplazamiento a la derecha en modo bit sin signo
`&=`	Asignación de AND en modo bit
`^=`	Asignación de XOR en modo bit
`|=`	Asignación de OR en modo bit

Condicionales

ActionScript 3.0 proporciona tres sentencias condicionales básicas que se pueden usar para controlar el flujo del programa.

if..else

La sentencia condicional `if..else` permite comprobar una condición y ejecutar un bloque de código si dicha condición existe, o ejecutar un bloque de código alternativo si la condición no existe. Por ejemplo, el siguiente fragmento de código comprueba si el valor de `x` es superior a 20 y genera una función `trace()` en caso afirmativo o genera una función `trace()` diferente en caso negativo:

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

Si no desea ejecutar un bloque de código alternativo, se puede utilizar la sentencia `if` sin la sentencia `else`.

if..else if

Puede comprobar varias condiciones utilizando la sentencia condicional `if..else if`. Por ejemplo, el siguiente fragmento de código no sólo comprueba si el valor de `x` es superior a 20, sino que también comprueba si el valor de `x` es negativo:

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

Si una sentencia `if` o `else` va seguida de una sola sentencia, no es necesario escribir dicha sentencia entre llaves. Por ejemplo, en el código siguiente no se usan llaves:

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

No obstante, Adobe recomienda utilizar siempre llaves, ya que podría producirse un comportamiento inesperado si más adelante se añadieran sentencias a una sentencia condicional que no esté escrita entre llaves. Por ejemplo, en el código siguiente el valor de `positiveNums` aumenta en 1 independientemente de si la evaluación de la condición devuelve `true`:

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

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

trace(positiveNums); // 1
```

switch

La sentencia `switch` resulta útil si hay varios hilos de ejecución que dependen de la misma expresión de condición. La funcionalidad que proporciona es similar a una serie larga de sentencias `if..else if`, pero su lectura resulta un tanto más sencilla. En lugar de probar una condición para un valor booleano, la sentencia `switch` evalúa una expresión y utiliza el resultado para determinar el bloque de código que debe ejecutarse. Los bloques de código empiezan por una sentencia `case` y terminan con una sentencia `break`. Por ejemplo, la siguiente sentencia `switch` imprime el día de la semana en función del número de día devuelto por el método `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;
}
```

Reproducir indefinidamente

Las sentencias de bucle permiten ejecutar un bloque específico de código repetidamente utilizando una serie de valores o variables. Adobe recomienda escribir siempre el bloque de código entre llaves (`{}`). Aunque puede omitir las llaves si el bloque de código sólo contiene una sentencia, no es recomendable que lo haga por la misma razón expuesta para las condicionales: aumenta la posibilidad de que las sentencias añadidas posteriormente se excluyan inadvertidamente del bloque de código. Si posteriormente se añade una sentencia que se desea incluir en el bloque de código, pero no se añaden las llaves necesarias, la sentencia no se ejecutará como parte del bucle.

for

El bucle `for` permite repetir una variable para un rango de valores específico. Debe proporcionar tres expresiones en una sentencia `for`: una variable que se establece con un valor inicial, una sentencia condicional que determina cuándo termina la reproducción en bucle y una expresión que cambia el valor de la variable con cada bucle. Por ejemplo, el siguiente código realiza cinco bucles. El valor de la variable `i` comienza en 0 y termina en 4, mientras que la salida son los números 0 a 4, cada uno de ellos en su propia línea.

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

for..in

El bucle `for..in` recorre las propiedades de un objeto o los elementos de un conjunto. Por ejemplo, se puede utilizar un bucle `for..in` para recorrer las propiedades de un objeto genérico (las propiedades de un objeto no se guardan en ningún orden concreto, por lo que pueden aparecer en un orden aparentemente impredecible):

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

También se pueden recorrer los elementos de un conjunto:

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

Lo que no se puede hacer es repetir las propiedades de un objeto si se trata de una instancia de una clase cerrada (incluyendo las clases incorporadas y las definidas por el usuario). Sólo se pueden repetir las propiedades de una clase dinámica. Incluso con instancias de clases dinámicas, sólo se pueden repetir las propiedades que se añadan dinámicamente.

for each..in

El bucle `for each..in` recorre los elementos de una colección, que puede estar formada por las etiquetas de un objeto XML o XMLList, los valores de las propiedades de un objeto o los elementos de un conjunto. Por ejemplo, como muestra el fragmento de código siguiente, el bucle `for each..in` se puede utilizar para recorrer las propiedades de un objeto genérico, pero al contrario de lo que ocurre con el bucle `for..in`, la variable de iteración de los bucles `for each..in` contiene el valor contenido por la propiedad en lugar del nombre de la misma:

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

Se puede recorrer un objeto XML o XMLList, como se indica en el siguiente ejemplo:

```
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
*/
```

También se pueden recorrer los elementos de un conjunto, como se indica en este ejemplo:

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

No se pueden recorrer las propiedades de un objeto si el objeto es una instancia de una clase cerrada. Tampoco se pueden recorrer las propiedades fijas (propiedades definidas como parte de una definición de clase), ni siquiera para las instancias de clases dinámicas.

while

El bucle `while` es como una sentencia `if` que se repite con tal de que la condición sea `true`. Por ejemplo, el código siguiente produce el mismo resultado que el ejemplo del bucle `for`:

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

Una desventaja que presenta el uso de los bucles `while` frente a los bucles `for` es que es más probable escribir un bucle infinito con bucles `while`. El código de ejemplo de bucle `for` no se compila si se omite la expresión que aumenta la variable de contador, mientras que el ejemplo de bucle `while` sí se compila si se omite dicho paso. Sin la expresión que incrementa `i`, el bucle se convierte en un bucle infinito.

do..while

El bucle `do..while` es un bucle `while` que garantiza que el bloque de código se ejecuta al menos una vez, ya que la condición se comprueba después de que se ejecute el bloque de código. El código siguiente muestra un ejemplo simple de un bucle `do..while` que genera una salida aunque no se cumple la condición:

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

Funciones

Las *funciones* son bloques de código que realizan tareas específicas y pueden reutilizarse en el programa. Hay dos tipos de funciones en ActionScript 3.0: *métodos* y *cierres de función*. Llamar a una función método o cierre de función depende del contexto en el que se define la función. Una función se denomina método si se define como parte de una definición de clase o se asocia a una instancia de un objeto. Y se denomina cierre de función si se define de cualquier otra manera.

Las funciones siempre han sido muy importantes en ActionScript. Por ejemplo, en ActionScript 1.0 la palabra clave `class` no existía, por lo que las "clases" se definían mediante funciones constructoras. Aunque la palabra clave `class` se añadió posteriormente al lenguaje, sigue siendo importante comprender a fondo las funciones para aprovechar al máximo las capacidades del lenguaje. Esto puede ser difícil de entender para los programadores que esperan que las funciones de ActionScript se comporten de forma similar a las funciones de lenguajes como C++ o Java. Aunque una definición básica de función e invocación no debe resultar difícil para los programadores con experiencia, algunas de las características más avanzadas de las funciones de ActionScript requieren una explicación.

Fundamentos de la utilización de funciones

Invocación de funciones

Para llamar a una función se utiliza su identificador seguido del operador paréntesis (`()`). Se puede utilizar el operador paréntesis para escribir los parámetros de función que se desea enviar a la función. Por ejemplo, `trace()` es una función de nivel superior en ActionScript 3.0:

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

Si se llama a una función sin parámetros, hay que utilizar un par de paréntesis vacíos. Por ejemplo, se puede utilizar el método `Math.random()`, que no admite parámetros, para generar un número aleatorio:

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

Funciones definidas por el usuario

Hay dos formas de definir una función en ActionScript 3.0: se puede utilizar una sentencia de función o una expresión de función. La técnica que se elija dependerá de si se prefiere un estilo de programación más estático o más dinámico. Si se prefiere la programación estática, o en modo estricto, se deben definir las funciones con sentencias de función. Las funciones deben definirse con expresiones de función si existe la necesidad específica de hacerlo. Las expresiones de función se suelen usar en programación dinámica (en modo estándar).

Sentencias de función

Las sentencias de función son la técnica preferida para definir funciones en modo estricto. Una sentencia de función empieza con la palabra clave `function`, seguida de:

- El nombre de la función

- Los parámetros, en una lista delimitada por comas y escrita entre paréntesis
- El cuerpo de la función (es decir, el código ActionScript que debe ejecutarse cuando se invoca la función), escrito entre llaves.

Por ejemplo, el código siguiente crea una función que define un parámetro y después llama a la función con la cadena `"hello"` como valor del parámetro:

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

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

Expresiones de función

La segunda manera de declarar una función es utilizar una sentencia de asignación con una expresión de función (también se suele llamar literal de función o función anónima). Éste es un método que requiere escribir más y que se usaba mucho en versiones anteriores de ActionScript.

Una sentencia de asignación con una expresión de función empieza por la palabra clave `var`, seguida de:

- El nombre de la función
- El operador dos puntos (`:`)
- La clase `Function` para indicar el tipo de datos
- El operador de asignación (=)
- La palabra clave `function`
- Los parámetros, en una lista delimitada por comas y escrita entre paréntesis
- El cuerpo de la función (es decir, el código ActionScript que debe ejecutarse cuando se invoca la función), escrito entre llaves.

  Por ejemplo, el código siguiente declara la función `traceParameter` mediante una expresión de función:

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

  Tenga en cuenta que, a diferencia de lo que ocurre en una sentencia de función, no se especifica un nombre de función. Otra diferencia importante entre las expresiones de función y las sentencias de función es que una expresión de función es una expresión, no una sentencia. Esto significa que una expresión de función no es independiente, como una sentencia de función. Una expresión de función sólo se puede utilizar como una parte de una sentencia (normalmente una sentencia de asignación). En el siguiente ejemplo se muestra la asignación de una expresión de función a un elemento de conjunto:

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

Criterios para elegir entre sentencias y expresiones

Como regla general, se debe utilizar una sentencia de función a menos que circunstancias específicas requieran una expresión. Las sentencias de función son menos detalladas y proporcionan una experiencia más uniforme entre el modo estricto y el modo estándar que las expresiones de función.

También son más fáciles de leer que las sentencias de asignación que contienen expresiones de función. Por otra parte, las sentencias de función hacen que el código sea más conciso; son menos confusas que las expresiones de función, que requieren utilizar las palabras clave `var` y `function`.

Además, proporcionan una experiencia más uniforme entre los dos modos de compilador, ya que permiten utilizar la sintaxis con punto en modo estándar y en modo estricto para llamar a un método declarado con una sentencia de función. Esto no es así necesariamente para los métodos declarados con una expresión de función. Por ejemplo, el código siguiente define una clase denominada Example con dos métodos: `methodExpression()`, que se declara con una expresión de función, y `methodStatement()`, que se declara con una sentencia de función. En modo estricto no se puede utilizar la sintaxis con punto para llamar al método `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
```

Las expresiones de función se consideran más apropiadas para la programación centrada en el comportamiento dinámico (en tiempo de ejecución). Si se prefiere utilizar el modo estricto, pero también hay que llamar a un método declarado con una expresión de función, se puede utilizar cualquiera de las dos técnicas. En primer lugar, se puede llamar al método utilizando corchetes (`[]`) el lugar del operador punto (`.`). La siguiente llamada a método funciona correctamente tanto en modo estricto como en modo estándar:

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

En segundo lugar, se puede declarar toda la clase como una clase dinámica. Aunque esto permite llamar al método con el operador punto, la desventaja es que se sacrifica parte de la funcionalidad en modo estricto para todas las instancias de la clase. Por ejemplo, el compilador no genera un error si se intenta acceder a una propiedad no definida en una instancia de una clase dinámica.

Hay algunas circunstancias en las que las expresiones de función son útiles. Las expresiones de función se suelen utilizar para crear funciones que se utilizan una sola vez y después se descartan. Otro uso menos común es asociar una función a una propiedad de prototipo. Para obtener más información, consulte El objeto prototype.

Hay dos diferencias sutiles entre las sentencias de función y las expresiones de función que se deben tener en cuenta al elegir la técnica que se va a utilizar. La primera diferencia es que las expresiones de función no existen de forma independiente como objetos con respecto a la administración de la memoria y la recolección de elementos no utilizados. Es decir, cuando se asigna una expresión de función a otro objeto, como un elemento de conjunto o una propiedad de objeto, se crea la única referencia a esa expresión de función en el código. Si el conjunto o el objeto al que la expresión de función está asociada se salen del ámbito o deja de estar disponible, se dejará de tener acceso a la expresión de función. Si se elimina el conjunto o el objeto, la memoria utilizada por la expresión de función quedará disponible para la recolección de elementos no utilizados, lo que significa que se podrá recuperar esa memoria y reutilizarla para otros propósitos.

En el siguiente ejemplo se muestra que, para una expresión de función, cuando se elimina la propiedad a la que está asignada la expresión, la función deja de estar disponible. La clase Test es dinámica, lo que significa que se puede añadir una propiedad denominada `functionExp` que contendrá una expresión de función. Se puede llamar a la función `functionExp()` con el operador punto, pero cuando se elimina la propiedad `functionExp`, la función deja de ser accesible.

```
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
```

Si, por otra parte, la función se define primero con una sentencia de función, existe como su propio objeto y seguirá existiendo incluso después de que se elimine la propiedad a la que está asociada. El operador `delete` sólo funciona en propiedades de objetos, por lo que incluso una llamada para eliminar la función `stateFunc()` no funciona.

