Aprendendo o ACTIONSCRIPT® 3.0

Avisos legais

Para ver os avisos legais, consulte http://help.adobe.com/pt_BR/legalnotices/index.html.

Conteúdo

Capítulo 1: Introdução ao ActionScript 3.0

Sobre o ActionScript

O ActionScript é a linguagem de programação dos ambientes de tempo de execução Adobe® Flash® Player e Adobe® AIR™. Ele permite interatividade, manipulação de dados e muito mais no conteúdo e nos aplicativos do Flash, Flex e AIR.

O ActionScript é executado com a AVM (ActionScript Virtual Machine), que faz parte do Flash Player e do AIR. O código do ActionScript é geralmente transformado no formato de código do bytes por um compilador. (O *código de bytes* é um tipo de linguagem de programação que é escrita e compreendida pelos computadores.) Entre os exemplos de compiladores estão aquele incorporado ao ® Flash® Professional e o que está incorporado no Adobe® Flash® Builder™, disponível no Adobe® Flex™ SDK. O código de bytes está incorporado nos arquivos SWF, que o Flash Player e o AIR executam.

O ActionScript 3.0 oferece um modelo de programação robusto que parece familiar aos desenvolvedores com um conhecimento básico de programação orientada a objetos. Alguns dos recursos principais do ActionScript 3.0 que foram aprimorados em relação à versão anterior incluem:

- Uma nova ActionScript Virtual Machine, chamada AVM2, que usa um novo conjunto de instruções de código de bytes e fornece aprimoramentos de desempenho significativos
- Uma base de código de compilador moderna que executa otimizações mais avançadas do que as versões anteriores do compilador
- Uma API (Interface de programação de aplicativo) expandida e aprimorada, com controle de baixo nível de objetos e um autêntico modelo orientado a objetos
- Uma API XML baseada na especificação de linguagem ECMAScript para XML (E4X) (ECMA-357 edição 2) E4X é a extensão de linguagem para ECMAScript que adiciona XML como um tipo de dados nativo da linguagem.
- Um modelo de evento baseado na Especificação de eventos DOM (Document Object Model) nível 3

Vantagens do ActionScript 3.0

O ActionScript 3.0 vai além dos recursos de script de suas versões anteriores. Ele foi criado para facilitar a criação de aplicativos altamente complexos com grandes conjuntos de dados e bases de código reutilizáveis orientadas a objetos. O ActionScript 3.0 não é necessário para o conteúdo que é executado no Adobe Flash Player. No entanto, ele abre a porta para os aprimoramentos de desempenho que só estão disponíveis com o AVM2 (a máquina virtual do ActionScript 3.0). O código do ActionScript 3.0 pode ser executado até 10 vezes mais rápido do que o código do ActionScript existente.

A versão anterior da AVM1 (ActionScript Virtual Machine) executa os códigos ActionScript 1.0 e ActionScript 2.0. O Flash Player 9 e 10 suportam a AVM1 em termos de compatibilidade reversa.

Novidades do ActionScript 3.0

O ActionScript 3.0 contém muitas classes e recursos que são semelhantes ao ActionScript 1.0 e 2.0. No entanto, o ActionScript 3.0 é diferente, em termos de arquitetura e conceitos, das versões anteriores do ActionScript. Os aprimoramentos do ActionScript 3.0 incluem novos recursos da linguagem central e uma API melhorada que fornece mais controle sobre objetos de baixo nível.

Recursos da linguagem central

A linguagem central define os blocos de construção básicos da linguagem de programação, como instruções, expressões, condições, loops e tipos. O ActionScript 3.0 contém muitos recursos que aceleram o processo de desenvolvimento.

Exceções de tempo de execução

O ActionScript 3.0 relata mais condições de erros que suas versões anteriores. As exceções de tempo de execução são usadas para condições de erro comuns, melhorar a experiência de depuração e permitir o desenvolvimento de aplicativos que manipulam erros de forma robusta. Os erros de tempo de execução fornecem rastreamentos de pilha anotados com informações sobre o arquivo de origem e o número de linha, ajudando a detectar os erros rapidamente.

Tipos de tempo de execução

No ActionScript 3.0, as informações de tipo são preservadas no tempo de execução. Essas informações são usadas para realizar a verificação de tipo em tempo de execução, melhorando a segurança de tipo do sistema. As informações sobre tipo também são usadas para retratar variáveis em representações, que melhoram o desempenho e reduzem o uso de memória. Por meio de comparação, no ActionScript 2.0, as anotações de tipo são principalmente uma ajuda ao desenvolvedor e todos os valores são dinamicamente digitados no tempo de execução.

Classes seladas

O ActionScript 3.0 inclui o conceito de classes seladas. Uma classe selada possui apenas o conjunto fixo de propriedades e métodos que são definidos em tempo de compilação; não é possível adicionar mais propriedades e métodos. A incapacidade de alterar uma classe em tempo de execução permite uma verificação mais estrita em tempo de compilação, resultando em programas mais robutos. Ela também melhora o uso de memória por não exigir uma tabela de hash interna para cada ocorrência de objeto. As classes dinâmicas também são possíveis usando a palavra-chave `dynamic`. Todas as classes no ActionScript 3.0 são seladas por padrão, mas podem ser declaradas para se tornar dinâmicas com a palavra-chave `dynamic`.

Fechamentos de método

O ActionScript 3.0 permite um fechamento de método que lembra automaticamente de sua ocorrência de objeto original. Esse recurso é útil para a manipulação de eventos. No ActionScript 2.0, os fechamentos de método não lembram de qual instância de objeto tinham sido extraídos, gerando comportamentos inesperados quando o fechamento de método é invocado.

ECMAScript para XML (E4X)

O ActionScript 3.0 implementa o ECMAScript para XML (E4X), recentemente padronizado como ECMA-357. O E4X oferece um conjunto fluente de construções de linguagem para manipular XML. Diferentemente das APIs tradicionais de análise de XML, o XML com E4X funciona como um tipo de dados nativo da linguagem. O E4X simplifica o desenvolvimento de aplicativos que manipulam XML, reduzindo drasticamente a quantidade de código necessária.

Para exibir a especificação E4X do ECMA, vá para www.ecma-international.org.

Expressões regulares

O ActionScript 3.0 inclui suporte nativo para expressões regulares, o que permite pesquisar e manipular seqüências de caracteres rapidamente. O Action Script 3.0 implementa o suporte a expressões regulares conforme definidas na especificação de linguagem ECMAScript (ECMA-262) edição 3.

Espaços para nomes

Os espaços para nomes são semelhantes aos especificadores de acesso tradicionais usados para controlar a visibilidade de declarações (`public`, `private`, `protected`). Eles funcionam como especificadores de acesso personalizados, que podem ter os nomes que você escolher. Os espaços para nomes são equipados com um URI (Identificador Universal de Recursos) para evitar colisões e também são usados para representar nomes para espaços XML no trabalho com E4X.

Novos tipos primitivos

O ActionScript 3.0 contém três tipos numéricos: Number, int e uint. Number representa uma número de precisão dupla com ponto de oscilação. O tipo int é um inteiro assinado de 32 bits que permite ao código ActionScript aproveitar os rápidos recursos matemáticos de inteiros da CPU. O tipo int é útil para contadores de loop e variáveis em que os inteiros são usados. O tipo uint é um tipo inteiro de 32 bits não assinado, útil para valores de cores RGB, contagens de bytes e muito mais. Por outro lado, o ActionScript 2.0 só tem um tipo numérico, Number.

Recursos da API

As APIs no ActionScript 3.0 contêm várias classes que permitem controlar objetos em um nível baixo. A arquitetura da linguagem foi projetada para ser mais intuitiva do que a das outras versões. Embora haja muitas classes para analisar em detalhes, vale a pena observar algumas diferenças significativas.

Modelo de eventos DOM3

O modelo de eventos Document Object Model Nível 3 (DOM3) fornece uma forma padronizada de gerar e tratar as mensagens de evento. Este modelo de evento foi projetado para permitir que os objetos nos aplicativos interajam e se comuniquem, mantenham o estado e respondam à alteração. O modelo de evento do ActionScript 3.0 é padronizado depois da Especificação de Eventos para DOM Nível 3 do World Wide Web Consortium. Este modelo fornece um mecanismo mais claro e mais eficiente do que os sistemas de evento disponíveis nas versões anteriores do ActionScript.

Os eventos e eventos de erros estão localizados no pacote flash.events. Os componentes do Flash Professional e a estrutura do Flex usam o moesmo modelo de eventos, por isso o sistema de eventos é unificado na Plataforma Flash.