```
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 segunda diferencia entre las sentencias de función y las expresiones de función es que las sentencias de función existen en todo el ámbito en que están definidas, incluso en sentencias que aparecen antes que la sentencia de función. En cambio, las expresiones de función sólo están definidas para las sentencias posteriores. Por ejemplo, el código siguiente llama correctamente a la función `scopeTest()` antes de que se defina:

```
statementTest(); // statementTest

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

Las expresiones de función no están disponibles antes de ser definidas, por lo que el código siguiente produce un error en tiempo de ejecución:

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

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

Devolución de valores de funciones

Para devolver un valor de la función se debe utilizar la sentencia `return` seguida de la expresión o el valor literal que se desea devolver. Por ejemplo, el código siguiente devuelve una expresión que representa al parámetro:

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

Tenga en cuenta que la sentencia `return` finaliza la función, por lo que las sentencias que estén por debajo de una sentencia `return` no se ejecutarán, como se indica a continuación:

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

En modo estricto se debe devolver un valor del tipo apropiado si se elige especificar un tipo devuelto. Por ejemplo, el código siguiente genera un error en modo estricto porque no devuelve un valor válido:

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

Funciones anidadas

Es posible anidar funciones, lo que significa que pueden declararse funciones dentro de otras funciones. Una función anidada sólo está disponible dentro de su función principal, a menos que se pase una referencia a la función a código externo. Por ejemplo, el código siguiente declara dos funciones anidadas dentro de la función `getNameAndVersion()`:

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

Cuando se pasan funciones anidadas a código externo, se pasan como cierres de función, lo que significa que la función retiene todas las definiciones que hubiera en el ámbito cuando se definió la función. Para obtener más información, consulte Ámbito de una función.

Parámetros de función

ActionScript 3.0 proporciona funcionalidad para los parámetros de función que puede resultar novedosa para los programadores que empiecen a estudiar el lenguaje. Aunque la mayoría de los programadores deberían estar familiarizados con la idea de pasar parámetros por valor o referencia, es posible que el objeto `arguments` y el parámetro ... (rest) sean desconocidos para muchos.

Pasar argumentos por valor o por referencia

En muchos lenguajes de programación, es importante comprender la diferencia entre pasar argumentos por valor o por referencia; esta diferencia puede afectar a la manera de diseñar el código.

Al pasar por valor, el valor del argumento se copia en una variable local para usarlo en la función. Al pasar por referencia, sólo se pasa una referencia al argumento, en lugar del valor real. No se realiza ninguna copia del argumento real. En su lugar, se crea una referencia a la variable pasada como argumento y se asigna dicha referencia a una variable local para usarla en la función. Como una referencia a una variable externa a la función, la variable local proporciona la capacidad de cambiar el valor de la variable original.

En ActionScript 3.0, todos los argumentos se pasan por referencia, ya que todos los valores se almacenan como objetos. No obstante, los objetos que pertenecen a los tipos de datos simples, como Boolean, Number, int, uint y String, tienen operadores especiales que hacen que se comporten como si se pasaran por valor. Por ejemplo, el código siguiente crea una función denominada `passPrimitives()` que define dos parámetros denominados `xParam` y `yParam`, ambos de tipo int. Estos parámetros son similares a variables locales declaradas en el cuerpo de la función `passPrimitives()`. Cuando se llama a la función con los argumentos `xValue` e `yValue`, los parámetros `xParam` e `yParam` se inicializan con referencias a los objetos int representados por `xValue` e `yValue`. Como los argumentos son valores simples, se comportan como si se pasaran por valor. Aunque `xParam` e `yParam` sólo contienen inicialmente referencias a los objetos `xValue` e `yValue`, los cambios realizados a las variables en el cuerpo de la función generan nuevas copias de los valores en la 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
```

En la función `passPrimitives()`, los valores de `xParam` e `yParam` se incrementan, pero esto no afecta a los valores de `xValue` e `yValue`, como se indica en la última sentencia `trace`. Esto es así aunque se asigne a los parámetros los mismos nombres que a las variables, `xValue` e `yValue`, ya que dentro de la función `xValue` e `yValue` señalarían nuevas ubicaciones de la memoria que existen por separado de las variables externas a la función que tienen el mismo nombre.

Todos los demás objetos (es decir, los objetos que no pertenecen a los tipos de datos simples) se pasan siempre por referencia, ya que esto ofrece la capacidad de cambiar el valor de la variable original. Por ejemplo, el código siguiente crea un objeto denominado `objVar` con dos propiedades, `x` e `y`. El objeto se pasa como un argumento a la función `passByRef()`. Como el objeto no es un tipo simple, no sólo se pasa por referencia, sino que también se mantiene como una referencia. Esto significa que los cambios realizados en los parámetros dentro de la función afectarán a las propiedades del objeto fuera de la función.

```
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
```

El parámetro `objParam` hace referencia al mismo objeto que la variable `objVar` global. Como se puede ver en las sentencias `trace` del ejemplo, los cambios realizados en las propiedades `x` e `y` del objeto `objParam` se reflejan en el objeto `objVar`.

Valores predeterminados de los parámetros

En ActionScript 3.0, se puede declarar *valores predeterminados de parámetros* para una función. Si una llamada a una función con valores predeterminados de parámetros omite un parámetro con valores predeterminados, se utiliza el valor especificado en la definición de la función para ese parámetro. Todos los parámetros con valores predeterminados deben colocarse al final de la lista de parámetros. Los valores asignados como valores predeterminados deben ser constantes de tiempo de compilación. La existencia de un valor predeterminado para un parámetro convierte de forma efectiva a ese parámetro en un *parámetro opcional*. Un parámetro sin un valor predeterminado se considera un *parámetro requerido*.

Por ejemplo, el código siguiente crea una función con tres parámetros, dos de los cuales tienen valores predeterminados. Cuando se llama a la función con un solo parámetro, se utilizan los valores predeterminados de los parámetros.

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

El objeto arguments

Cuando se pasan parámetros a una función, se puede utilizar el objeto `arguments` para acceder a información sobre los parámetros pasados a la función. Algunos aspectos importantes del objeto `arguments` son:

- El objeto `arguments` es un conjunto que incluye todos los parámetros pasados a la función.
- La propiedad `arguments.length` notifica el número de parámetros pasados a la función.
- La propiedad `arguments.callee` proporciona una referencia a la misma función, que resulta útil para llamadas recursivas a expresiones de función.

  ***Nota:** el objeto `arguments` no estará disponible si algún parámetro tiene el nombre `arguments` o si se utiliza el parámetro ... (rest).*

  Si se hace referencia al objeto `arguments` en el cuerpo de una función, ActionScript 3.0 permite que las llamadas a funciones incluyan más parámetros que los definidos en la definición de la función, pero generará un error del compilador en modo estricto si el número de parámetros no coincide con el número de parámetros requeridos (y de forma opcional, los parámetros opcionales). Se puede utilizar el conjunto aspect del objeto `arguments` para acceder a cualquier parámetro pasado a la función, independientemente de si ese parámetro está definido en la definición de la función. En el ejemplo siguiente, que sólo compila en modo estándar, se utiliza el conjunto `arguments` junto con la propiedad `arguments.length` para hacer un seguimiento de todos los parámetros pasados a la función `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 propiedad `arguments.callee` se suele utilizar en funciones anónimas para crear recursión. Se puede utilizar para añadir flexibilidad al código. Si el nombre de una función recursiva cambia a lo largo del ciclo de desarrollo, no es necesario preocuparse de cambiar la llamada recursiva en el cuerpo de la función si se utiliza `arguments.callee` en lugar del nombre de la función. La propiedad `arguments.callee` se utiliza en la siguiente expresión de función para habilitar la recursión:

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

Si se utiliza el parámetro ... (rest) en la declaración de la función, el objeto `arguments` no estará disponible. Hay que acceder a los parámetros a través de los nombres de parámetro declarados.

También hay que procurar no utilizar la cadena `"arguments"` como nombre de parámetro, ya que ocultará el objeto `arguments`. Por ejemplo, si se vuelve a escribir la función `traceArgArray()` de forma que se añade un parámetro `arguments`, las referencias a `arguments` en el cuerpo de la función hacen referencia al parámetro, en lugar de al objeto `arguments.` El siguiente código no produce ningún resultado:

```
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
```

El objeto `arguments` de versiones anteriores de ActionScript también contenía una propiedad denominada `caller`, que es una referencia a la función que llamó a la función actual. La propiedad `caller` no existe en ActionScript 3.0, pero si se necesita una referencia a la función que llama, se puede modificar dicha función de forma que pase un parámetro adicional que sea una referencia sí mismo.

El parámetro ...(rest)

ActionScript 3.0 introduce una declaración de un parámetro nuevo que se llama ... (rest). Este parámetro permite especificar un parámetro de tipo conjunto que acepta un número arbitrario de argumentos delimitados por comas. El parámetro puede tener cualquier nombre que no sea una palabra reservada. Este parámetro debe especificarse el último. El uso de este parámetro hace que el objeto `arguments` no esté disponible. Aunque el parámetro ... (rest) ofrece la misma funcionalidad que el conjunto `arguments` y la propiedad `arguments.length`, no proporciona funcionalidad similar a la que ofrece `arguments.callee`. Hay que asegurarse de que no es necesario utilizar `arguments.callee` antes de utilizar el parámetro ... (rest).

En el ejemplo siguiente se reescribe la función `traceArgArray()` con el parámetro ... (rest) en lugar del objeto `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
```

El parámetro ... (rest) también puede utilizarse con otros parámetros, con tal de que sea el último parámetro de la lista. En el ejemplo siguiente se modifica la función `traceArgArray()` de forma que su primer parámetro, `x`, sea de tipo int y el segundo parámetro utilice el parámetro ... (rest). La salida omite el primer valor porque el primer parámetro ya no forma parte del conjunto creada por el parámetro ... (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
```

Funciones como objetos

En ActionScript 3.0 las funciones son objetos. Al crear una función, se crea un objeto que no sólo se puede pasar como un parámetro a otra función, sino que además tiene propiedades y métodos asociados.

Las funciones pasadas como argumentos a otra función se pasan por referencia, no por valor. Al pasar una función como un argumento sólo se utiliza el identificador y no el operador paréntesis que se utiliza para llamar al método. Por ejemplo, el código siguiente pasa una función denominada `clickListener()` como un argumento al método `addEventListener()`:

```
addEventListener(MouseEvent.CLICK, clickListener);
```

Aunque pueda parecer extraño a los programadores sin experiencia en ActionScript, las funciones pueden tener propiedades y métodos, igual que cualquier otro objeto. De hecho, cada función tiene una propiedad de sólo lectura denominada `length` que almacena el número de parámetros definidos para la función. Es distinta de la propiedad `arguments.length`, que notifica el número de argumentos enviados a la función. Debe recordarse que en ActionScript el número de argumentos enviados a una función pueden superar el número de parámetros definidos para dicha función. En el ejemplo siguiente, que sólo se compila en modo estándar porque el modo estricto requiere una coincidencia exacta entre el número de argumentos pasados y el número de parámetros definidos, se muestra la diferencia entre las dos propiedades:

```
// 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 */
```

En modo estándar se pueden definir propiedades de función propias fuera del cuerpo de la función. Las propiedades de función pueden servir como propiedades casi estáticas que permiten guardar el estado de una variable relacionada con la función. Por ejemplo, si se desea hacer un seguimiento del número de veces que se llama a una función determinada. Esta funcionalidad puede ser útil cuando se programa un juego y se desea hacer un seguimiento del número de veces que un usuario utiliza un comando específico, aunque también se podría utilizar una propiedad de clase estática para esto. El ejemplo siguiente, que sólo compila en modo estándar porque el modo estricto no permite añadir propiedades dinámicas a funciones, crea una propiedad de función fuera de la declaración de función e incrementa la propiedad cada vez que se llama a la función:

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

Ámbito de una función

El ámbito de una función determina no sólo en qué partes de un programa se puede llamar a esa función, sino también a qué definiciones tiene acceso la función. Las mismas reglas de ámbito que se aplican a los identificadores de variable se aplican a los identificadores de función. Una función declarada en el ámbito global estará disponible en todo el código. Por ejemplo, ActionScript 3.0 contiene funciones globales, como `isNaN()` y `parseInt()`, que están disponibles desde cualquier punto del código. Una función anidada (una función declarada dentro de otra función) puede utilizarse en cualquier punto de la función en que se declaró.

La cadena de ámbitos

Cuando se inicia la ejecución de una función, se crean diversos objetos y propiedades. En primer lugar, se crea un objeto especial denominado *objeto de activación* que almacena los parámetros y las variables o funciones locales declaradas en el cuerpo de la función. No se puede acceder al objeto de activación directamente, ya que es un mecanismo interno. En segundo lugar, se crea una *cadena de ámbitos* que contiene una lista ordenada de objetos en las que el motor de ejecución comprueba las declaraciones de identificadores. Cada función que ejecuta tiene una cadena de ámbitos que se almacena en una propiedad interna. Para una función anidada, la cadena de ámbitos empieza en su propio objeto de activación, seguido del objeto de activación de la función principal. La cadena continúa de esta manera hasta que llega al objeto global. El objeto global se crea cuando se inicia un programa de ActionScript y contiene todas las variables y funciones globales.

Cierres de función

Un *cierre de función* es un objeto que contiene una instantánea de una función y su *entorno léxico*. El entorno léxico de una función incluye todas las variables, propiedades, métodos y objetos de la cadena de ámbitos de la función, junto con sus valores. Los cierres de función se crean cada vez que una función se ejecuta aparte de un objeto o una clase. El hecho de que los cierres de función conserven el ámbito en que se definieron crea resultados interesantes cuando se pasa una función como un argumento o un valor devuelto en un ámbito diferente.

Por ejemplo, el código siguiente crea dos funciones: `foo()`, que devuelve una función anidada denominada `rectArea()` que calcula el área de un rectángulo y `bar()`, que llama a `foo()` y almacena el cierre de función devuelto en una variable denominada `myProduct`. Aunque la función `bar()` define su propia variable local `x` (con el valor 2), cuando se llama al cierre de función `myProduct()`, conserva la variable `x` (con el valor 40) definida en la función `foo()`. Por tanto, la función `bar()` devuelve el valor `160` en lugar de`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
```

Los métodos se comportan de manera similar, ya que también conservan información sobre el entorno léxico en que se crearon. Esta característica se aprecia especialmente cuando se extrae un método de su instancia, lo que crea un método vinculado. La diferencia principal entre un cierre de función y un método vinculado es que el valor de la palabra clave `this` en un método vinculado siempre hace referencia a la instancia a la que estaba asociado originalmente, mientras que en un cierre de función el valor de la palabra clave `this` puede cambiar.

Capítulo 4: Programación orientada a objetos con ActionScript

Introducción a la programación orientada a objetos

La programación orientada a objetos es una forma de organizar el código de un programa agrupándolo en objetos. En este sentido, el término *objeto* indica un elemento individual que contienen información (valores de datos) y funcionalidad. La utilización de un enfoque orientado a objetos para organizar un programa permite agrupar partes de información con una funcionalidad común o acciones asociadas con dicha información. Por ejemplo, se puede agrupar información musical, como el título de álbum, el título de la pista o el nombre del artista, junto con determinada funcionalidad como "añadir pista a la lista de reproducción" o "reproducir todas las canciones de este artista".Estos elementos se combinan en un solo elemento, denominado objeto (por ejemplo, un objeto "Album" o "MusicTrack"). La agrupación de valores y funciones ofrece varias ventajas. Una ventaja fundamental es que sólo es necesario utilizar una sola variable en lugar de varias. Asimismo, la funcionalidad relacionada se mantiene asociada. Finalmente, la combinación de información y funcionalidad permite estructurar los programas que modo que reflejen mejor el mundo real.

Clases

Una clase es una representación abstracta de un objeto. Una clase almacena información sobre los tipos de datos que un objeto puede contener y los comportamientos que un objeto puede exhibir. La utilidad de esta abstracción puede no ser apreciable al escribir scripts sencillos que sólo contienen unos pocos objetos que interactúan entre sí. Sin embargo, conforme crece el ámbito de un programa aumenta el número de objetos que deben administrarse. En este caso, las clases permiten controlar mejor el modo en que se crean los objetos y la forma en que interaccionan unos con otros.

Con ActionScript 1.0 los programadores podían utilizar objetos Function para crear construcciones similares a las clases. ActionScript 2.0 añadió formalmente la compatibilidad con las clases con palabras clave como `class` y `extends`. En ActionScript 3.0 no sólo se mantienen las palabras clave introducidas en ActionScript 2.0, sino que también se han añadido algunas capacidades nuevas. Por ejemplo, ActionScript 3.0 incluye un control de acceso mejorado con los atributos `protected` e `internal`. También proporciona un mejor control de la herencia con las palabras clave `final` y `override`.

A los desarrolladores con experiencia en la creación de clases con lenguajes de programación como Java, C++ o C#, ActionScript les resultará familiar. ActionScript comparte muchas de las mismas palabras clave y nombres de atributo como, por ejemplo, `class`, `extends` y `public`.

***Nota:** en la documentación de Adobe ActionScript, el término propiedad designa cualquier miembro de un objeto o una clase, incluidas variables, constantes y métodos. Por otra parte, aunque los términos class y static se suelen utilizar como sinónimos, aquí tienen un significado diferente. Por ejemplo, la frase "propiedades de clase" hace referencia a todos los miembros de una clase, no únicamente a los miembros estáticos.*

Definiciones de clase

En las definiciones de clase de ActionScript 3.0 se utiliza una sintaxis similar a la utilizada en las definiciones de clase de ActionScript 2.0. La sintaxis correcta de una definición de clase requiere la palabra clave `class` seguida del nombre de la clase. El cuerpo de la clase, que se escribe entre llaves (`{}`), sigue al nombre de la clase. Por ejemplo, el código siguiente crea una clase denominada Shape que contiene una variable denominada `visible`:

```
public class Shape
{
    var visible:Boolean = true;
}
```

Un cambio de sintaxis importante afecta a las definiciones de clase que están dentro de un paquete. En ActionScript 2.0, si una clase estaba dentro de un paquete, había que incluir el nombre de paquete en la declaración de la clase. En ActionScript 3.0 se incluye la sentencia `package` y hay que incluir el nombre del paquete en la declaración del paquete, no en la declaración de clase. Por ejemplo, las siguientes declaraciones de clase muestran la manera de definir la clase BitmapData, que forma parte del paquete flash.display, en ActionScript 2.0 y en ActionScript 3.0:

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

Atributos de clase

ActionScript 3.0 permite modificar las definiciones de clase mediante uno de los cuatro atributos siguientes:

Atributo	Definición
`dynamic`	Permite añadir propiedades a instancias en tiempo de ejecución.
`final`	No debe ser ampliada por otra clase.
`internal` (valor predeterminado)	Visible para referencias dentro del paquete actual.
`public`	Visible para referencias en todas partes.

Todos estos atributos, salvo `internal`, deben ser incluidos explícitamente para obtener el comportamiento asociado. Por ejemplo, si no se incluye el atributo `dynamic` al definir una clase, no se podrá añadir propiedades a una instancia de clase en tiempo de ejecución. Un atributo se asigna explícitamente colocándolo al principio de la definición de clase, como se muestra en el código siguiente:

```
dynamic class Shape {}
```

Hay que tener en cuenta que la lista no incluye un atributo denominado `abstract`. Las clases abstractas no se admiten en ActionScript 3.0. También se debe tener en cuenta que la lista no incluye atributos denominados `private` y `protected`. Estos atributos sólo tienen significado dentro de una definición de clase y no se pueden aplicar a las mismas clases. Si no se desea que una clase sea visible públicamente fuera de un paquete, debe colocarse la clase dentro de un paquete y marcarse con el atributo `internal`. Como alternativa, se pueden omitir los atributos `internal` y `public` y el compilador añadirá automáticamente el atributo `internal`. También se puede definir una clase para que sólo se pueda ver en el archivo de origen en el que se define. Sitúe la clase en la parte inferior del archivo de origen, bajo la llave de cierre de la definición del paquete.

Cuerpo de la clase

El cuerpo de la clase se escribe entre llaves. Define las variables, constantes y métodos de la clase. En el siguiente ejemplo se muestra la declaración para la clase Accessibility en ActionScript 3.0:

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

También se puede definir un espacio de nombres dentro de un cuerpo de clase. En el siguiente ejemplo se muestra cómo se puede definir un espacio de nombres en el cuerpo de una clase y utilizarse como atributo de un método en dicha clase:

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 permite incluir en el cuerpo de una clase no sólo definiciones, sino también sentencias. Las sentencias que están dentro de un cuerpo de clase pero fuera de una definición de método se ejecutan una sola vez.Esta ejecución se produce cuando se encuentra por primera vez la definición de la clase y se crea el objeto de clase asociado. En el ejemplo siguiente se incluye una llamada a una función externa, `hello()`, y una sentencia `trace` que emite un mensaje de confirmación cuando se define la clase:

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

En ActionScript 3.0 se permite definir en un mismo cuerpo de clase una propiedad estática y una propiedad de instancia con el mismo nombre. Por ejemplo, el código siguiente declara una variable estática denominada `message` y una variable de instancia con el mismo nombre:

```
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
```

Atributos de propiedad de clase

En las descripciones del modelo de objetos de ActionScript, el término *propiedad* significa cualquier cosa que pueda ser un miembro de una clase, incluidas variables, constantes y métodos. Sin embargo, en Adobe ActionScript 3.0 Reference for the Adobe Flash Platform, el término se aplica a un concepto menos amplio. En ese contexto la propiedad del término sólo incluye miembros de clase que son variables o se definen mediante un método captador o definidor. En ActionScript 3.0 hay un conjunto de atributos que se pueden utilizar con cualquier propiedad de una clase. En la tabla siguiente se muestra este conjunto de atributos.

Atributo	Definición
`internal` (valor predeterminado)	Visible para referencias dentro del mismo paquete.
`private`	Visible para referencias dentro de la misma clase.
`protected`	Visible para referencias en la misma clase y en clases derivadas.
`public`	Visible para referencias en todas partes.
`static`	Especifica que una propiedad pertenece a la clase en lugar de a las instancias de la clase.
`UserDefinedNamespace`	Nombre de espacio de nombres personalizado definido por el usuario.

Atributos del espacio de nombres de control de acceso

ActionScript 3.0 proporciona cuatro atributos especiales que controlan el acceso a las propiedades definidas dentro de una clase: `public`, `private`, `protected` e `internal`.

El atributo `public` hace que una propiedad esté visible en cualquier parte del script. Por ejemplo, para hacer que un método esté disponible para el código fuera de su paquete, hay que declarar el método con el atributo `public.` Esto se cumple para cualquier propiedad, independientemente de que se declare con la palabra clave `var`, `const` o `function`.

El atributo `private` hace que una propiedad sólo esté visible para los orígenes de llamada de la clase en la que se define la propiedad. Este comportamiento difiere del comportamiento del atributo `private` en ActionScript 2.0, que permitía a una subclase tener acceso a una propiedad privada de una superclase. Otro cambio importante de comportamiento está relacionado con el acceso en tiempo de ejecución. En ActionScript 2.0, la palabra clave `private` sólo prohibía el acceso en tiempo de compilación y se podía evitar fácilmente en tiempo de ejecución. Esto ya no se cumple en ActionScript 3.0. Las propiedades marcadas como `private` no están disponibles en tiempo de compilación ni en tiempo de ejecución.

Por ejemplo, el código siguiente crea una clase simple denominada PrivateExample con una variable privada y después intenta acceder a la variable privada desde fuera de la clase.

```
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.
```

En ActionScript 3.0, un intento de acceder a una propiedad privada mediante el operador punto (`myExample.privVar`) provoca un error de tiempo de compilación si se utiliza el modo estricto. De lo contrario, el error se notifica en tiempo de ejecución, de la misma que manera que al usar el operador de acceso a una propiedad (`myExample["privVar"]`).

En la tabla siguiente se resumen los resultados de intentar acceder a una propiedad privada que pertenece a una clase cerrada (no dinámica):

	Modo estricto	**Modo estándar**
operador punto (`.`)	error en tiempo de compilación	error en tiempo de ejecución
operador corchete (`[]`)	error en tiempo de ejecución	error en tiempo de ejecución

En clases declaradas con el atributo `dynamic`, los intentos de acceder a una variable privada no provocarán un error en tiempo de ejecución. En su lugar, la variable no está visible, por lo que se devuelve el valor `undefined`. No obstante, se producirá un error en tiempo de compilación si se utiliza el operador punto en modo estricto. El ejemplo siguiente es igual que el anterior, con la diferencia de que la clase PrivateExample se declara como una clase dinámica:

```
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
```

Las clases dinámicas generalmente devuelven el valor `undefined` en lugar de generar un error cuando código externo a una clase intenta acceder a una propiedad privada. En la tabla siguiente se muestra que sólo se genera un error cuando se utiliza el operador punto para acceder a una propiedad privada en modo estricto:

	Modo estricto	**Modo estándar**
operador punto (`.`)	error en tiempo de compilación	`undefined`
operador corchete (`[]`)	`undefined`	`undefined`

El atributo `protected`, que es una de las novedades de ActionScript 3.0, hace que una propiedad esté visible para los orígenes de llamada en su propia clase o en una subclase. Es decir, una propiedad protegida está disponible en su propia clase o para clases de nivel inferior en la jerarquía de herencia. Esto se cumple tanto si la subclase está en el mismo paquete como si está en un paquete diferente.

Para los usuarios familiarizados con ActionScript 2.0, esta funcionalidad es similar al atributo `private` en ActionScript 2.0. El atributo `protected` de ActionScript 3.0 también es similar al atributo `protected` en Java. Difiere en que la versión de Java también permite acceder a quien realiza la llamada en el mismo paquete. El atributo `protected` resulta útil cuando se tiene una variable o método requerido por las subclases que se desea ocultar del código que esté fuera de la cadena de herencia.

El atributo `internal`, que es una de las novedades de ActionScript 3.0, hace que una propiedad esté visible para los orígenes de llamada en su propio paquete. Es el atributo predeterminado para el código de un paquete y se aplica a cualquier propiedad que no tenga ninguno de los siguientes atributos:

- `public`
- `private`
- `protected`
- un espacio de nombres definido por el usuario

El atributo `internal` es similar al control de acceso predeterminado en Java, aunque en Java no hay ningún nombre explícito para este nivel de acceso y sólo se puede alcanzar mediante la omisión de cualquier otro modificador de acceso. El atributo `internal` está disponible en ActionScript 3.0 para ofrecer la opción de indicar explícitamente la intención de hacer que una propiedad sólo sea visible para orígenes de llamada de su propio paquete.

Atributo static

El atributo `static`, que se puede utilizar con propiedades declaradas con las palabras clave `var`, `const` o `function`, permite asociar una propiedad a la clase en lugar de asociarla a instancias de la clase. El código externo a la clase debe llamar a propiedades estáticas utilizando el nombre de la clase en lugar de un nombre de instancia.

Las subclases no heredan las propiedades estáticas, pero las propiedades forman parte de una cadena de ámbitos de subclase. Esto significa que en el cuerpo de una subclase se puede utilizar una variable o un método estático sin hacer referencia a la clase en la que se definió.

Atributos de espacio de nombres definido por el usuario

Como alternativa a los atributos de control de acceso predefinidos se puede crear un espacio de nombres personalizado para usarlo como un atributo. Sólo se puede utilizar un atributo de espacio de nombres por cada definición y no se puede utilizar en combinación con uno de los atributos de control de acceso (`public`, `private`, `protected`, `internal`).

Variables

Las variables pueden declararse con las palabras clave `var` o `const`. Es posible cambiar los valores de las variables declaradas con la palabra clave `var` varias veces durante la ejecución de un script. Las variables declaradas con la palabra clave `const` se denominan *constantes* y se les puede asignar valores una sola vez. Un intento de asignar un valor nuevo a una constante inicializada provoca un error.

Variables estáticas

Las variables estáticas se declaran mediante una combinación de la palabra clave `static` y la sentencia `var` o `const`. Las variables estáticas, que se asocian a una clase en lugar de a una instancia de una clase, son útiles para almacenar y compartir información que se aplica a toda una clase de objetos. Por ejemplo, una variable estática es adecuada si se desea almacenar un recuento del número de veces que se crea una instancia de una clase o si se desea almacenar el número máximo de instancias de la clase permitidas.

En el ejemplo siguiente se crea una variable `totalCount` para hacer un seguimiento del número de instancias de clase creadas y una constante `MAX_NUM` para almacenar el número máximo de instancias creadas. Las variables `totalCount` y `MAX_NUM` son estáticas porque contienen valores que se aplican a la clase como un todo en lugar de a una instancia concreta.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

El código externo a la clase StaticVars y a cualquiera de sus subclases sólo puede hacer referencia a las propiedades `totalCount` y `MAX_NUM` a través de la misma clase. Por ejemplo, el siguiente código funciona:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

No se puede acceder a variables estáticas a través de una instancia de la clase, por lo que el código siguiente devuelve errores:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Las variables declaradas con las palabras clave `static` y `const` deben ser inicializadas a la vez que se declara la constante, como hace la clase StaticVars para `MAX_NUM`. No se puede asignar un valor a `MAX_NUM` dentro del constructor o de un método de instancia. El código siguiente generará un error, ya que no es una forma válida de inicializar una constante estática:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Variables de instancia
Las variables de instancia incluyen propiedades declaradas con las palabras clave `var` y `const`, pero sin la palabra clave `static`. Las variables de este tipo, que se asocian a instancias de clase en lugar de a una clase completa, son útiles para almacenar valores específicos de una instancia. Por ejemplo, la clase Array tiene una propiedad de instancia denominada `length` que almacena el número de elementos de conjunto contenidos en una instancia concreta de la clase Array.

En una subclase no se pueden sustituir las variables de instancia, aunque se declaren como `var` o `const`. Sin embargo, se puede obtener una funcionalidad similar a la sustitución de variables sustituyendo métodos de captador y definidor.

Métodos

Los métodos son funciones que forman parte de una definición de clase. Cuando se crea una instancia de la clase, se vincula un método a esa instancia. A diferencia de una función declarada fuera de una clase, un método sólo puede utilizarse desde la instancia a la que está asociado.

Los métodos se definen con la palabra clave `function`. Al igual que sucede con cualquier propiedad de clase, puede aplicar cualquiera de sus atributos a los métodos, incluyendo private, protected, public, internal, static o un espacio de nombres personalizado. Puede utilizar una sentencia de función como la siguiente:

```
public function sampleFunction():String {}
```

También se puede utilizar una variable a la que se asigna una expresión de función, de la manera siguiente:

```
public var sampleFunction:Function = function () {}
```

En la mayoría de los casos se deseará utilizar una sentencia de función en lugar de una expresión de función por los siguientes motivos:

- Las sentencias de función son más concisas y fáciles de leer.
- Permiten utilizar las palabras clave `override` y `final`.
- Crean un vínculo más fuerte entre el identificador (es decir, el nombre de la función) y el código del cuerpo del método. Como es posible cambiar el valor de una variable con una sentencia de asignación, la conexión entre una variable y su expresión de función se puede hacer más fuerte en cualquier momento. Aunque se puede solucionar este problema declarando la variable con `const` en lugar de `var`, esta técnica no se considera una práctica recomendable, ya que hace que el código sea difícil de leer e impide el uso de las palabras clave `override` y `final`.

Un caso en el que hay que utilizar una expresión de función es cuando se elige asociar una función al objeto prototipo.

Métodos constructores

Los métodos constructores, denominados en ocasiones *constructores*, son funciones que comparten el nombre con la clase en la que se definen. Todo el código que se incluya en un método constructor se ejecutará siempre que una instancia de la clase se cree con la palabra clave `new`. Por ejemplo, el código siguiente define una clase simple denominada Example que contiene una sola propiedad denominada `status`. El valor inicial de la variable `status` se establece en la función constructora.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Los métodos constructores sólo pueden ser públicos, pero el uso del atributo `public` es opcional. No se puede utilizar en un constructor ninguno de los otros especificadores de control de acceso, incluidos `private`, `protected` e `internal` . Tampoco se puede utilizar un espacio de nombres definido por el usuario con un método constructor.

Un constructor puede hacer una llamada explícita al constructor de su superclase directa utilizando la sentencia`super()`. Si no se llama explícitamente al constructor de la superclase, el compilador inserta automáticamente una llamada antes de la primera sentencia en el cuerpo del constructor. También se puede llamar a métodos de la superclase mediante el prefijo `super` como una referencia a la superclase. Si se decide utilizar `super()` y `super` en el mismo cuerpo de constructor, hay que asegurarse de llamar primero a `super()`. De lo contrario, la referencia `super` no se comportará de la manera esperada. También se debe llamar al constructor `super()` antes que a cualquier sentencia `throw` o `return`.

En el ejemplo siguiente se ilustra lo que sucede si se intenta utilizar la referencia `super` antes de llamar al constructor `super()`. Una nueva clase, ExampleEx, amplía la clase Example. El constructor de ExampleEx intenta acceder a la variable de estado definida en su superclase, pero lo hace antes de llamar a `super()`. La sentencia `trace()` del constructor de ExampleEx produce el valor `null` porque la variable `status` no está disponible hasta que se ejecuta el constructor `super()`.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Aunque se puede utilizar la sentencia `return` dentro de un constructor, no se permite devolver un valor. Es decir, las sentencias `return` no deben tener expresiones o valores asociados. Por consiguiente, no se permite que los métodos constructores devuelvan valores, lo que significa que no se puede especificar ningún tipo de devolución.

Si no se define un método constructor en la clase, el compilador creará automáticamente un constructor vacío. Si la clase amplía otra clase, el compilador incluirá una llamada `super()` en el constructor que genera.

Métodos estáticos

Los métodos estáticos, también denominados *métodos de clase*, son métodos que se declaran con la palabra clave `static`. Estos métodos, que se asocian a una clase en lugar de a una instancia de clase, son útiles para encapsular la funcionalidad que afecta a algo más que el estado de una instancia individual. Como los métodos estáticos se asocian a una clase como un todo, sólo se puede acceder a dichos métodos a través de una clase, no a través de una instancia de la clase.

Los métodos estáticos son útiles para encapsular la funcionalidad que no se limita a afectar al estado de las instancias de clase. Es decir, un método debe ser estático si proporciona funcionalidad que no afecta directamente al valor de una instancia de clase. Por ejemplo, la clase Date tiene un método estático denominado `parse()`, que convierte una cadena en un número. El método es estático porque no afecta a una instancia individual de la clase. El método `parse()` recibe una cadena que representa un valor de fecha, analiza la cadena y devuelve un número con un formato compatible con la representación interna de un objeto Date. Este método no es un método de instancia porque no tiene sentido aplicar el método a una instancia de la clase Date.

El método `parse()` estático puede compararse con uno de los métodos de instancia de la clase Date, como `getMonth()`. El método `getMonth()` es un método de instancia porque opera directamente en el valor de una instancia recuperando un componente específico, el mes, de una instancia de Date.

Como los métodos estáticos no están vinculados a instancias individuales, no se pueden utilizar las palabras clave `this` o `super` en el cuerpo de un método estático. Las referencias `this` y `super` sólo tienen sentido en el contexto de un método de instancia.

En contraste con otros lenguajes de programación basados en clases, en ActionScript 3.0 los métodos estáticos no se heredan.

Métodos de instancia

Los métodos de instancia son métodos que se declaran sin la palabra clave `static`. Estos métodos, que se asocian a instancias de una clase en lugar de a la clase como un todo, son útiles para implementar funcionalidad que afecta a instancias individuales de una clase. Por ejemplo, la clase Array contiene un método de instancia denominado `sort()`, que opera directamente en instancias de Array.

En el cuerpo de un método de instancia, las variables estáticas y de instancia están dentro del ámbito, lo que significa que se puede hacer referencia a las variables definidas en la misma clase mediante un identificador simple. Por ejemplo, la clase siguiente, CustomArray, amplía la clase Array. La clase CustomArray define una variable estática denominada `arrayCountTotal` para hacer un seguimiento del número total de instancias de clase, una variable de instancia denominada `arrayNumber` que hace un seguimiento del orden en que se crearon las instancias y un método de instancia denominado `getPosition()` que devuelve los valores de estas variables.

```
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);
    }
}
```

Aunque el código externo a la clase debe acceder a la variable estática `arrayCountTotal` a través del objeto de clase mediante `CustomArray.arrayCountTotal`, el código que reside dentro del cuerpo del método `getPosition()` puede hacer referencia directamente a la variable estática `arrayCountTotal`. Esto se cumple incluso para variables estáticas de superclases. Aunque en ActionScript 3.0 las propiedades estáticas no se heredan, las propiedades estáticas de las superclases están dentro del ámbito. Por ejemplo, la clase Array tiene unas pocas variables estáticas, una de las cuales es una constante denominada `DESCENDING`. El código que reside en una subclase de Array puede acceder a la constante estática `DESCENDING` mediante un identificador simple:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

El valor de la referencia `this` en el cuerpo de un método de instancia es una referencia a la instancia a la que está asociado el método. El código siguiente muestra que la referencia `this` señala a la instancia que contiene el método:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

La herencia de los métodos de instancia se puede controlar con las palabras clave `override` y `final`. Se puede utilizar el atributo `override` para redefinir un método heredado y el atributo `final` para evitar que las subclases sustituyan un método.

Métodos descriptores de acceso (captador y definidor)

Las funciones descriptoras de acceso get y set, también denominadas *captadores* y *definidores*, permiten implementar los principios de programación relacionados con la ocultación de información y encapsulación a la vez que ofrecen una interfaz de programación fácil de usar para las clases que se crean. Estas funciones permiten mantener las propiedades de clase como privadas de la clase ofreciendo a los usuarios de la clase acceso a esas propiedades como si accedieran a una variable de clase en lugar de llamar a un método de clase.

La ventaja de este enfoque es que permite evitar las funciones descriptoras de acceso tradicionales con nombres poco flexibles, como `getPropertyName()` y `setPropertyName()`. Otra ventaja de los captadores y definidores es que permiten evitar tener dos funciones públicas por cada propiedad que permita un acceso de lectura y escritura.

La siguiente clase de ejemplo, denominada GetSet, incluye funciones descriptoras de acceso get y set denominadas `publicAccess()` que proporcionan acceso a la variable privada denominada `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Si se intenta directamente acceder a la propiedad `privateProperty` se producirá un error, como se muestra a continuación:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

En su lugar, un usuario de la clase GetSet utilizará algo que parece ser una propiedad denominada `publicAccess`, pero que en realidad es un par de funciones descriptoras de acceso get y set que operan en la propiedad privada denominada `privateProperty`. En el ejemplo siguiente se crea una instancia de la clase GetSet y después se establece el valor de `privateProperty` mediante el descriptor de acceso público denominado `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Las funciones captadoras y definidoras también permiten sustituir propiedades heredadas de una superclase, algo que no es posible al usar variables miembro de clase normales. Las variables miembro de clase que se declaran mediante la palabra clave `var` no se pueden sustituir en una subclase. Sin embargo, las propiedades que se crean mediante funciones captadoras y definidoras no tienen esta restricción. Se puede utilizar el atributo `override` en funciones captadoras y definidoras heredadas de una superclase.

Métodos vinculados

Un método vinculado, a veces denominado *cierre de método*, es simplemente un método que se extrae de su instancia. Los métodos que se pasan como argumentos a una función o se devuelven como valores desde una función son ejemplos de métodos vinculados. Una de las novedades de ActionScript 3.0 es que un método vinculado es similar a un cierre de función, ya que conserva su entorno léxico incluso cuando se extrae de su instancia. Sin embargo, la diferencia clave entre un método vinculado y un cierre de función es que la referencia `this` para un método vinculado permanece vinculada a la instancia que implementa el método. Es decir, la referencia `this` de un método vinculado siempre señala al objeto original que implementó el método. Para los cierres de función, la referencia `this` es genérica, lo que significa que señala al objeto con el que esté relacionada la función cuando se llame.

Es importante comprender los métodos vinculados para utilizar la palabra clave `this`. Debe recordarse que la palabra clave `this` proporciona una referencia al objeto principal de un método. La mayoría de los programadores que utilizan ActionScript esperan que la palabra clave `this` siempre represente el objeto o la clase que contiene la definición de un método. Sin embargo, sin la vinculación de métodos esto no se cumplirá siempre. Por ejemplo, en las versiones anteriores de ActionScript, la referencia `this` no siempre hacía referencia a la instancia que implementaba el método.

En ActionScript 2.0, cuando se extraen métodos de una instancia no sólo no se vincula la referencia `this` a la instancia original, sino que además las variables y los métodos miembro de la clase de la instancia no están disponibles. Esto es no un problema en ActionScript 3.0 porque se crean métodos vinculados automáticamente cuando se pasa un método como parámetro. Los métodos vinculados garantizan que la palabra clave `this` siempre haga referencia al objeto o la clase en que se define un método.

El código siguiente define una clase denominada ThisTest, que contiene un método denominado `foo()` que define el método vinculado y un método denominado `bar()` que devuelve el método vinculado. El código externo a la clase crea una instancia de la clase ThisTest, llama al método `bar()` y almacena el valor devuelto en una variable denominada `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 */
```

Las dos últimas líneas de código muestran que la referencia `this` del método vinculado `foo()` sigue señalando a una instancia de la clase ThisTest, aunque la referencia `this` de la línea inmediatamente anterior señala al objeto global. Además, el método vinculado almacenado en la variable `myFunc` sigue teniendo acceso a las variables miembro de la clase ThisTest. Si se ejecutara este mismo código en ActionScript 2.0, las referencias `this` coincidirían y el valor de la variable `num` sería `undefined`.