API de lista de exibição

A API para acessar a lista de exibição — a árvore que contém os elementos visuais no aplicativo — consiste em classes para trabalhar com primitivas visuais.

A classe Sprite é um bloco estrutural leve, projetado para ser uma classe base para os elementos visuais, como os componentes da interface de usuário. A classe Shape representa formas de vetor brutas. Essas classes podem ser instanciadas naturalmente com o operador `new` e atribuídas a um pai dinamicamente a qualquer momento.

O gerenciamento de profundidade é automático. Os métodos foram fornecidos para especificar e gerenciar a ordem de empilhamento dos objetos.

Manipulação de dados e conteúdo dinâmicos

O ActionScript 3.0 contém mecanismos para carregar e manipular ativos e dados no seu aplicativo que são intuitivos e consistentes na API. A classe Loader fornece um único mecanismo para carregar arquivos SWF e ativos de imagem e fornece uma forma de acessar informações detalhadas sobre o conteúdo carregado. A classe URLLoader fornece um mecanismo separado para carregar texto e dados binários em aplicativos orientados a dados. A classe Socket fornece um meio de ler e gravar dados binários nos soquetes do servidor em qualquer formato.

Acesso a dados de baixo nível

Várias APIs fornecem acesso de baixo nível aos dados. Para os dados obtidos por download, a classe URLStream fornece acesso a dados como dados binários brutos enquanto estão sendo baixados. A classe ByteArray permite otimizar a leitura, a gravação e o trabalho com dados binários. A API Sound fornece controle detalhado de som por meio das classes SoundChannel e SoundMixer. As APIs de segurança fornecem informações sobre os privilégios de segurança de um arquivo SWF ou conteúdo carregado, permitindo manipular os erros de segurança.

Trabalho com texto

O ActionScript 3.0 contém um pacote flash.text para todas as APIs relacionadas a texto. A classe TextLineMetrics fornece uma métrica detalhada para uma linha de texto em um campo de texto; ela substitui o método `TextFormat.getTextExtent()` no ActionScript 2.0. A classe TextField contém vários métodos de baixo nível que fornecem informações específicas sobre uma linha de texto ou sobre um único caractere em um campo de texto. Por exemplo, o método `getCharBoundaries()` retorna um retângulo que representa a caixa delimitadora de um caractere. O método `getCharIndexAtPoint()` retorna o índice do caractere em um ponto especificado. O método `getFirstCharInParagraph()` retorna o índice do primeiro caractere em um parágrafo. Os métodos de nível de linha incluem `getLineLength()`, que retorna o número de caracteres em uma linha de texto especificada, e `getLineText()`, que retorna o texto da linha especificada. A classe Font fornece um meio de gerenciar fontes incorporadas em arquivos SWF.

Até para o controle de nível inferior sobre o texto, as classes no pacote flash.text.engine compõem o Mecanismo de Texto do Flash. Este conjunto de classes fornece um controle de baixo nível sobre o texto e é projetado para criar estruturas e componentes de texto.

Capítulo 2: Introdução do ActionScript

Fundamentos de programação

Como o ActionScript é uma linguagem de programação, para conhecê-lo, primeiro será necessário compreender alguns conceitos gerais de programação de computador.

O que os programas de computador fazem

Em primeiro lugar, é bom saber o que é um programa de computador e o que ele faz. Um programa de computador consiste em dois aspectos principais:

- Ele é uma série de instruções ou etapas que o computador deve executar.
- Cada etapa envolve a manipulação de algumas informações ou dados.

Em termos gerais, um programa de computador é apenas uma série de comandos passo a passo que você fornece ao computador e ele executa. Cada comando é conhecido como *instrução*. No ActionScript, cada instrução é escrita com um ponto-e-vírgula no final.

Em essência, tudo o que uma determinada instrução faz em um programa é manipular alguns dados que estão armazenados na memória do computador. Um exemplo simples é instruir o computador a acrescentar dois números e armazenar o resultado na memória. Um exemplo mais complexo é se houver o desenho de um retângulo em um lugar na tela e você quer escrever um programa a fim de movê-lo para outro lugar. O computador se lembra de determinadas informações sobre o retângulo: as coordenadas x, y nas quais ele está localizado, sua largura, altura, cor, e assim por diante. Cada uma dessas informações está armazenada em um local na memória do computador. Um programa para mover o retângulo para um local diferente teria etapas como "trocar a coordenada x para 200; trocar a coordenada y para 150". Em outras palavras, ele especificaria novos valores para as coordenadas x e y. Em segundo plano, o computador faz alguma coisa com os dados para, na verdade, transformar esses números na imagem que aparece na tela do computador. No entanto, no nível básico de detalhamento, é suficiente saber que o processo de "mover um retângulo na tela" só envolve alterar os bits de dados na memória do computador.

Variáveis e constantes

A programação envolve principalmente trocar as informações na memória do computador. Consequentemente, é importante ter uma forma de representar uma informação em um programa. Uma *variável* é um nome que representa um valor na memória do computador. Durante a escrita de instruções para manipular valores, o nome da variável é escrito no lugar do valor. Sempre que se deparar com o nome da variável no seu programa, o computador consultará a memória e usará o valor que encontrar nela. Por exemplo, se você tiver duas variáveis chamadas `value1` e `value2`, cada uma contendo um número, para adicionar esses dois números, você pode escrever a instrução:

```
value1 + value2
```

Quando executar as etapas, o computador verifica os valores de cada variável e os adiciona juntos.

No ActionScript 3.0, uma variável consiste em três partes diferentes:

- O nome da variável
- O tipo de dados que pode ser armazenado nela
- O valor real armazenado na memória do computador

Você viu como o computador usa o nome como alocador de espaço para o valor. O tipo de dados também é importante. Ao criar uma variável no ActionScript, você especifica o tipo específico de dados que devem ser mantidos. Deste ponto em diante, as instruções do seu programa podem armazenar apenas aquele tipo de dado na variável. Você pode manipular o valor usando as características em particular associadas com seu tipo de dados. No ActionScript, a criação de uma variável (conhecida como *declarar* a variável) requer o uso da instrução `var`:

```
var value1:Number;
```

Esse exemplo diz para o computador criar uma variável chamada `value1`, que pode armazenar apenas dados de Number. ("Number" é um tipo de dados específico definido no ActionScript.) Você também pode armazenar um valor na variável imediatamente:

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

Adobe Flash Professional
No Flash Professional, existe outro meio de declarar uma variável. Durante a colocação de um símbolo de clipe de filme, símbolo de botão ou campo de texto no Palco, você pode lhe dar um nome de ocorrência no Inspetor de propriedades. Em segundo plano, o Flash Professional cria uma variável com o mesmo nome que o nome da instância. Você pode usar esse nome no seu código ActionScript para representar esse item de palco. Suponhamos, por exemplo, que você tenha um símbolo de clipe de filme no Palco e dê a ele o nome da instância `rocketShip`. Sempre que usar a variável `rocketShip` no código ActionScript, você, na verdade, está manipulando esse clipe de filme.

Uma *constante* é semelhante a uma varíavel. Trata-se de um nome que representa um valor na memória do computador com o tipo de dado especificado. A diferença é que um valor só pode ser atribuído a uma constante uma única vez no processamento do aplicativo do ActionScript. Assim que é atribuído, o valor da constante é o mesmo em todo o aplicativo. A sintaxe para declarar uma constante é quase igual à sintaxe usada com uma variável. A única diferença é que você usa a palavra-chave `const` no lugar da palavra-chave `var`:

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

Uma constante é útil para definir um valor que é usado em vários locais em um projeto e que não é alterado sob circunstâncias normais. O uso de uma constante em vez de um valor literal torna o código mais legível. Por exemplo, pense em duas versões do mesmo código. Uma multiplica um preço por `SALES_TAX_RATE`. A outra multiplica o preço por `0.07`. A versão que usa a constante `SALES_TAX_RATE` é mais fácil de entender. Além disso, suponhamos que o valor definido pela constante seja alterado. Se você usar uma constante para representar esse valor em todo o projeto, você poderá trocar o valor em um lugar (a declaração constante). Por outro lado, você teria que trocá-lo em vários lugares se tiver usado valores literais hard-coded.