La adición de métodos vinculados se aprecia mejor en aspectos como los controladores de eventos, ya que el método `addEventListener()` requiere que se pase una función o un método como un argumento.

Enumeraciones con clases

Las *enumeraciones* son tipos de datos personalizados que se crean para encapsular un pequeño conjunto de valores. ActionScript 3.0 no ofrece una capacidad de enumeración específica, a diferencia de C++, que incluye la palabra clave `enum`, o Java, con su interfaz Enumeration. No obstante, se pueden crear enumeraciones utilizando clases y constantes estáticas. Por ejemplo, la clase PrintJob en ActionScript 3.0 utiliza una enumeración denominada PrintJobOrientation para almacenar el conjunto de valores formado por `"landscape"` y `"portrait"`, como se muestra en el código siguiente:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Por convención, una clase de enumeración se declara con el atributo `final` porque no es necesario ampliarla. La clase sólo contiene miembros estáticos, lo que significa que no se crean instancias de la clase. sino que se accede a los valores de la enumeración directamente a través del objeto de clase, como se indica en el siguiente fragmento de código:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Todas las clases de enumeración en ActionScript 3.0 sólo contienen variables de tipo String, int o uint. La ventaja de utilizar enumeraciones en lugar de valores literales numéricos o de cadena es que es más fácil detectar errores tipográficos con las enumeraciones. Si se escribe incorrectamente el nombre de una enumeración, el compilador de ActionScript genera un error. Si se utilizan valores literales, el compilador no mostrará ninguna advertencia en caso de que encuentre una palabra escrita incorrectamente o se utilice un número incorrecto. En el ejemplo anterior, el compilador genera un error si el nombre de la constante de enumeración es incorrecto, como se indica en el siguiente fragmento:

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Sin embargo, el compilador no generará un error si se escribe incorrectamente un valor literal de cadena, como se muestra a continuación:

```
    if (pj.orientation == "portrai") // no compiler error
```

Otra técnica para crear enumeraciones también implica crear una clase independiente con propiedades estáticas para la enumeración. No obstante, esta técnica difiere en que cada una de las propiedades estáticas contiene una instancia de la clase en lugar de una cadena o un valor entero. Por ejemplo, el código siguiente crea una clase de enumeración para los días de la semana:

```
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();
}
```

En ActionScript 3.0 no se emplea esta técnica, pero la utilizan muchos desarrolladores que prefieren la verificación de tipos mejorada que proporciona. Por ejemplo, un método que devuelve un valor de enumeración puede restringir el valor devuelto al tipo de datos de la enumeración. El código siguiente muestra, además de una función que devuelve un día de la semana, una llamada a función que utiliza el tipo de la enumeración como una anotación de 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();
```

También se puede mejorar la clase Day de forma que asocie un entero con cada día de la semana y proporcione un método `toString()` que devuelva una representación de cadena del día.

Clases de activos incorporados

ActionScript 3.0 utiliza clases especiales, denominadas *clases de activos incorporados*, para representar elementos incorporados. Un *activo incorporado* es un activo, como un sonido, una imagen o una fuente, que se incluye en un archivo SWF en tiempo de compilación. Incorporar un activo en lugar de cargarlo dinámicamente garantiza que estará disponible en tiempo de ejecución, pero a cambio el tamaño del archivo SWF será mayor.

Uso de clases de activos incorporados en Flash Professional

Para incorporar un activo, hay que añadir primero el activo a una biblioteca de archivo FLA. A continuación, hay que usar la propiedad linkage del activo para asignar un nombre a la clase de activo incorporado del activo. Si no se encuentra una clase con ese nombre en la ruta de clases, se generará una clase automáticamente. Después se puede utilizar una instancia de la clase de activo incorporado y utilizar las propiedades y los métodos definidos por la clase. Por ejemplo, se puede utilizar el código siguiente para reproducir un sonido incorporado vinculado a una clase de activo incorporado denominada PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

De forma alternativa, se puede utilizar la etiqueta de metadatos `[Embed]` para incorporar activos en un proyecto de Flash Professional, tal y como se describe a continuación. Si se utiliza la etiqueta `[Embed]` en el código, Flash Professional emplea el compilador Flex para compilar el proyecto en lugar del compilador Flash Professional.

Uso de clases de activos incorporados mediante el compilador Flex

Si se está compilando código con el compilador Flex, para incorporar un activo al código de ActionScript, utilice la etiqueta de metadatos `[Embed]`. Coloque el activo en la carpeta principal de código fuente o en otra carpeta situada en la ruta de compilación del proyecto. Cuando el compilador Flex encuentra una etiqueta de metadatos Embed, crea automáticamente la clase de activo incorporado. Puede acceder a la clase a través una variable de tipo de datos Class que se declara de forma inmediata siguiendo la etiqueta de metadatos `[Embed]`. Por ejemplo, el siguiente código incorpora un sonido denominado sound1.mp3 y utiliza una variable llamada `soundCls` para almacenar una referencia a la clase de activo incorporado asociada al sonido. Posteriormente el ejemplo crea una instancia de la clase de activo incorporado y llama al método `play()` en dicha instancia:

```
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

Para utilizar la etiqueta de metadatos `[Embed]` en un proyecto de ActionScript de Flash Builder, debe importar todas las clases necesarias de la arquitectura Flex. Por ejemplo, para incorporar sonidos, debe importar la clase mx.core.SoundAsset. Para utilizar la arquitectura Flex, incluya el archivo framework.swc en la ruta de compilación de ActionScript. Con ello aumentará el tamaño del archivo SWF.

Adobe Flex

Otra posibilidad consiste en incorporar en Flex un activo con la directiva `@Embed()` en una definición de etiqueta MXML.

Interfaces

Una interfaz es una colección de declaraciones de métodos que permite la comunicación entre objetos que no están relacionados. Por ejemplo, ActionScript 3.0 define la interfaz IEventDispatcher, que contiene declaraciones de métodos que una clase puede utilizar para controlar objetos de evento. La interfaz IEventDispatcher establece una forma estándar de pasar objetos de evento entre objetos. El código siguiente muestra la definición de la interfaz 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;
}
```

Las interfaces se basan en la distinción entre la interfaz de un método y su implementación. La interfaz de un método incluye toda la información necesaria para llamar a dicho método, como el nombre del método, todos sus parámetros y el tipo que devuelve. La implementación de un método no sólo incluye la información de la interfaz, sino también las sentencias ejecutables que implementan el comportamiento del método. Una definición de interfaz sólo contiene interfaces de métodos; cualquier clase que implemente la interfaz debe encargarse de definir las implementaciones de los métodos.

En ActionScript 3.0, la clase EventDispatcher implementa la interfaz IEventDispatcher definiendo todos los métodos de la interfaz IEventDispatcher y añadiendo el código a cada uno de los métodos. El código siguiente es un fragmento de la definición de la clase EventDispatcher:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

La interfaz IEventDispatcher constituye un protocolo que las instancias de EventDispatcher utilizan para procesar objetos de evento y pasárselos a otros objetos que también tienen implementada la interfaz IEventDispatcher.

Otra forma de describir una interfaz es decir que define un tipo de datos de la misma manera que una clase. Por consiguiente, una interfaz se puede utilizar como una anotación de tipo, igual que una clase. Al ser un tipo de datos, una interfaz también se puede utilizar con operadores, como los operadores `is` y `as`, que requieren un tipo de datos. Sin embargo, a diferencia de una clase, no se puede crear una instancia de una interfaz. Esta distinción hace que muchos programadores consideren que las interfaces son tipos de datos abstractos y las clases son tipos de datos concretos.

Definición de una interfaz

La estructura de una definición de interfaz es similar a la de una definición de clase, con la diferencia de que una interfaz sólo puede contener métodos sin código de método. Las interfaces no pueden incluir variables ni constantes, pero pueden incluir captadores y definidores. Para definir una interfaz se utiliza la palabra clave `interface`. Por ejemplo, la siguiente interfaz, IExternalizable, forma parte del paquete flash.utils en ActionScript 3.0. La interfaz IExternalizable define un protocolo para serializar un objeto, lo que implica convertir un objeto en un formato adecuado para el almacenamiento en un dispositivo o para el transporte a través de la red.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

La interfaz IExternalizable se declara con el modificador de control de acceso `public`. Las definiciones de interfaz sólo pueden modificarse mediante los especificadores de control de acceso `public` e `internal`. Las declaraciones de métodos en una definición de interfaz no pueden incluir ningún especificador de control de acceso.

ActionScript 3.0 sigue una convención por la que los nombres de interfaz empiezan por una `I` mayúscula, pero se puede utilizar cualquier identificador válido como nombre de interfaz. Las definiciones de interfaz se suelen colocar en el nivel superior de un paquete. No pueden colocarse en una definición de clase ni en otra definición de interfaz.

Las interfaces pueden ampliar una o más interfaces. Por ejemplo, la siguiente interfaz, IExample, amplía la interfaz IExternalizable:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Cualquier clase que implemente la interfaz IExample debe incluir implementaciones no sólo para el método `extra()`, sino también para los métodos `writeExternal()` y `readExternal()` heredados de la interfaz IExternalizable.

Implementación de una interfaz en una clase

Una clase es el único elemento del lenguaje ActionScript 3.0 que puede implementar una interfaz. Se puede utilizar la palabra clave `implements` en una declaración de clase para implementar una o más interfaces. En el ejemplo siguiente se definen dos interfaces, IAlpha e IBeta, y una clase, Alpha, que implementa las dos interfaces:

```
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 {}
}
```

En una clase que implementa una interfaz, los métodos implementados deben:

- Utilizar el identificador de control de acceso `public`.
- Utilizar el mismo nombre que el método de interfaz.
- Tener el mismo número de parámetros, cada uno con un tipo de datos que coincida con los tipos de datos de los parámetros del método de interfaz.
- Devolver el mismo tipo de datos.

```
public function foo(param:String):String {}
```

Sin embargo, hay cierta flexibilidad para asignar nombres a los parámetros de métodos implementados. Aunque el número de parámetros y el tipo de datos de cada parámetro del método implementado deben coincidir con los del método de interfaz, los nombres de los parámetros no tienen que coincidir. Por ejemplo, en el ejemplo anterior el parámetro del método `Alpha.foo()` se denomina `param`:

En el método de interfaz `IAlpha.foo()`, el parámetro se denomina `str`:

```
function foo(str:String):String;
```

También hay cierta flexibilidad con los valores predeterminados de parámetros. Una definición de interfaz puede incluir declaraciones de funciones con valores predeterminados de parámetros. Un método que implementa una declaración de función de este tipo debe tener un valor predeterminado de parámetro que sea miembro del mismo tipo de datos que el valor especificado en la definición de interfaz, pero el valor real no tiene que coincidir. Por ejemplo, el código siguiente define una interfaz que contiene un método con un valor predeterminado de parámetro igual a 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

La siguiente definición de clase implementa la interfaz Igamma, pero utiliza un valor predeterminado de parámetro distinto:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

La razón de esta flexibilidad es que las reglas para implementar una interfaz se han diseñado específicamente para garantizar la compatibilidad de los tipos de datos y no es necesario exigir que los nombres de parámetros y los valores predeterminados de los parámetros sean idénticos para alcanzar ese objetivo.

Herencia

La herencia es una forma de reutilización de código que permite a los programadores desarrollar clases nuevas basadas en clases existentes. Las clases existentes se suelen denominar *clases base* o *superclases*, y las clases nuevas se denominan *subclases*. Una ventaja clave de la herencia es que permite reutilizar código de una clase base manteniendo intacto el código existente. Además, la herencia no requiere realizar ningún cambio en la interacción entre otras clases y la clase base. En lugar de modificar una clase existente que puede haber sido probada minuciosamente o que ya se está utilizando, la herencia permite tratar esa clase como un módulo integrado que se puede ampliar con propiedades o métodos adicionales. Se utiliza la palabra clave `extends` para indicar que una clase hereda de otra clase.

La herencia también permite beneficiarse del *polimorfismo* en el código. El polimorfismo es la capacidad de utilizar un solo nombre de método para un método que se comporta de distinta manera cuando se aplica a distintos tipos de datos. Un ejemplo sencillo es una clase base denominada Shape con dos subclases denominadas Circle y Square. La clase Shape define un método denominado `area()` que devuelve el área de la forma. Si se implementa el polimorfismo, se puede llamar al método `area()` en objetos de tipo Circle y Square, y se harán automáticamente los cálculos correctos. La herencia activa el polimorfismo al permitir que las subclases hereden y redefinan (*sustituyan*) los métodos de la clase base. En el siguiente ejemplo, se redefine el método `area()` mediante las clases Circle y 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
```

Como cada clase define un tipo de datos, el uso de la herencia crea una relación especial entre una clase base y una clase que la amplía. Una subclase posee todas las propiedades de su clase base, lo que significa que siempre se puede sustituir una instancia de una subclase como una instancia de la clase base. Por ejemplo, si un método define un parámetro de tipo Shape, se puede pasar un argumento de tipo Circle, ya que Circle amplía Shape, como se indica a continuación:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Herencia y propiedades de instancia

Todas las subclases heredan una propiedad de instancia definida con la palabra clave `function`, `var` o `const`, con tal de que no se haya declarado la propiedad con el atributo `private` en la clase base. Por ejemplo, la clase Event en ActionScript 3.0 tiene diversas subclases que heredan propiedades comunes a todos los objetos de evento.