Tipos de dados

No ActionScript, há vários tipos de dados que você pode usar como os tipos de dados da variável que você criar. Alguns desses tipos de dados podem ser entendidos como "simples" ou "fundamentais":

- Seqüência de caracteres: um valor textual, como um nome ou o texto do capítulo de um livro
- Numérico: o ActionScript 3.0 inclui três tipos de dados específicos para dados numéricos:
  - Número: qualquer valor numérico, incluindo valores com ou sem uma fração
  - int: um inteiro (um número inteiro sem uma fração)
  - uint: um inteiro "sem sinal", que significa um número inteiro que não pode ser negativo
- Booleano: um valor do tipo verdadeiro ou falso, tal como se uma opção está ativa ou se dois valores são iguais

Os tipos de dados simples representam uma única informação: por exemplo, um único número ou uma única seqüência de texto. No entanto, a maioria dos tipos de dados definidos no ActionScript são complexos. Eles representam um conjunto de valores em um contêiner simples. Por exemplo, uma variável com o tipo de dados Date representa um valor único (um momento no tempo). No entanto, esse valor de data é representado com diversos valores: dia, mês, ano, horas, minutos, segundos etc., que são números individuais. Em geral, as pessoas pensam em uma data como um valor único, e você pode tratar uma data como um valor único criando uma varíavel Date. No entanto, internamente o computador pensa nele como um grupo de vários valores que, postos juntos, definem uma data única.

A maioria dos tipos de dados embutidos, bem como os definidos pelos programadores, são tipos de dados complexos. Entre os tipos de dados complexos que talvez você conheça estão:

- MovieClip: um símbolo de clipe de filme
- TextField: um campo de texto de entrada ou dinâmico
- SimpleButton: um símbolo de botão
- Date: informações sobre um momento único no tempo (uma data e hora)

Duas palavras que, em geral, são usadas como sinônimos de tipos de dados são classe e objeto. Uma *classe* é simplesmente a definição de um tipo de dado. É como um modelo para todos os objetos do tipo de dados, como dizer "todas as variáveis do tipo de dados Example têm estas características: A, B and C". Um *objeto*, por outro lado, é apenas uma instância real de uma classe. Por exemplo, a variável cujo tipo de dados é MovieClip poderia ser descrita como um objeto MovieClip. Estas são formas diferentes de dizer a mesma coisa:

- O tipo de dados da variável `myVariable` é Number.
- A variável `myVariable` é uma ocorrência de Number.
- A variável `myVariable` é um objeto Number.
- A variável `myVariable` é uma ocorrência da classe Number.

Trabalho com o objetos

O ActionScript é conhecido como uma linguagem de programação orientada a objetos. A programação orientada a objetos é simplesmente uma abordagem a essa programação. Na verdade, não é nada além de uma forma de organizar o código em um programa, usando objetos.

Antes o termo "programa de computador" era definido como uma série de etapas ou instruções que o computador realiza. De forma conceitual, então, você pode imaginar um programa de computador como uma longa lista de instruções. No entanto, na programação orientada a objetos, as instruções do programa são divididas entre objetos diferentes. O código é agrupado em pedaços de funcionalidade; por isso, os tipos relacionados de funcionalidade ou as informações relacionadas são agrupados em um contêiner.

Adobe Flash Professional

Se você já trabalhou com símbolos no Flash Professional, então está acostumado a trabalhar com objetos. Imagine que você definiu um símbolo de clipe de filme, como o desenho de um retângulo, e colocou uma cópia dele no Palco. Esse símbolo de clipe de filme também é (literalmente) um objeto no ActionScript; é uma ocorrência da classe MovieClip.

Há diversas características do clipe de filme que você pode modificar. Quando está selecionado, você pode trocar os valores no inspetor de propriedades, como a coordenada x ou sua largura. Você também pode fazer vários ajustes de cor, como trocar o alfa (transparência) ou aplicar um filtro de sombra projetada. Outras ferramentas do Flash Professional permitem fazer mais alterações, como usar a ferramenta Transformação livre para girar o retângulo. Todas essas formas por meio das quais você pode modificar um símbolo de clipe de filme no Flash Professional também estão disponíveis no ActionScript. Você modifica o clipe de filme no ActionScript alterando os dados que são colocados juntos em um único pacote chamado objeto MovieClip.

Na programação orientada a objetos do ActionScript, há três tipos de características que qualquer classe pode incluir:

- Propriedades
- Métodos
- Eventos

Esses elementos ajudam a gerenciar as partes dos dados usadas pelo programa e a decidir quais ações são executadas em uma determinada ordem.

Propriedades

Uma propriedade representa uma das partes dos dados que são compactados em um objeto. Um exemplo de objeto de música pode ter propriedades chamadas `artist` e `title`; a classe MovieClip tem propriedades como `rotation`, `x`, `width` e `alpha`. Você trabalha com propriedades como variáveis individuais. Na verdade, você pode pensar nas propriedades como simplesmente as variáveis "filho" contidas em um objeto.

A seguir, apresentamos alguns exemplos de código do ActionScript que usa propriedades. Esta linha de código move o MovieClip chamado `square` para 100 pixels na coordenada x:

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

Este código usa a propriedade de rotação fazendo com que o MovieClip `square` gire para corresponder à rotação do MovieClip `triangle`:

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

Este código altera a escala horizontal do MovieClip `square` para que ele fique uma vez e meia maior do que antes:

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

Observe a estrutura comum: você usa uma variável (`square`, `triangle`) como o nome do objeto, seguido de um ponto (`.`) e, depois, o nome da propriedade (`x`, `rotation`, `scaleX`). O ponto, conhecido como *operador ponto*, é usado para indicar que você está acessando um dos elementos filho de um objeto. A estrutura inteira reunida, "nome da variável-ponto-nome da propriedade", é usada como uma única variável, como um nome para um único valor na memória do computador.

Métodos

Um *método* é uma ação que um objeto pode realizar. Por exemplo, suponhamos que você tenha feito um símbolo de clipe de filme no Flash Professional com vários quadros-chave e animação em sua linha de tempo. Esse clipe de filme pode reproduzir, ou parar, ou ser instruído a mover o indicador de reprodução para um quadro em particular.

Este código instrui o MovieClip chamado `shortFilm` a iniciar a reprodução:

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

Esta linha faz o MovieClip chamado `shortFilm` parar a reprodução (o indicador de reprodução pára; é como pausar um vídeo):

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

Este código faz um MovieClip chamado `shortFilm` mover o indicador de reprodução para o Quadro 1 e parar a reprodução (é como retroceder um vídeo):

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

Os métodos, assim como as propriedades, são acessados escrevendo-se o nome do objeto (uma variável), um ponto e o nome do método seguido de parênteses. Os parênteses são uma forma de indicar que você está *chamando* o método ou, em outras palavras, instruindo o objeto a executar aquela ação. Às vezes, os valores (ou as variáveis) são colocados entre parênteses, como uma forma de passar adiante informações adicionais necessárias para executar a ação. Esses valores são conhecidos como *parâmetros* de método. Por exemplo, o método `gotoAndStop()` precisa de informações sobre para qual quadro ir, e por isso requer um único parâmetro entre parênteses. Outros métodos, como `play()` e `stop()`, são auto-explicativos, por isso não requerem informações extras. No entanto, eles ainda são escritos com parênteses.

Diferentemente das propriedades (e variáveis), os métodos não são usados como alocadores de espaço de valor. Entretanto, alguns métodos podem executar cálculos e retornar um resultado que pode ser usado como uma variável. Por exemplo, o método `toString()` da classe Number converte o valor em sua representação de texto:

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

Por exemplo, você usaria o método `toString()` se quisesse exibir o valor de uma variável Number em um campo de texto na tela. A propriedade `text` da classe TextField é definida como um String, por isso pode conter apenas valores de texto. (A `propriedade` do texto representa o conteúdo de texto real exibido na tela.) Essa linha de código converte o valor numérico na variável `numericData` para texto. Então, ela cria o valor apresentado na tela no objeto TextField, chamado `calculatorDisplay`:

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

Eventos

Um programa de computador é uma série de instruções que o computador executa em etapas. Alguns programas de computador simples consistem em nada mais do que algumas etapas que ele executa e depois é encerrado. Entretanto, os programas do ActionScript são criados para continuar a execução e esperar a entrada do usuário ou a ocorrência de outras coisas. Os eventos são mecanismos que determinam quais instruções o computador executa em um determinado momento.