Para algunos tipos de eventos, la clase Event contiene todas las propiedades necesarias para definir el evento. Estos tipos de eventos no requieren más propiedades de instancia que las definidas en la clase Event. Algunos ejemplos de estos eventos son el evento `complete`, que se produce cuando los datos se han cargado correctamente, y el evento `connect`, que se produce cuando se establece una conexión de red.

El ejemplo siguiente es un fragmento de la clase Event que muestra algunas de las propiedades y los métodos que heredarán las subclases. Como las propiedades se heredan, una instancia de cualquier subclase puede acceder a estas propiedades.

```
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 {}
    ...
}
```

Otros tipos de eventos requieren propiedades únicas que no están disponibles en la clase Event. Estos eventos se definen creando subclases de la clase Event para añadir nuevas propiedades a las propiedades definidas en la clase Event. Un ejemplo de una subclase de este tipo es la clase MouseEvent, que añade propiedades únicas a eventos asociados con el movimiento o los clics del ratón, como los eventos `mouseMove` y `click`. El ejemplo siguiente es un fragmento de la clase MouseEvent que muestra la definición de propiedades que se han añadido a la subclase y, por tanto, no existen en la clase 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 {}
    ...
}
```

Herencia y especificadores de control de acceso

Si una propiedad se declara con la palabra clave `public`, estará visible en cualquier parte del código. Esto significa que la palabra clave `public`, a diferencia de las palabras clave `private`, `protected` e `internal`, no restringe de ningún modo la herencia de propiedades.

Si se declara una propiedad con la palabra clave `private`, sólo será visible en la clase que la define, lo que significa que ninguna subclase la heredará. Este comportamiento es distinto del de las versiones anteriores de ActionScript, en las que el comportamiento de la palabra clave `private` era más parecido al de la palabra clave `protected` de ActionScript 3.0.

La palabra clave `protected` indica que una propiedad es visible en la clase que la define y en todas sus subclases. A diferencia de la palabra clave `protected` del lenguaje de programación Java, la palabra clave `protected` de ActionScript 3.0 no hace que una propiedad esté visible para todas las clases de un mismo paquete. En ActionScript 3.0 sólo las subclases pueden acceder a una propiedad declarada con la palabra clave `protected.` Además, una propiedad protegida estará visible para una subclase tanto si la subclase está en el mismo paquete que la clase base como si está en un paquete distinto.

Para limitar la visibilidad de una propiedad al paquete en el que se define, se debe utilizar la palabra clave `internal` o no utilizar ningún especificador de control de acceso. Cuando no se especifica ninguno especificador de control de acceso, el predeterminado es `internal`. Una propiedad marcada como `internal` sólo podrá ser heredada por una subclase que resida en el mismo paquete.

Se puede utilizar el ejemplo siguiente para ver cómo afecta cada uno de los especificadores de control de acceso a la herencia más allá de los límites del paquete. El código siguiente define una clase principal de aplicación denominada AccessControl y otras dos clases denominadas Base y Extender. La clase Base está en un paquete denominado foo y la clase Extender, que es una subclase de la clase Base, está en un paquete denominado bar. La clase AccessControl sólo importa la clase Extender y crea una instancia de Extender que intenta acceder a una variable denominada `str` definida en la clase Base. La variable `str` se declara como `public` de forma que el código se compile y ejecute como se indica en el siguiente fragmento:

```
// 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
        }
    }
}
```

Para ver cómo afectan los otros especificadores de control de acceso a la compilación y la ejecución del ejemplo anterior, se debe cambiar el especificador de control de acceso de la variable `str` a `private`, `protected` o `internal` después de eliminar o marcar como comentario la línea siguiente de la clase `AccessControl`:

```
trace(myExt.str);// error if str is not public
```

No se permite la sustitución de variables

Las propiedades declaradas con la palabra clave `var` o `const` se pueden heredar, pero no se pueden sustituir. Para sustituir una propiedad hay que redefinirla en una subclase. El único tipo de propiedad que se puede sustituir son los descriptores de acceso get y set (es decir, propiedades declaradas con la palabra clave `function`). Aunque no es posible sustituir una variable de instancia, se puede obtener una funcionalidad similar creando métodos captador y definidor para la variable de instancia y sustituyendo los métodos.

Sustitución de métodos

Para sustituir un método hay que redefinir el comportamiento de un método heredado. Los métodos estáticos no se heredan y no se pueden sustituir. Sin embargo, las subclases heredan los métodos de instancia, que se pueden sustituir con tal de que se cumplan los dos criterios siguientes:

- El método de instancia no se ha declarado con la palabra clave `final` en la clase base. Si se utiliza la palabra clave `final` con un método de instancia, indica la intención del programador de evitar que las subclases sustituyan el método.
- El método de instancia no se declara con el especificador de control de acceso `private` de la clase base. Si se marca un método como `private` en la clase base, no hay que usar la palabra clave `override` al definir un método con el mismo nombre en la subclase porque el método de la clase base no será visible para la subclase.

  Para sustituir un método de instancia que cumpla estos criterios, la definición del método en la subclase debe utilizar la palabra clave `override` y debe coincidir con la versión de la superclase del método en los siguientes aspectos:

- El método sustituto debe tener el mismo nivel de control de acceso que el método de la clase base. Los métodos marcados como internal tienen el mismo nivel de control de acceso que los métodos que no tienen especificador de control de acceso.
- El método sustituto debe tener el mismo número de parámetros que el método de la clase base.
- Los parámetros del método sustituto deben tener las mismas anotaciones de tipo de datos que los parámetros del método de la clase base.
- El método sustituto debe devolver el mismo tipo de datos que el método de la clase base.

Sin embargo, los nombres de los parámetros del método sustituto no tienen que coincidir con los nombres de los parámetros de la clase base, con tal de que el número de parámetros y el tipo de datos de cada parámetro coincidan.

La sentencia super

Al sustituir un método, los programadores generalmente desean añadir funcionalidad al comportamiento del método de la superclase que van a sustituir, en lugar de sustituir completamente el comportamiento. Esto requiere un mecanismo que permita a un método de una subclase llamar a la versión de sí mismo de la superclase. La sentencia `super` proporciona este mecanismo, ya que contiene una referencia a la superclase inmediata. El ejemplo siguiente define una clase denominada Base que contiene un método denominado `thanks()` y una subclase de la clase Base denominada Extender que reemplaza el método `thanks()`. El método `Extender.thanks()` utiliza la sentencia `super` para llamar a `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";
    }
}
```

Sustitución de captadores y definidores

Aunque las variables definidas en una superclase no se pueden sustituir, sí se pueden sustituir los captadores y definidores. Por ejemplo, el código siguiente sustituye un captador denominado `currentLabel` definido en la clase MovieClip en 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;
        }
    }
}
```

El resultado de la sentencia `trace()` en el constructor de la clase OverrideExample es `Override: null`, lo que indica que el ejemplo pudo sustituir la propiedad heredada `currentLabel`.

Propiedades estáticas no heredadas

Las subclases no heredan las propiedades estáticas. Esto significa que no se puede acceder a las propiedades estáticas a través de una instancia de una subclase. Sólo se puede acceder a una propiedad estática a través del objeto de clase en el que está definida. Por ejemplo, en el código siguiente se define una clase base denominada Base y una subclase que amplía Base denominada Extender. Se define una variable estática denominada `test` en la clase Base. El código del siguiente fragmento no se compila en modo estricto y genera un error en tiempo de ejecución en modo estándar.

```
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 { }
```

La única manera de acceder a la variable estática `test` es a través del objeto de clase, como se indica en el código siguiente:

```
Base.test;
```

No obstante, se puede definir una propiedad de instancia con el mismo nombre que una propiedad estática. Esta propiedad de instancia puede definirse en la misma clase que la propiedad estática o en una subclase. Por ejemplo, la clase Base del ejemplo anterior podría tener una propiedad de instancia denominada `test`. El código siguiente se compila y ejecuta, ya que la propiedad de instancia es heredada por la clase Extender. El código también se compilará y ejecutará si la definición de la variable de instancia de prueba se mueve (en lugar de copiarse) a la clase 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 {}
```

Propiedades estáticas y cadena de ámbitos

Aunque las propiedades estáticas no se heredan, están en la cadena de ámbitos de la clase en la que se definen y de cualquier subclase de dicha clase. Así, se dice que las propiedades estáticas están *dentro del ámbito* de la clase en la que se definen y de sus subclases. Esto significa que una propiedad estática es directamente accesible en el cuerpo de la clase en la que se define y en cualquier subclase de dicha clase.

En el ejemplo siguiente se modifican las clases definidas en el ejemplo anterior para mostrar que la variable estática `test` definida en la clase Base está en el ámbito de la clase Extender. Es decir, la clase Extender puede acceder a la variable estática `test` sin añadir a la variable el nombre de la clase que define `test` como prefijo.

```
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
    }

}
```

Si se define una propiedad de instancia que utiliza el mismo nombre que una propiedad estática en la misma clase o en una superclase, la propiedad de instancia tiene precedencia superior en la cadena de ámbitos. Se dice que la propiedad de instancia *oculta* la propiedad estática, lo que significa que se utiliza el valor de la propiedad de instancia en lugar del valor de la propiedad estática. Por ejemplo, el código siguiente muestra que si la clase Extender define una variable de instancia denominada `test`, la sentencia `trace()` utiliza el valor de la variable de instancia en lugar del valor de la variable estática:

```
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
    }

}
```

Temas avanzados

Historia de la implementación de la programación orientada a objetos en ActionScript

Como el diseño de ActionScript 3.0 se ha basado en las versiones anteriores de ActionScript, puede resultar útil comprender la evolución del modelo de objetos de ActionScript. ActionScript empezó siendo un mecanismo sencillo de creación de scripts para las primeras versiones de Flash Professional. Con el tiempo, los programadores empezaron a crear aplicaciones cada vez más complejas con ActionScript. En respuesta a las necesidades de los programadores, en cada nueva versión se añadieron características al lenguaje para facilitar la creación de aplicaciones complejas.

ActionScript 1.0

ActionScript 1.0 es la versión del lenguaje utilizada en Flash Player 6 y en versiones anteriores. En esta fase inicial del desarrollo, el modelo de objetos de ActionScript ya se basaba en el concepto de objeto como tipo de datos básico. Un objeto de ActionScript es un tipo de datos compuesto con un conjunto de *propiedades*. Al describir el modelo de objetos, el término *propiedades* abarca todo lo que está asociado a un objeto, como las variables, las funciones o los métodos.

Aunque esta primera generación de ActionScript no permitía definir clases con una palabra clave `class`, se podía definir una clase mediante un tipo de objeto especial denominado objeto prototipo. En lugar de utilizar una palabra clave `class` para crear una definición de clase abstracta a partir de la cual se puedan crear instancias de objetos, como se hace en otros lenguajes basados en clases como Java y C++, los lenguajes basados en prototipos como ActionScript 1.0 utilizan un objeto existente como modelo (o prototipo) para crear otros objetos. En un lenguaje basado en clases, los objetos pueden señalar a una clase como su plantilla; en cambio, en un lenguaje basado en prototipos los objetos señalan a otro objeto, el prototipo, que es su plantilla.

Para crear una clase en ActionScript 1.0, se define una función constructora para dicha clase. En ActionScript, las funciones son objetos reales, no sólo definiciones abstractas. La función constructora que se crea sirve como objeto prototipo para las instancias de dicha clase. El código siguiente crea una clase denominada Shape y define una propiedad denominada `visible` establecida en `true` de manera predeterminada:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Esta función constructora define una clase Shape de la que se puede crear una instancia mediante el operador `new`, de la manera siguiente:

```
myShape = new Shape();
```

De la misma manera que el objeto de función constructora de `Shape()` es el prototipo para instancias de la clase Shape, también puede ser el prototipo para subclases de Shape (es decir, otras clases que amplíen la clase Shape).

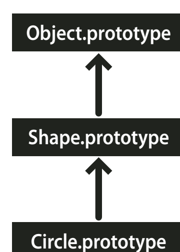
La creación de una clase que es una subclase de la clase Shape es un proceso en dos pasos. En primer lugar debe crearse la clase definiendo una función constructora, de la manera siguiente:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

En segundo lugar, se utiliza el operador `new` para declarar que la clase Shape es el prototipo para la clase Circle. De manera predeterminada, cualquier clase que se cree utilizará la clase Object como prototipo, lo que significa que `Circle.prototype` contiene actualmente un objeto genérico (una instancia de la clase Object). Para especificar que el prototipo de Circle es Shape y no Object, se debe utilizar el código siguiente para cambiar el valor de `Circle.prototype` de forma que contenga un objeto Shape en lugar de un objeto genérico:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

La clase Shape y la clase Circle ahora están vinculadas en una relación de herencia que se suele llamar *cadena de prototipos*. El diagrama ilustra las relaciones de una cadena de prototipos:



La clase base al final de cada cadena de prototipos es la clase Object. La clase Object contiene una propiedad estática denominada `Object.prototype` que señala al objeto prototipo base para todos los objetos creados en ActionScript 1.0. El siguiente objeto de la cadena de prototipos de ejemplo es el objeto Shape. Esto se debe a que la propiedad `Shape.prototype` no se ha establecido explícitamente, por lo que sigue conteniendo un objeto genérico (una instancia de la clase Object). El vínculo final de esta cadena es la clase Circle, que está vinculada a su prototipo, la clase Shape (la propiedad `Circle.prototype` contiene un objeto Shape).