Basicamente, os *eventos* são fenômenos que acontecem, sobre os quais o ActionScript é informado e aos quais pode responder. Muitos eventos estão relacionados com a interação do usuário, como o usuário clicar em um botão ou pressionar uma tecla no teclado. Também há outros tipos de eventos. Por exemplo, se você usar o ActionScript para carregar uma imagem externa, há um evento que poderá avisá-lo quando o carregamento da imagem for concluído. Quando um programa ActionScript está sendo executado, conceitualmente ele apenas "senta" e espera que certas coisas aconteçam. Quando essas coisas acontecem, o código específico do ActionScript que você especificou para esses eventos é executado.

Tratamento de eventos básico

A técnica para especificar determinadas ações que devem ser executadas em resposta a eventos específicos é conhecida como *tratamento de eventos*. Durante a escrita do código do ActionScript para executar o tratamento de eventos, há três elementos importantes a serem identificados:

- A origem do evento: para qual objeto o evento deverá ocorrer? Por exemplo, em qual botão se clicou, ou qual objeto Loader está carregando a imagem? A origem do evento também é conhecida como *destino do evento*. Tem esse nome porque é o objeto no qual o computador focaliza o evento (ou seja, onde o evento realmente acontece).
- O evento: o que vai acontecer, a qual fenômeno você deseja responder? É importante identificar o evento específico, porque muitos objetos acionam diversos eventos.
- A resposta: quais etapas você deseja executar quando o evento acontecer?

Em qualquer momento no qual você grava o código ActionScript para manipular eventos, ele exige esses três elementos. O código segue esta estrutura básica (os elementos em negrito são espaços reservados que você preencheria para o seu caso específico):

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

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

Esse código faz duas coisas. Primeiro, ele define uma função, que é a forma de especificar as ações que você deseja executar em resposta ao evento. Em seguida, ele chama o método `addEventListener()` do objeto de origem. Chamar `addEventListener()` essencialmente "inscreve" a função no evento especificado. Quando o evento acontece, são realizadas as ações da função. Pensemos em cada uma dessas partes com mais detalhes.

Uma *função* fornece um meio de agrupar as ações, com um único nome, que é como um nome de atalho, para executar as ações. Uma função é idêntica a um método, com a diferença de que não está necessariamente associada com uma classe específica. (Na verdade, o termo "método" poderia ser definido como uma função que está associada com uma classe em particular.) Durante a criação de uma função para tratamento de eventos, você escolhe o nome da função (chamada de `eventResponse` neste caso). Você também especifica um parâmetro (chamado `eventObject` neste exemplo). A especificação de um parâmetro de função é como declarar uma variável, por isso você também deve indicar o tipo de dados do parâmetro. (Neste exemplo, o tipo de dados do parâmetro é `EventType`.)

Cada tipo de evento que você deseja escutar é associado a uma classe do ActionScript. O tipo de dados que você define para o parâmetro de função é sempre a classe associada ao evento específico ao qual deseja responder. Por exemplo, um evento `click` (acionado quando o usuário clica em um item) é associado à classe MouseEvent. Para escrever uma função de ouvinte para um evento `click`, você define essa função com um parâmetro com o tipo de dados MouseEvent. Finalmente, entre chaves ({ ... }), você escreve as instruções que deseja que o computador execute quando o evento ocorrer.

A função de manipulação de eventos é escrita. Em seguida, você diz para o objeto de origem de evento (o objeto para o qual o evento acontece, por exemplo, o botão) que você deseja que ele chame sua função quando o evento acontecer. Você registra sua função com o objeto de origem de evento chamando o método `addEventListener()` daquele objeto (todos os objetos que têm eventos também têm um método `addEventListener()`). O método `addEventListener()` usa dois parâmetros:

- O primeiro é o nome do evento específico ao qual você deseja responder. Cada evento está afiliado com uma classe específica. Todas as classes de evento têm um valor especial, que é como um nome exclusivo, definido para cada um de seus eventos. Você usa esse valor para o primeiro parâmetro.

- O segundo é o nome da função de resposta do evento. Observe que um nome de função é escrito sem parênteses quando transmitido como um parâmetro.

Processo de tratamento de eventos

Veja a seguir uma descrição passo a passo do processo que acontece durante a criação de um ouvinte de eventos. Neste caso, é um exemplo de criação de uma função de ouvinte que é chamada quando um objeto `myButton` é clicado.

O código real escrito pelo programador é o seguinte:

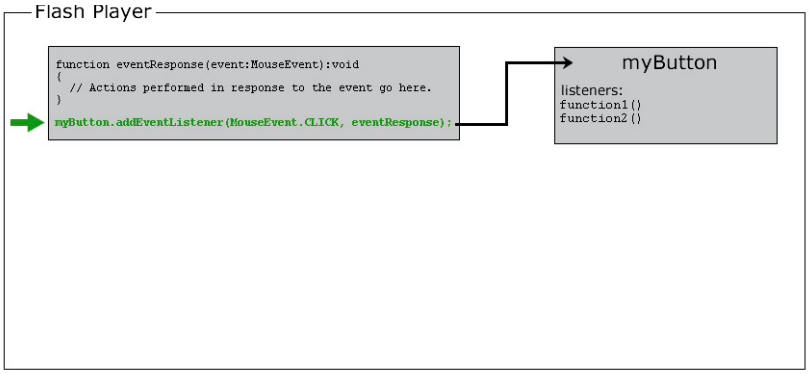
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

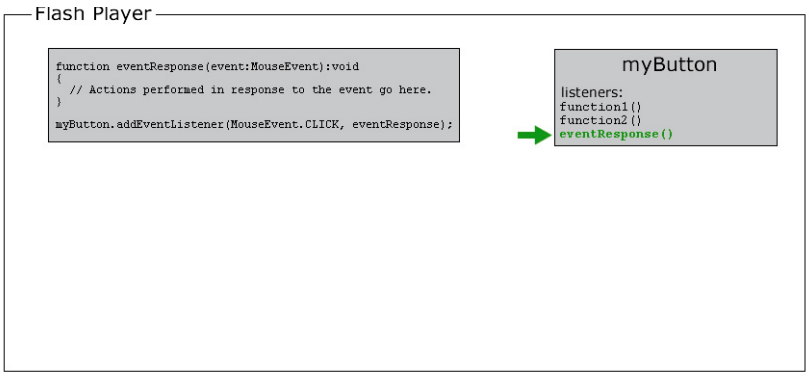
Apresentaremos a seguir o funcionamento desse código, quando executado:

1 Quando o arquivo SWF é carregado, o computador registra o fato de que há uma função chamada `eventResponse()`.

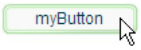
2 Em seguida, o computador executa o código (especificamente, as linhas de código que não estão contidas em uma função). Neste caso, é apenas uma linha de código: chamar o método `addEventListener()` no objeto de origem do evento (chamado `myButton`) e transmitir a função `eventResponse` como um parâmetro.



Internamente, `myButton` mantém uma lista de funções que estão ouvindo cada um de seus eventos. Quando o método `addEventListener()` é chamado, `myButton` armazena a função `eventResponse()` em sua lista de ouvintes de evento.

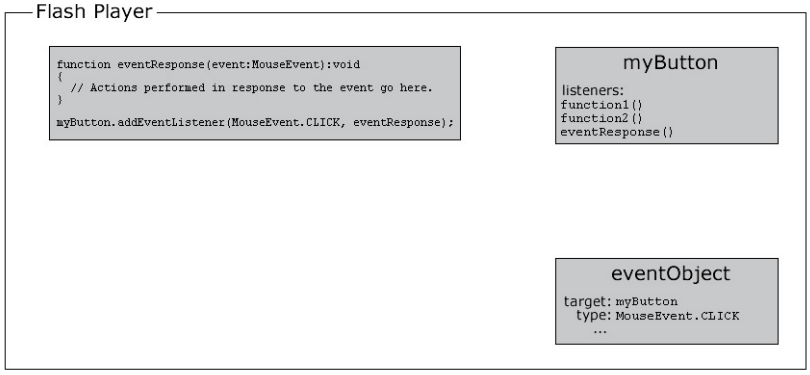


3 Em algum momento, o usuário clica no objeto `myButton`, acionando o evento `click` (identificado como `MouseEvent.CLICK` no código).