Si se crea una instancia de la clase Circle, como en el siguiente ejemplo, la instancia hereda la cadena de prototipos de la clase Circle:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Antes se creó una propiedad denominada `visible` como un miembro de la clase Shape. En este ejemplo, la propiedad `visible` no existe como parte del objeto `myCircle`, sólo existe como miembro del objeto Shape; sin embargo, la siguiente línea de código devuelve `true`:

```
trace(myCircle.visible); // output: true
```

El motor de ejecución puede determinar que el objeto `myCircle` hereda la propiedad `visible` recorriendo la cadena de prototipos. Al ejecutar este código, el motor de ejecución busca primero una propiedad denominada `visible` en las propiedades del objeto `myCircle`, pero no la encuentra. A continuación, busca en el objeto `Circle.prototype`, pero sigue sin encontrar una propiedad denominada `visible`. Sigue por la cadena de prototipos hasta que encuentra la propiedad `visible` definida en el objeto `Shape.prototype` y devuelve el valor de dicha propiedad.

Para simplificar se omiten muchos de los detalles y las complejidades de la cadena de prototipos. El objetivo es proporcionar información suficiente para comprender el modelo de objetos de ActionScript 3.0.

ActionScript 2.0

En ActionScript 2.0 se incluyeron palabras clave nuevas, como `class`, `extends`, `public` y `private`, que permitían definir clases de una manera familiar para cualquiera que trabaje con lenguajes basados en clases como Java y C++. Es importante saber que el mecanismo de herencia subyacente no ha cambiado entre ActionScript 1.0 y ActionScript 2.0. En ActionScript 2.0 simplemente se ha añadido una nueva sintaxis para la definición de clases. La cadena de prototipos funciona de la misma manera en ambas versiones del lenguaje.

La nueva sintaxis introducida en ActionScript 2.0, que se muestra en el siguiente fragmento, permite definir clases de una manera más intuitiva para muchos programadores:

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

En ActionScript 2.0 también se introdujeron las anotaciones de tipos de datos para la verificación de tipos en tiempo de compilación. Esto permite declarar que la propiedad `visible` del ejemplo anterior sólo debe contener un valor booleano. La nueva palabra clave `extends` también simplifica el proceso de crear una subclase. En el siguiente ejemplo, el proceso que requería dos pasos en ActionScript 1.0 se realiza con un solo paso mediante la palabra clave `extends`:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

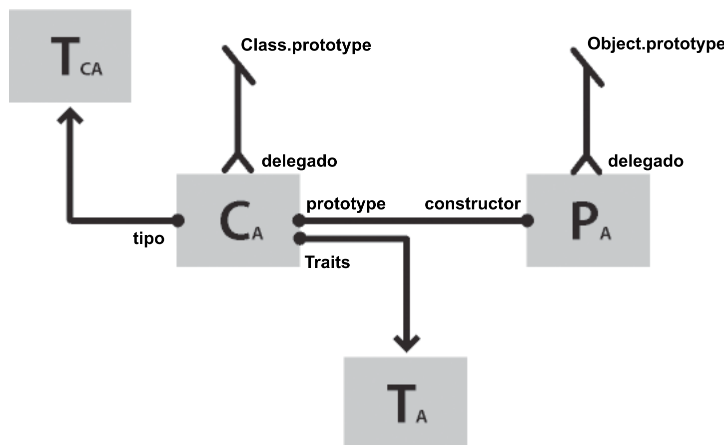
Ahora el constructor se declara como parte de la definición de clase y las propiedades de clase `id` y `radius` también deben declararse explícitamente.

En ActionScript 2.0 también se permite definir interfaces, lo que permite refinar los programas orientados a objetos con protocolos definidos formalmente para la comunicación entre objetos.

El objeto de clase de ActionScript 3.0

Un paradigma común de la programación orientada a objetos, que se suele asociar con Java y C++, utiliza las clases para definir tipos de objetos. Los lenguajes de programación que adoptan este paradigma también tienden a utilizar clases para crear instancias del tipo de datos definido por la clase. ActionScript utiliza clases para ambos propósitos, pero sus raíces como lenguaje basado en prototipos le añaden una característica interesante. ActionScript crea para cada definición de clase un objeto de clase especial que permite compartir el comportamiento y el estado. Sin embargo, para muchos programadores que utilizan ActionScript, esta distinción puede no tener ninguna consecuencia práctica en la programación. ActionScript 3.0 se ha diseñado de forma que se puedan crear sofisticadas aplicaciones orientadas a objetos sin tener que utilizar, ni si quiera comprender, estos objetos de clase especiales.

En el diagrama siguiente se muestra la estructura de un objeto de clase que representa una clase simple denominada A que se define con la sentencia `class A {}`:



Cada rectángulo del diagrama representa un objeto. Cada objeto del diagrama tiene un carácter de subíndice para representar que pertenece a la clase A. El objeto de clase (CA) contiene referencias a una serie de otros objetos importantes. Un objeto traits de instancia (TA) almacena las propiedades de instancia definidas en una definición de clase. Un objeto traits de clase (TCA) representa el tipo interno de la clase y almacena las propiedades estáticas definidas por la clase (el carácter de subíndice C significa “class”). El objeto prototipo (PA) siempre hace referencia al objeto de clase al que se asoció originalmente mediante la propiedad `constructor`.

El objeto traits

El objeto traits, que es una novedad en ActionScript 3.0, se implementó para mejorar el rendimiento. En versiones anteriores de ActionScript, la búsqueda de nombres era un proceso lento, ya que Flash Player recorría la cadena de prototipos. En ActionScript 3.0, la búsqueda de nombres es mucho más rápida y eficaz, ya que las propiedades heredadas se copian desde las superclases al objeto traits de las subclases.

No se puede acceder directamente al objeto traits desde el código, pero las mejoras de rendimiento y de uso de la memoria reflejan el efecto que produce. El objeto traits proporciona a la AVM2 información detallada sobre el diseño y el contenido de una clase. Con este conocimiento, la AVM2 puede reducir considerablemente el tiempo de ejecución, ya que generalmente podrá generar instrucciones directas de código máquina para acceder a las propiedades o llamar a métodos directamente sin tener que realizar una búsqueda lenta de nombres.

Gracias al objeto traits, la huella en la memoria de un objeto puede ser considerablemente menor que la de un objeto similar en las versiones anteriores de ActionScript. Por ejemplo, si una clase está cerrada (es decir, no se declara como dinámica), una instancia de la clase no necesita una tabla hash para propiedades añadidas dinámicamente y puede contener poco más que un puntero a los objetos traits y espacio para las propiedades fijas definidas en la clase. En consecuencia, un objeto que requería 100 bytes de memoria en ActionScript 2.0 puede requerir tan sólo 20 bytes de memoria en ActionScript 3.0.

***Nota:** el objeto traits es un detalle de implementación interna y no hay garantías de que no cambie o incluso desaparezca en versiones futuras de ActionScript.*

El objeto prototype

Cada clase de objeto de ActionScript tiene una propiedad denominada `prototype`, que es una referencia al objeto prototipo de la clase. El objeto prototipo es un legado de las raíces de ActionScript como lenguaje basado en prototipos. Para obtener más información, consulte Historia de la implementación de la programación orientada a objetos en ActionScript.

La propiedad `prototype` es de sólo lectura, lo que significa que no se puede modificar para que señale a otros objetos. Esto ha cambiado con respecto a la propiedad `prototype` de una clase en las versiones anteriores de ActionScript, en las que se podía reasignar el prototipo de forma que señalara a una clase distinta. Aunque la propiedad `prototype` es de sólo lectura, el objeto prototipo al que hace referencia no lo es. Es decir, se pueden añadir propiedades nuevas al objeto prototipo. Las propiedades añadidas al objeto prototipo son compartidas por todas las instancias de la clase.

La cadena de prototipos, que fue el único mecanismo de herencia en versiones anteriores de ActionScript, sirve únicamente como función secundaria en ActionScript 3.0. El mecanismo de herencia principal, herencia de propiedades fijas, se administra internamente mediante el objeto traits. Una propiedad fija es una variable o un método que se define como parte de una definición de clase. La herencia de propiedades fijas también se denomina herencia de clase porque es el mecanismo de herencia asociado con palabras clave como `class`, `extends` y `override`.

La cadena de prototipos proporciona un mecanismo de herencia alternativo que es más dinámico que la herencia de propiedades fijas. Se pueden añadir propiedades a un objeto prototipo de la clase no sólo como parte de la definición de clase, sino también en tiempo de ejecución mediante la propiedad `prototype` del objeto de clase. Sin embargo, hay que tener en cuenta que si se establece el compilador en modo estricto, es posible que no se pueda acceder a propiedades añadidas a un objeto prototipo a menos que se declare una clase con la palabra clave `dynamic`.

La clase Object es un buen ejemplo de clase con varias propiedades asociadas al objeto prototipo. Los métodos `toString()` y `valueOf()` de la clase Object son en realidad funciones asignadas a propiedades del objeto prototipo de la clase Object. A continuación se muestra un ejemplo del aspecto que tendría en teoría la declaración de estos métodos (la implementación real difiere ligeramente a causa de los detalles de implementación):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Como se mencionó anteriormente, se puede asociar una propiedad a un objeto prototipo de clase fuera de la definición de clase. Por ejemplo, el método `toString()` también se puede definir fuera de la definición de clase Object, de la manera siguiente:

```
Object.prototype.toString = function()
{
    // statements
};
```

Sin embargo, a diferencia de la herencia de propiedades fijas, la herencia de prototipo no requiere la palabra clave `override` si se desea volver a definir un método en una subclase. Por ejemplo, si se desea volver a definir el método `valueOf()` en una subclase de la clase Object, hay tres opciones. En primer lugar, se puede definir un método `valueOf()` en el objeto prototipo de la subclase, dentro de la definición de la clase. El código siguiente crea una subclase de Object denominada Foo y redefine el método `valueOf()` en el objeto prototipo de Foo como parte de la definición de clase. Como todas las clases heredan de Object, no es necesario utilizar la palabra clave `extends`.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

En segundo lugar, se puede definir un método `valueOf()` en el objeto prototipo de Foo, fuera de la definición de clase, como se indica en el código siguiente:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Por último, se puede definir una propiedad fija denominada `valueOf()` como parte de la clase Foo. Esta técnica difiere de las otras en que mezcla la herencia de propiedades fijas con la herencia de prototipo. Cualquier subclase de Foo que vaya a redefinir `valueOf()` debe utilizar la palabra clave `override`. El código siguiente muestra `valueOf()` definido como una propiedad fija en Foo:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

El espacio de nombres AS3

La existencia de dos mecanismos de herencia independientes, la herencia de propiedades fijas y la herencia de prototipo, crea un reto interesante para la compatibilidad con respecto a las propiedades y los métodos de las clases principales. La compatibilidad con la especificación del lenguaje ECMAScript en la que se basa ActionScript, requiere utilizar herencia de prototipo, lo que significa que las propiedades y los métodos de una clase principal se definen en el objeto prototipo de esa clase. Por otra parte, la compatibilidad con ActionScript 3.0 requiere utilizar la herencia de propiedades fijas, lo que significa que las propiedades y los métodos de una clase principal se definen en la definición de clase mediante las palabras clave `const`, `var` y `function`. Además, el uso de propiedades fijas en lugar de las versiones de prototipos puede proporcionar un aumento considerable de rendimiento en tiempo de ejecución.

ActionScript 3.0 resuelve este problema utilizando tanto la herencia de prototipo como la herencia de propiedades fijas para las clases principales. Cada clase principal contiene dos conjuntos de propiedades y métodos. Un conjunto se define en el objeto prototipo por compatibilidad con la especificación ECMAScript y el otro conjunto se define con propiedades fijas y el espacio de nombres AS3.0 por compatibilidad con ActionScript 3.0.

El espacio de nombres AS3 proporciona un cómodo mecanismo para elegir entre los dos conjuntos de propiedades y métodos. Si no se utiliza el espacio de nombres AS3, una instancia de una clase principal hereda las propiedades y métodos definidos en el objeto prototipo de la clase principal. Si se decide utilizar el espacio de nombres AS3, una instancia de una clase principal hereda las versiones de AS3, ya que siempre se prefieren las propiedades fijas a las propiedades de prototipo. Es decir, siempre que una propiedad fija está disponible, se utilizará en lugar de una propiedad de prototipo con el mismo nombre.

Se puede utilizar de forma selectiva la versión del espacio de nombres AS3 de una propiedad o un método calificándolo con el espacio de nombres AS3. Por ejemplo, el código siguiente utiliza la versión AS3 del método `Array.pop()`:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Como alternativa, se puede utilizar la directiva `use namespace` para abrir el espacio de nombres AS3 para todas las definiciones de un bloque de código. Por ejemplo, el código siguiente utiliza la directiva `use namespace` para abrir el espacio de nombres AS3 para los métodos `pop()` y `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 también proporciona opciones de compilador para cada conjunto de propiedades de forma que se pueda aplicar el espacio de nombres AS3 a todo el programa. La opción de compilador `-as3` representa el espacio de nombres AS3 y la opción de compilador `-es` representa la opción de herencia de prototipo (`es` significa ECMAScript). Para abrir el espacio de nombres AS3 para todo el programa, se debe establecer la opción de compilador `-as3` en `true` y la opción de compilador `-es` en `false`. Para utilizar las versiones de prototipos, las opciones de compilador deben establecerse en los valores opuestos. La configuración de compilador predeterminada para Flash Builder y Flash Professional es `-as3 = true` y `-es = false`.

Si se pretende ampliar alguna de las clases principales y sustituir algún método, hay que comprender cómo puede afectar el espacio de nombres AS3 a la manera de declarar un método sustituido. Si se utiliza el espacio de nombres AS3, cualquier método sustituto de un método de clase principal también debe utilizar el espacio de nombres AS3, junto con el atributo `override`. Si no se utiliza el espacio de nombres AS3 y se desea redefinir un método de clase principal en una subclase, no se debe utilizar el espacio de nombres AS3 ni la palabra clave `override`.

Ejemplo: GeometricShapes

La aplicación de ejemplo GeometricShapes muestra cómo se pueden aplicar algunos conceptos y características de la orientación a objetos con ActionScript 3.0:

- Definición de clases
- Ampliación de clases
- Polimorfismo y la palabra clave `override`
- Definición, ampliación e implementación de interfaces

También incluye un "método de fábrica" que crea instancias de clase y muestra la manera de declarar un valor devuelto como una instancia de una interfaz y utilizar ese objeto devuelto de forma genérica.

Para obtener los archivos de la aplicación de este ejemplo, consulte www.adobe.com/go/learn_programmingAS3samples_flash_es. Los archivos de la aplicación GeometricShapes se encuentran en la carpeta Samples/GeometricShapes. La aplicación consta de los siguientes archivos:

Archivo	Descripción
GeometricShapes.mxml o GeometricShapes.fla	El archivo de aplicación principal en Flash (FLA) o Flex (MXML)
com/example/programmingas3/geometricshapes/IGeometricShape.as	Los métodos básicos de definición de interfaz que se van a implementar en todas las clases de la aplicación GeometricShapes.
com/example/programmingas3/geometricshapes/IPolygon.as	Una interfaz que define los métodos que se van a implementar en las clases de la aplicación GeometricShapes que tienen varios lados.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Un tipo de forma geométrica que tiene lados de igual longitud, simétricamente ubicados alrededor del centro de la forma.
com/example/programmingas3/geometricshapes/Circle.as	Un tipo de forma geométrica que define un círculo.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Una subclase de RegularPolygon que define un triángulo con todos los lados de la misma longitud.
com/example/programmingas3/geometricshapes/Square.as	Una subclase de RegularPolygon que define un rectángulo con los cuatro lados de la misma longitud.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Una clase que contiene un método de fábrica para crear formas de un tipo y un tamaño específicos.

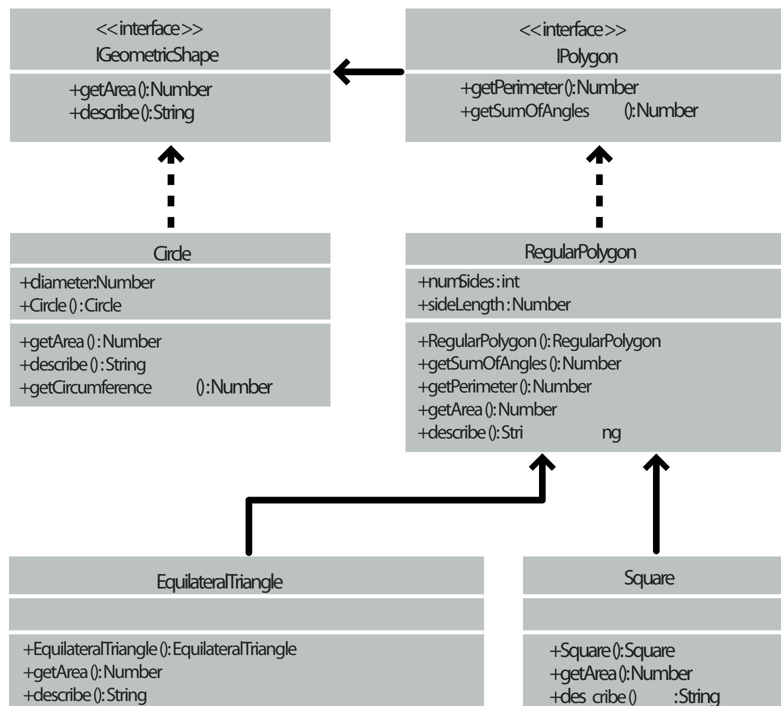
Definición de las clases de GeometricShapes

La aplicación GeometricShapes permite al usuario especificar un tipo de forma geométrica y un tamaño. Responde con una descripción de la forma, su área y su perímetro.

La interfaz de usuario de la aplicación es trivial: incluye algunos controles para seleccionar el tipo de forma, establecer el tamaño y mostrar la descripción. La parte más interesante de esta aplicación está bajo la superficie, en la estructura de las clases y las interfaces.

Esta aplicación manipula formas geométricas, pero no las muestra gráficamente.

Las clases e interfaces que definen las formas geométricas en este ejemplo se muestran en el diagrama siguiente con notación UML (Unified Modeling Language):



Clases de ejemplo GeometricShapes

Definición del comportamiento común en una interfaz

La aplicación GeometricShapes trata con tres tipos de formas: círculos, cuadrados y triángulos equiláteros. La estructura de la clase GeometricShapes empieza por una interfaz muy sencilla, IGeometricShape, que muestra métodos comunes a los tres tipos de formas:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

La interfaz define dos métodos: `getArea()`, que calcula y devuelve el área de la forma y `describe()`, que crea una descripción textual de las propiedades de la forma.

También se pretende obtener el perímetro de cada forma. Sin embargo, el perímetro de un círculo es su circunferencia y se calcula de una forma única, por lo que en el caso del círculo el comportamiento es distinto del de un triángulo o un cuadrado. De todos modos, los triángulos, los cuadrados y otros polígonos son suficientemente similares, así que tiene sentido definir una nueva clase de interfaz para ellos: IPolygon. La interfaz IPolygon también es bastante sencilla, como se muestra a continuación:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Esta interfaz define dos métodos comunes para todos los polígonos: el método `getPerimeter()` que mide la distancia combinada de todos los lados y el método `getSumOfAngles()` que suma todos los ángulos interiores.

La interfaz IPolygon amplía la interfaz IGeometricShape, lo que significa que cualquier clase que implemente la interfaz IPolygon debe declarar los cuatro métodos: dos de la interfaz IGeometricShape y dos de la interfaz IPolygon.

Definición de las clases de formas

Cuando ya se conozcan los métodos comunes a cada tipo de forma, se pueden definir las clases de las formas. Por la cantidad de métodos que hay que implementar, la forma más sencilla es la clase Circle, que se muestra a continuación:

```
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 clase Circle implementa la interfaz IGeometricShape, por lo que hay que proporcionar código para el método `getArea()` y el método `describe()`. Además, define el método `getCircumference()`, que es único para la clase Circle. La clase Circle también declara una propiedad, `diameter`, que no se encuentra en las clases de los otros polígonos.

Los otros dos tipos de formas, cuadrados y triángulos equiláteros, tienen algunos aspectos en común: cada uno de ellos tiene lados de longitud simular y existen fórmulas comunes que se pueden utilizar para calcular el perímetro y la suma de los ángulos interiores para ambos. De hecho, esas fórmulas comunes se aplicarán a cualquier otro polígono regular que haya que definir en el futuro.

La clase RegularPolygon será la superclase para la clase Square y la clase EquilateralTriangle. Una superclase permite centralizar la definición de métodos comunes de forma que no sea necesario definirlos por separado en cada subclase. A continuación se muestra el código para la clase 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;
        }
    }
}
```

En primer lugar, la clase RegularPolygon declara dos propiedades que son comunes a todos los polígonos regulares: la longitud de cada lado (propiedad `sideLength`) y el número de lados (propiedad `numSides`).

La clase RegularPolygon implementa la interfaz IPolygon y declara los cuatro métodos de la interfaz IPolygon. Implementa dos de ellos, `getPerimeter()` y `getSumOfAngles()`, utilizando fórmulas comunes.

Como la fórmula para el método `getArea()` varía de una forma a otra, la versión de la clase base del método no puede incluir lógica común que pueda ser heredada por los métodos de la subclase. Sólo devuelve el valor predeterminado 0, para indicar que no se ha calculado el área. Para calcular el área de cada forma correctamente, las subclases de la clase RegularPolygon tendrán que sustituir el método `getArea()`.

El código siguiente para la clase EquilateralTriangle muestra cómo se sustituye el método `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 palabra clave `override` indica que el método `EquilateralTriangle.getArea()` sustituye de forma intencionada el método `getArea()` de la superclase RegularPolygon. Cuando se llama al método `EquilateralTriangle.getArea()`, se calcula el área con la fórmula del fragmento de código anterior y no se ejecuta el código del método `RegularPolygon.getArea()`.

En cambio, la clase EquilateralTriangle no define su propia versión del método `getPerimeter()`. Cuando se llama al método `EquilateralTriangle.getPerimeter()`, la llamada sube por la cadena de herencia y ejecuta el código del método `getPerimeter()` de la superclase RegularPolygon.

El constructor de `EquilateralTriangle()` utiliza la sentencia `super()` para invocar explícitamente el constructor de `RegularPolygon()` de su superclase. Si ambos constructores tuvieran el mismo conjunto de parámetros, se podría omitir el constructor de `EquilateralTriangle()` y se ejecutaría en su lugar el constructor de `RegularPolygon()`. No obstante, el constructor de `RegularPolygon()` requiere un parámetro adicional, `numSides`. Así, el constructor de `EquilateralTriangle()` llama a `super(len, )`, que pasa el parámetro de entrada `len` y el valor 3 para indicar que el triángulo tendrá tres lados.

El método `describe()` también utiliza la sentencia `super()`, pero de un modo diferente. La utiliza para invocar a la versión de la superclase RegularPolygon del método`describe()`. El método `EquilateralTriangle.describe()` establece primero la variable de cadena `desc` en una declaración del tipo de forma. A continuación, obtiene los resultados del método `RegularPolygon.describe()` llamando a `super.describe()` y añade el resultado a la cadena `desc`.

No se proporciona en esta sección una descripción detallada de la clase Square, pero es similar a la clase EquilateralTriangle; proporciona un constructor y su propia implementación de los métodos `getArea()` y `describe()`.

Polimorfismo y el método de fábrica

Un conjunto de clases que haga buen uso de las interfaces y la herencia se puede utilizar de muchas maneras interesantes. Por ejemplo, todas las clases de formas descritas hasta ahora implementan la interfaz IGeometricShape o amplían una superclase que lo hace. Así, si se define una variable como una instancia de IGeometricShape, no hay que saber si es realmente una instancia de la clase Circle o de la clase Square para llamar a su método `describe()`.

El código siguiente muestra cómo funciona esto:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Cuando se llama a `myShape.describe()`, se ejecuta el método `Circle.describe()` ya que, aunque la variable se define como una instancia de la interfaz IGeometricShape, Circle es su clase subyacente.

En este ejemplo se muestra el principio del polimorfismo en activo: la misma llamada al método tiene como resultado la ejecución de código diferente, dependiendo de la clase del objeto cuyo método se esté invocando.

La aplicación GeometricShapes aplica este tipo de polimorfismo basado en interfaz con una versión simplificada de un patrón de diseño denominado método de fábrica. El término *método de fábrica* hace referencia a una función que devuelve un objeto cuyo tipo de datos o contenido subyacente puede variar en función del contexto.

La clase GeometricShapeFactory mostrada define un método de fábrica denominado `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();
        }
    }
}
```

El método de fábrica `createShape()` permite a los constructores de subclases de formas definir los detalles de las instancias que crean, devolviendo los objetos nuevos como instancias de IGeometricShape de forma que puedan ser manipulados por la aplicación de una manera más general.

El método `describeShape()` del ejemplo anterior muestra cómo se puede utilizar el método de fábrica en una aplicación para obtener una referencia genérica a un objeto más específico. La aplicación puede obtener la descripción para un objeto Circle recién creado así:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

A continuación, el método `describeShape()` llama al método de fábrica `createShape()` con los mismos parámetros y almacena el nuevo objeto Circle en una variable estática denominada `currentShape`, a la que se le asignó el tipo de un objeto IGeometricShape. A continuación, se llama al método `describe()` en el objeto `currentShape` y se resuelve esa llamada automáticamente para ejecutar el método `Circle.describe()`, lo que devuelve una descripción detallada del círculo.

Mejora de la aplicación de ejemplo

La verdadera eficacia de las interfaces y la herencia se aprecia al ampliar o cambiar la aplicación.

Por ejemplo, se puede añadir una nueva forma (un pentágono) a esta aplicación de ejemplo. Para ello se crea una clase Pentagon que amplía la clase RegularPolygon y define sus propias versiones de los métodos `getArea()` y `describe()`. A continuación, se añade una nueva opción Pentagon al cuadro combinado de la interfaz de usuario de la aplicación. Y ya está. La clase Pentagon recibirá automáticamente la funcionalidad de los métodos `getPerimeter()` y `getSumOfAngles()` de la clase RegularPolygon por herencia. Como hereda de una clase que implementa la interfaz IGeometricShape, una instancia de Pentagon puede tratarse también como una instancia de IGeometricShape. Esto significa que para agregar un nuevo tipo de forma, no es necesario cambiar la firma del método de ningún método en la clase GeometricShapeFactory (y por lo tanto, tampoco no es necesario modificar ninguna parte de código que utilice la clase GeometricShapeFactory).

Se puede añadir una clase Pentagon al ejemplo Geometric Shapes como ejercicio, para ver cómo facilitan las interfaces y la herencia el trabajo de añadir nuevas características a una aplicación.