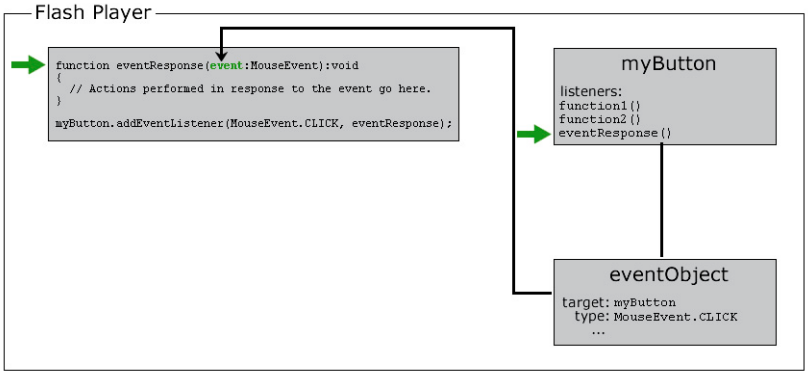


Então, acontece o seguinte:

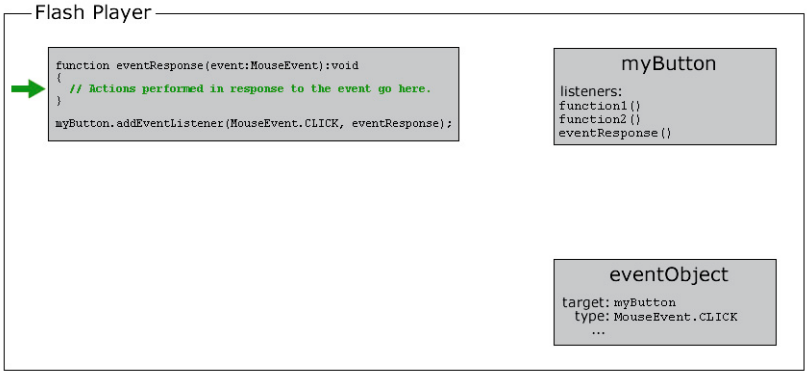
a É criado um objeto que é uma ocorrência da classe associada com o evento em questão (MouseEvent neste exemplo). Para muitos eventos, esse objeto é uma ocorrência da classe Event. Para os eventos de mouse, é uma ocorrência MouseEvent. Para outros eventos, é uma ocorrência da classe que está associada com esse evento. Esse objeto criado é conhecido como o *objeto de evento* e contém informações específicas sobre o evento que ocorreu: qual é o tipo de evento, onde ele ocorreu e outras informações específicas de evento, se aplicável.



b Em seguida, o computador verifica a lista de ouvintes de eventos armazenada por `myButton`. Ele verifica todas essas funções, chamando uma de cada vez e transmitindo o objeto de evento para a função como um parâmetro. Como a função `eventResponse()` é um dos ouvintes de `myButton`, como parte desse processo, o computador chama a função `eventResponse()`.



c Quando a função `eventResponse()` é chamada, o código nela é executado e as ações especificadas são realizadas.



Exemplos de tratamento de eventos

A seguir há outros exemplos concretos do código de manipulação de evento. Esses exemplos servem para lhe dar uma ideia de alguns elementos de evento comuns e as possíveis variações disponíveis quando você grava o código de manipulação de evento:

- Clicar em um botão para iniciar a reprodução do clipe de filme atual. No exemplo a seguir, `playButton` é o nome da ocorrência do botão, e `this` é um nome especial que significa "o objeto atual":

```
this.stop();

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

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

- Detectar tipos em um campo de texto: Neste exemplo, `entryText` é um campo de texto de entrada, e `outputText` é um campo de texto dinâmico:

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

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

- Clicar em um botão para navegar em uma URL. Nesse caso, `linkButton` é o nome de ocorrência do botão:

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

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

Criar ocorrências de objetos

Para que você possa usar um objeto no ActionScript, primeiro, ele deve existir. Uma parte da criação de um objeto é declarar uma variável; porém, a declaração de uma variável só cria um lugar vazio na memória do computador. Sempre atribua um valor real à variável (criar um objeto e armazená-lo na variável), antes de tentar usá-lo ou manipulá-lo. O processo de criar um objeto é conhecido como *instanciar* o objeto. Em outras palavras, você cria uma ocorrência de uma classe em particular.

Existe uma maneira simples de criar uma ocorrência de objeto que não envolve o ActionScript. No Flash Professional, coloque um símbolo de clipe de filme, símbolo de botão ou campo de texto no Palco e atribua-o a um nome de instância. O Flash Professional automaticamente declara uma variável com esse nome de instância, cria uma instância de objeto e armazena esse objeto na variável. De forma semelhante, no Flex você cria um componente no MXML ou codificando uma tag MXML ou colocando o componente no editor, no modo Flash Builder Design. Ao atribuir um ID a esse componente, este ID se torna o nome de uma variável ActionScript que contém essa ocorrência do componente.

No entanto, nem sempre você deseja criar um objeto visualmente, e para os objetos não-visuais você não pode. Há várias formas adicionais através das quais você pode criar instâncias de objeto usando apenas o ActionScript.

Com vários tipos de dados do ActionScript, você pode criar uma ocorrência usando uma *expressão literal*, que é um valor escrito diretamente no código do ActionScript. Eis alguns exemplos:

- Valor numérico literal (insere o número diretamente):

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

- Valor da seqüência de caracteres literal (envolve o texto com aspas duplas):

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

- Valor booleano literal (usa os valores literais `true` ou `false`):

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

- Valor Array literal (envolve uma lista de valores separados por vírgula entre colchetes):

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

- Valor XML literal (insere o XML diretamente):

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

O ActionScript também define expressões literais para os tipos de dados Array, RegExp, Object e Function.

A forma mais comum de criar uma ocorrência para qualquer tipo de dados é usar o operador `new` com o nome da classe, como mostrado aqui:

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

O ato de criar de um objeto usando o operador `new`, muitas vezes, é descrito como "chamar o construtor da classe" Um *construtor* é um método especial que é chamado como parte do processo de criar uma ocorrência de uma classe. Observe que, ao criar uma ocorrência nesta forma, você coloca parênteses depois do nome da classe. Às vezes, você especifica os valores do parâmetro nos parênteses. Há duas coisas que você pode fazer ao chamar um método.

Mesmo para esses tipos de dados que permitem criar ocorrências usando uma expressão literal, você também pode usar o operador `new` para criar uma ocorrência de objeto. Por exemplo, as duas linhas de código a seguir fazem a mesma coisa:

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

É importante se familiarizar com a forma como o `novo` *ClassName*`()` cria objetos. Muitos tipos de dados ActionScript não têm uma representação visual. Consequentemente, não podem ser criados colocando-se um item no Palco do Flash Professional ou no modo Design do editor MXML do Flash Builder. Você só pode criar uma ocorrência de qualquer um desses tipos de dados no ActionScript usando o operador `new`.

Adobe Flash Professional
No Flash Professional, o operador `new` também pode ser usado para criar uma ocorrência de um símbolo de clipe de filme que é definido na Biblioteca mas não é colocado no Palco.

Mais tópicos da Ajuda

Trabalho com matrizes

Uso de expressões regulares

Criação de objetos MovieClip com o ActionScript

Elementos de programa comuns

Há alguns blocos estruturais extras que você pode usar para criar um programa ActionScript.

Operadores

Os *operadores* são símbolos (ou, ocasionalmente, palavras) essenciais que são usados para executar cálculos. Eles são muito usados em operações matemáticas e também para comparar valores. Em geral, um operador usa um ou mais valores e "obtém" um único resultado. Por exemplo:

- O operador de adição (`+`) adiciona dois valores, tendo como resultado um único número:

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

- O operador de multiplicação (`*`) multiplica dois valores, tendo como resultado um único número:

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

- O operador de igualdade (`==`) compara se dois valores são iguais, tendo como resultado um único valor verdadeiro ou falso (booleano):

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

  Conforme mostrado aqui, o operador de igualdade e os outros operadores de "comparação" são usados, em geral, com a instrução `if` para determinar se certas instruções devem ou não ser executadas.

Comentários

À medida que estiver escrevendo ActionScript, você vai querer, em geral, deixar notas para você mesmo. Por exemplo, às vezes você quer explicar como certas linhas de código funcionam, ou por que você fez uma escolha em particular. Os *comentários de código* são uma ferramenta que você pode usar para escrever um texto que o computador ignora no seu código. O ActionScript inclui dois tipos de comentários:

- Comentário de uma linha: um comentário de uma linha é criado colocando duas barras em qualquer lugar de uma linha. O computador ignora tudo depois das barras invertidas, até o final daquela linha:

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

- Comentário de várias linhas: um comentário de várias linhas inclui um marcador de comentário inicial (`/*`), o conteúdo do comentário e um marcador de comentário final (`*/`). O computador ignora tudo entre os marcadores de início e fim, independentemente de quantas linhas o comentário abrange:

  ```
  /*
  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.
  */
  ```

Outro uso comum dos comentários é "desligar" temporariamente uma ou mais linhas do código. Por exemplo, você pode usar os comentários se estiver testando uma forma diferente de fazer alguma coisa. Também use-os para tentar descobrir por que alguns códigos ActionScript não estão funcionando da forma esperada.

Controle do fluxo

Muitas vezes em um programa, você deseja repetir determinadas ações, executar apenas algumas e outras não, executar ações conforme condições específicas etc. O *controle de fluxo* é o controle sobre as ações que são executadas. Há vários tipos de elementos de controle de fluxo disponíveis no ActionScript.

- Funções: as funções são como atalhos. Fornecem um meio de agrupar uma série de ações sob um único nome e podem ser usadas para realizar cálculos. As funções são necessárias para tratar eventos, mas também são usadas como ferramenta geral para agrupar uma série de instruções.
- Loops: as estruturas de loop permitem designar um conjunto de instruções que o computador executa por um determinado número de vezes ou até que alguma condição seja alterada. Com freqüência, os loops são usados para manipular vários itens relacionados, empregando uma variável cujo valor é alterado sempre que o computador completa o loop.
- Instruções condicionais: as instruções condicionais fornecem uma forma de designar certas instruções que são realizadas apenas sob determinadas circunstâncias. Também são usadas para fornecer conjuntos alternativos de instruções para condições diferentes. O tipo mais comum de instrução condicional é a instrução `if`. A instrução `if` verifica um valor ou uma expressão entre parênteses. Se o valor for `true`, as linhas de código entre chaves são realizadas. Do contrário, são ignoradas. Por exemplo:

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

  A parceira da instrução `if`, a instrução `else`, permite designar instruções alternativas que o computador realiza se a condição não for `true`:

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

Exemplo: parte de portfólio de animação (Flash Professional)

Este exemplo foi criado para lhe oferecer uma oportunidade de ver pela primeira vez como é possível juntar partes do ActionScript para obter um aplicativo completo. A parte de portfólio de animação é um exemplo de como você poderia pegar uma animação linear existente e acrescentar alguns elementos interativos menores. Por exemplo, você poderia incorporar uma animação criada para um cliente em um portfólio on-line. O comportamento interativo que você vai adicionar à animação inclui dois botões nos quais o espectador poderá clicar: um para iniciar a animação e outro para navegar em uma URL separada (como o menu do portfólio ou a home page do autor).

O processo de criar essa peça pode ser dividido nestas seções principais:

1. Preparar o arquivo FLA para adicionar elementos interativos e do ActionScript.
2. Criar e adicionar os botões.
3. Escrever o código do ActionScript.

4 Testar o aplicativo.

Preparação para adicionar interatividade

Para que você possa adicionar elementos interativos à sua animação, é bom configurar o arquivo FLA criando alguns locais para adicionar o novo conteúdo. Essa tarefa inclui criar espaço real no Palco em que os botões podem ser colocados. Também inclui a criação de "espaço" no arquivo FLA para manter separados os itens que são diferentes.

Para configurar o FLA para adicionar elementos interativos:

1 Crie um arquivo FLA com uma animação simples, como uma interpolação de movimento ou uma interpolação de forma. Se você já tiver um arquivo FLA contendo a animação que está apresentando no projeto, abra esse arquivo e salve-o com um novo nome.

2 Decida onde na tela você vai querer que os dois botões apareçam. Um botão serve para iniciar a animação e o outro, para criar um link para o portfólio do autor ou para a home page. Se necessário, limpe o Palco ou adicione espaço para esse novo conteúdo. Se a animação já não tiver uma, você pode criar uma tela de inicialização no primeiro quadro. Nesse caso, você provavelmente vai querer trocar a animação, para que comece no Quadro 2 ou depois disso.

3 Adicione uma nova camada, acima das outras camadas na Linha de tempo, e nomeie-a como **buttons**. É nessa camada que você vai acrescentar os botões.

4 Adicione uma nova camada, acima das camadas de botões, e renomeie-a como **actions**. Nela, você adicionará o código do ActionScript para seu aplicativo.

Criação e adição de botões

Em seguida, você vai criar e posicionar os botões que formam o centro do aplicativo interativo.

Para criar e adicionar botões ao FLA:

1 Usando as ferramentas de desenho, crie a aparência visual do seu primeiro botão (o botão "reproduzir") na camada de botões. Por exemplo, desenhe uma figura oval na horizontal com texto em cima.

2 Usando a ferramenta Seleção, selecione todas as partes gráficas do botão sozinho.

3 No menu principal, escolha Modificar > Converter em símbolo.

4 Na caixa de diálogo, escolha Botão como o tipo de símbolo, dê um nome ao símbolo e clique em OK.

5 Com o botão selecionado, no Inspetor de propriedades, dê ao botão o nome de ocorrência **playButton**.

6 Repita as etapas de 1 a 5 para criar o botão que leva o espectador à home page do autor. Chame este botão de **homeButton**.

Gravação do código

O código do ActionScript para este aplicativo pode ser dividido em três conjuntos de funcionalidade, embora todos o insiram no mesmo lugar. O código faz três coisas:

- Parar o indicador de reprodução assim que o arquivo SWF carregar (quando o indicador de reprodução entrar no Quadro 1).
- Monitorar um evento para iniciar a reprodução do arquivo SWF quando o usuário clicar no botão de reprodução.
- Monitorar um evento para enviar o navegador à URL apropriada quando o usuário clicar no botão da home page do autor.

Para criar um código que pare o indicador de reprodução quando ele entrar no Quadro 1:

1 Selecione o quadro-chave no Quadro 1 da camada de ações.

2 Para abrir o painel Ações, no menu principal, escolha Janela > Ações.

3 No painel Script, digite o seguinte código:

```
stop();
```

Para escrever código para iniciar a animação quando o botão de reprodução for clicado:

1 No fim do código digitado nas etapas anteriores, adicione duas linhas vazias.

2 Digite o seguinte código na parte inferior do script:

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

Este código define uma função chamada `startMovie()`. Quando `startMovie()` é chamado, elefaz com que a linha de tempo principal comece a reproduzir.

3 Na linha seguinte ao código adicionado na etapa anterior, digite esta linha de código:

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

Esta linha de código registra a função `startMovie()` como um ouvinte para o evento `click` de `playButton`. Em outras palavras, com ela, sempre que o botão chamado `playButton` é clicado, a função `startMovie()` é chamada.

Para escrever o código por meio do qual o navegador acessa uma URL quando o botão da home page for clicado:

1 No fim do código digitado nas etapas anteriores, adicione duas linhas vazias.

2 Digite este código na parte inferior do script:

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

Este código define uma função chamada `gotoAuthorPage()`. Essa função primeiro cria uma instância URLRequest que representa o URL http://exemplo.com/. Em seguida, ela passa esse URL para a função `navigateToURL()`, fazendo com que o navegador do usuário abra esse URL.

3 Na linha seguinte ao código adicionado na etapa anterior, digite esta linha de código:

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

Esta linha de código registra a função `gotoAuthorPage()` como um ouvinte para o evento `click` de `homeButton`. Em outras palavras, com ela, sempre que o botão chamado `homeButton` é clicado, a função `gotoAuthorPage()` é chamada.

Teste do aplicativo

Agora, o aplicativo está completamente funcional. Vamos testá-lo para ver se isso é verdade.

Para testar o aplicativo:

1 Do menu principal, escolha Controlar > Testar filme. O Flash Professional cria o arquivo SWF e o abre em uma janela do Flash Player.

2 Tente os dois botões para verificar se eles agem conforme o esperado.

3 Se os botões não funcionarem, veja algumas coisas que você pode verificar:

- Os dois botões têm nomes de ocorrência distintos?
- As chamadas de método `addEventListener()` usam os mesmos nomes que os nomes de ocorrência dos botões?
- Os nomes de evento corretos foram usados nas chamadas de método `addEventListener()`?
- Foi especificado o parâmetro correto para cada uma das funções? (Os dois métodos precisam de um único parâmetro com o tipo de dados MouseEvent.)

Todos esses erros e a maioria de outros erros possíveis resultam em uma mensagem de erro. A mensagem de erro pode aparecer quando você escolh o comando Testar filme ou quando você clica no botão enquanto está testando o projeto. Analise o painel Erros do compilador para ver os erros (aqueles que ocorrem quando você escolhe pela primeira vez o Testar filme). Verifique se há erros em tempo de execução no painel Saída, os quais ocorrem enquanto o conteúdo está sendo reproduzido, por exemplo, quando clica em um botão.

Criação de aplicativos com o ActionScript

O processo de escrever no ActionScript para criar um aplicativo vai além de apenas conhecer a sintaxe e os nomes das classes que serão usadas. A maioria da documentação da Plataforma Flash trata desses dois tópicos (sintaxe e uso das classes ActionScript). No entanto, para construir um aplicativo ActionScript, você também vai querer saber informações como:

- Que programas podem ser usados para escrever ActionScript?
- Como você organiza o código ActionScript?
- Como você inclui o código ActionScript em um aplicativo?
- Que etapas você segue para desenvolver um aplicativo ActionScript?

Opções para organizar seu código

Você pode usar o código do ActionScript 3.0 para acionar tudo, desde simples animações gráficas até sistemas complexos de processamento de transações de cliente-servidor. Dependendo do tipo de aplicativo que está sendo criado, use uma ou mais dessas formas de incluir o ActionScript no seu projeto.

Armazenamento de código em quadros em uma linha de tempo do Flash Professional

No Flash Professional, você pode adicionar código do ActionScript a qualquer quadro na linha de tempo. Esse código é executado enquanto o filme estiver sendo reproduzido, quando o indicador de reprodução entrar nesse quadro.

A colocação de código do ActionScript em quadros fornece um meio simples de adicionar comportamentos a aplicativos incorporados no Flash Professional. Você pode adicionar código a qualquer quadro na linha de tempo principal ou na linha de tempo de qualquer símbolo do MovieClip. Entretanto, essa flexibilidade tem um preço. Durante a criação de aplicativos grandes, é fácil perder o controle de quais quadros contêm quais scripts. Essa estrutura complicada pode fazer com que fique mais difícil manter o aplicativo com o tempo.

Muitos desenvolvedores simplificam a organização do código do ActionScript no Flash Professional, colocando o código somente no primeiro quadro de uma linha de tempo ou em uma camada específica no documento Flash. A separação do seu código facilita a localização e a manutenção do código nos arquivos FLA do Flash. No entanto, o mesmo código não pode ser usado em outro projeto do Flash Professional sem copiar e colar o código no novo arquivo.

Para facilitar o uso do seu código ActionScript em outros projetos do Flash Professional no futuro, armazene o código em arquivos externos do ActionScript (os arquivos de texto com a extensão .as).

Incorporação de código em arquivos MXML do Flex

Em um ambiente de desenvolvimento Flex como o Flash Builder, você pode incluir o código ActionScript dentro de uma tag `<fx:Script>` em um arquivo Flex MXML. Entretanto, essa técnica pode aumentar a complexidade nos projetos grandes e dificultar o uso do mesmo código em outro projeto Flex. Para facilitar o uso do seu código ActionScript em outros projetos Flex no futuro, armazene o código em arquivos ActionScript externos.

Nota: *Você pode especificar um parâmetro de origem para uma tag `<fx:Script>`. Usar um parâmetro de origem permite "importar" o código ActionScript de um arquivo externo, como se fosse digitado diretamente na tag `<fx:Script>`. O arquivo de origem usado, porém, não pode definir sua própria classe, que limita sua capacidade de reutilização.*

Armazenamento de código em arquivos do ActionScript

Se o seu projeto envolve uma quantidade significativa de código do ActionScript, a melhor maneira de organizar seu código é em arquivos de origem do ActionScript separados (arquivos de texto com a extensão .as). Um arquivo do ActionScript pode ser estruturado de duas formas, dependendo da maneira como você pretende usá-lo no aplicativo.

- Código do ActionScript não estruturado: linhas de código do ActionScript, incluindo instruções ou definições de funções, escritas como se fossem inseridas diretamente em um script de linha de tempo ou arquivo MXML.

  O ActionScript escrito dessa forma pode ser acessado usando a instrução `include` no ActionScript ou a tag `<fx:Script>` em Flex MXML. A instrução `include` de ActionScript diz para o compilador incluir o conteúdo de um arquivo ActionScript externo em um local específico e dentro de um determinado escopo em um script. O resultado é o mesmo que se o código fosse inserido lá diretamente. Na linguagem MXML, o uso de uma tag `<fx:Script>` com um atributo de origem identifica um ActionScript externo que o compilador carrega naquele ponto no aplicativo. Por exemplo, a seguinte tag carrega um arquivo externo do ActionScript chamado Box.as:

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

- Definição da classe do ActionScript: uma definição de uma classe do ActionScript, incluindo suas definições de método e propriedade.

  Ao definir uma classe, você pode acessar o código ActionScript na classe criando uma instância da classe e usando suas propriedades, métodos e eventos. O uso de suas próprias classes é idêntico ao uso de qualquer uma das classes ActionScript incorporadas, e exige duas partes:

  - Use a instrução `import` para especificar o nome completo da classe, para que o compilador do ActionScript saiba onde encontrá-la. Por exemplo, para usar a classe MovieClip no ActionScript, importe a classe usando seu nome completo, incluindo o pacote e a classe:

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

    Se preferir, você pode importar o pacote que contém a classe MovieClip, que é equivalente a escrever instruções `import` separadas para cada classe no pacote:

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

    As classes de nível superior são a única exceção à regra de que uma classe deve ser importada para usá-la no seu código. Essas classes não são definidas em um pacote.

  - Escreva o código que usa especificamente o nome da classe. Por exemplo, declare uma variável tendo essa classe como seu tipo de dados e crie uma ocorrência da classe para armazenar na variável. Usando uma classe no código do ActionScript, você instrui o compilador a carregar a definição dessa classe. Por exemplo, dada uma classe externa chamada Box, essa declaração cria uma ocorrência da classe Box:

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

Na primeira vez em que o compilador se deparar com a referência à classe Box, ele pesquisa o código de origem disponível para localizar a definição da classe Box.

Escolha da ferramenta correta

Você pode usar uma de várias ferramentas (ou várias ferramentas juntas) para gravar e editar seu código do ActionScript.

Flash Builder

O Adobe Flash Builder é a principal ferramenta para criar projetos com a estrutura Flex ou projetos que consistem, principalmente, de código do ActionScript. O Flash Builder também inclui um editor de ActionScript cheio de recursos, bem como recursos de layout visual e edição de MXML. Ele pode ser usado para criar projetos Flex ou apenas do ActionScript. O Flex fornece várias vantagens, incluindo um rico conjunto de controles pré-criados da interface do usuário, controles flexíveis de layout dinâmico e mecanismos embutidos para trabalhar com dados remotos e vincular dados externos a elementos da interface do usuário. No entanto, devido ao código adicional necessário para fornecer esses recursos, os projetos que usam Flex podem ter um tamanho de arquivo SWF maior do que suas contrapartes não-Flex.

Use o Flash Builder se você estiver criando aplicativos orientados a dados e cheios de recursos para a Internet com o Flex. Use-o quando desejar editar código de ActionScript, editar código MXML e dispor visuamente o seu aplicativo, tudo dentro da mesma ferramenta.

Muitos usuários do Flash Professional que criam projetos que usam muito o ActionScript utilizam o Flash Professional para criar ativos visuais e o Flash Builder como editor para o código do ActionScript.

Flash Professional

Além dos recursos gráficos e da criação de animação, o Flash Professional inclui ferramentas para trabalhar com o código do ActionScript. O código pode ser vinculado aos elementos em um arquivo FLA ou nos arquivos externos somente para ActionScript. O Flash Professional é ideal para projetos que envolvem animação ou vídeo significativo. Trata-se de algo valioso quando você deseja criar você mesmo a maioria dos ativos gráficos. Outra razão pra usar o Flash Professional para desenvolver seu projeto em ActionScript é criar ativos visuais e escrever código no mesmo aplicativo. O Flash Professional também inclui componentes pré-criados da interface do usuário. Você pode usar esses componentes para obter arquivos SWF menores e usar as ferramentas visuais para projetá-os para o seu projeto.

O Flash Professional inclui duas ferramentas para escrever códigos do ActionScript:

- Painel Ações: disponível para trabalhar em um arquivo FLA, este painel permite escrever código do ActionScript anexado a quadros em uma linha de tempo.
- Janela Script: a janela Script é um editor de texto dedicado para trabalhar com arquivos de código do ActionScript (.as).

Editor do ActionScript de terceiros

Como os arquivos do ActionScript (.as) são armazenados como arquivos de texto simples, qualquer programa que consegue editar arquivos de texto sem formatação pode ser usado para escrever arquivos do ActionScript. Além dos produtos do ActionScript da Adobe, foram criados diversos programas de edição de texto de terceiros com recursos específicos do ActionScript. Você pode escrever um arquivo MXML ou classes do ActionScript usando qualquer programa de editor de texto. Você pode então criar um aplicativo desses arquivos usando o Flex SDK. O projeto pode usar o Flex ou ser um aplicativo apenas do ActionScript. Como alternativa, alguns desenvolvedores usam o Flash Builder ou um editor de ActionScript de outra empresa para escrever classes do ActionScript, em combinação com o Flash Professional para criar conteúdo gráfico.

Entre as razões para escolher um editor de ActionScript de outra empresa estão:

- Você prefer escrever código do ActionScript em um programa separado e projetar os elementos visuais no Flash Professional.
- Você usa um aplicativo para programação que não é do ActionScript (como criação de páginas HTML ou de aplicativos em outra linguagem de programação). Você deseja usar o mesmo aplicativo para o código do ActionScript também.
- Você quer criar projetos do Flex ou somente ActionScript usando o Flex SDK sem o Flash Professional ou o Flash Builder.

Alguns dos notáveis editores de código que fornecem suporte específico ao ActionScript incluem:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (com Pacotes do ActionScript e Flex)

O processo de desenvolvimento do ActionScript

Não importando se o seu projeto do ActionScript é grande ou pequeno, usar um processo para projetar e desenvolver seu aplicativo torna o trabalho mais eficiente e eficaz. As etapas a seguir descrevem um processo de desenvolvimento básico para criar um aplicativo que usa o ActionScript 3.0:

1. Crie seu aplicativo.

   Descreva seu aplicativo de alguma forma antes de começar a criá-lo.

2. Componha o código do ActionScript 3.0.

   Você pode criar código do ActionScript usando o Flash Professional, Flash Builder, Dreamweaver ou um editor de texto.

3. Crie um projeto do Flash ou Flex para executar o código.

   No Flash Professional, crie um arquivo FLA, defina as configurações de publicação, acrescente ao aplicativo os componentes da interface do usuário e faça referência ao código do ActionScript. No Flex, defina o aplicativo, acrescente os componentes da interface do usuário usando MXML e faça referência ao código do ActionScript.

4. Publique e teste o aplicativo do ActionScript.

   Testar seu aplicativo envolve a execução do seu aplicativo de dentro do ambiente de desenvolvimento e ter a certeza de que ele faz tudo o que você queria.

Não é preciso seguir essas etapas na ordem nem concluir uma etapa completamente antes de começar outra. Por exemplo, você pode criar uma tela do aplicativo (etapa 1) e imagens gráficas, botões, etc. (etapa 3), antes de escrever código do ActionScript (etapa 2) e testar (etapa 4). Ou você pode criar parte disso e depois adicionar um botão ou um elemento da interface por vez, escrevendo o ActionScript para cada um e testando-o durante a criação. É útil lembrar esses quatro estágios do processo de desenvolvimento. No entanto, no desenvolvimento real, é mais eficaz avançar e voltar nos estágios conforme apropriado.

Criação de suas próprias classes

O processo de criar as classes que serão usadas nos projetos pode parecer assustador. Entretanto, a parte mais difícil da criação de uma classe é a tarefa de projetar os métodos, as propriedades e os eventos da classe.

Estratégias para criar uma classe

O tópico de criação orientada a objetos é complexo; existem cargos totalmente dedicados ao estudo acadêmico e à prática profissional dessa disciplina. No entanto, apresentamos algumas sugestões de abordagens que podem ajudá-lo a começar.

1. Pense na função que as ocorrências dessa classe exercem no aplicativo. Em geral, os objetos cumprem uma destas três funções:
   - Objeto Value: esses objetos servem principalmente como contêineres de dados. Provavelmente, têm várias propriedades e menos métodos (ou às vezes nenhum método). Em geral, são representações de código de itens claramente definidos. Por exemplo, um aplicativo de reprodução de música poderia incluir uma classe Song, que representa uma única canção real, e uma classe Playlist, que representa um grupo conceitual de músicas.
   - Objeto de exibição: são os objetos que realmente aparecem na tela. Entre os exemplos estão elementos da interface do usuário como uma lista suspensa ou exibição de status, elementos gráficos como criaturas em um videogame, e assim por diante.
   - Estrutura do aplicativo: esses objetos exercem uma ampla gama de funções de suporte na lógica ou no processamento executados pelos aplicativos. Por exemplo, você pode fazer um objeto realizar certos cálculos em uma simulação de biologia. Você pode criar um que seja responsável por sincronizar os valores entre o controle de dial e a leitura de volume no aplicativo de reprodução de música. Outro pode administrar as regras em um video game. Ou você pode fazer uma classe para carregar uma imagem salva em um aplicativo de desenho.
2. Escolha a funcionalidade específica de que a classe precisa. Os diferentes tipos de funcionalidade, em geral, se tornam métodos da classe.
3. Se a classe for servir como um objeto de valor, decida quais dados as ocorrências incluem. Esses itens são bons candidatos para propriedades.
4. Como a classe está sendo criada especificamente para seu projeto, o mais importante é fornecer a funcionalidade de que o aplicativo precisa. Tente responder a estas perguntas:
   - Que informações o aplicativo está armazenando, controlando e manipulando? Responder a esta pergunta ajuda-o a identificar os objetos e as propriedades de valor de que você precisa.
   - Que conjuntos de ações o aplicativo realiza? Por exemplo, o que acontece quando o aplicativo é carregado, quando se clica em um botão em particular, quando um filme para de ser reproduzido e assim por diante? Esses itens são bons candidatos para métodos. Eles também podem ser propriedade se as "ações" envolverem a alteração de valores individuais.
   - Para qualquer ação determinada, que informações são necessárias para realizar essa ação? Essas informações se tornam os parâmetros do método.
   - À medida que o aplicativo executa seu trabalho, o que muda na sua classe que outras partes do aplicativo devem saber? Esses itens são bons candidatos para eventos.

5 Há um objeto atual que seja semelhante ao objeto de que você precisa, com a diferença de que está faltando alguma funcionalidade que você deseja acrescentar? Pense em criar uma subclasse. (A *subclasse* é uma classe que se baseia na funcionalidade de uma classe já existente, e não na definição de toda sua própria funcionalidade.) Por exemplo, para criar uma classe que seja um objeto visual na tela, use o comportamento de um objeto existente de exibição como base para sua classe. Nesse caso, o objeto de exibição (como MovieClip ou Sprite) seria a *classe base*, e sua classe estenderia essa classe.

Escrita do código para uma classe

Depois de ter um projeto para sua classe, ou pelo menos alguma ideia de quais informações ele armazena e que ações realiza, a sintaxe real de escrever uma classe é bastante direta.

Veja as etapas mínimas para criar sua própria classe do ActionScript:

1 Abra um novo documento de texto no seu programa de edição de texto do ActionScript.

2 Insira uma instrução `class` para definir o nome da classe. Para acrescentar uma instrução `class`, insira as palavras `public class` e depois o nome da classe. Acrescente as chaves de abertura e fechamento para abranger o conteúdo da classe (as definições de método e propriedade). Por exemplo:

```
public class MyClass
{
}
```

A palavra `public` indica que a classe pode ser acessada de qualquer outro código. Para obter alternativas, consulte Atributos de espaço para nomes de controle de acesso.

3 Digite uma instrução `package` para indicar o nome do pacote que contém sua classe. A sintaxe é a palavra `package`, seguida pelo nome completo do pacote, seguido pelas chaves de abertura e fechamento ao redor do bloco de instrução `class`. Por exemplo, troque o código na etapa anterior para o seguinte:

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

4 Defina cada propriedade na classe usando a instrução `var` dentro do corpo da classe. A sintaxe é a mesma usada para declarar qualquer variável (com a adição do modificador `public`). Por exemplo, a inclusão destas linhas entre as chaves de abertura e fechamento da definição da classe cria propriedades chamadas `textProperty`, `numericProperty` e `dateProperty`:

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

5 Defina cada método na classe usando a mesma sintaxe usada para definir uma função. Por exemplo:

- Para criar um método `myMethod()`, digite:

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

- Para criar um construtor (o método especial que é chamado como parte do processo de criar uma ocorrência de uma classe), crie um método cujo nome corresponda exatamente ao nome da classe:

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

Se você não incluir um método constructor em sua classe, o compilador cria automaticamente um construtor vazio em sua classe. (Em outras palavras, um construtor sem parâmetros e sem instruções.)

Há alguns outros elementos de classe que você pode definir. Esses elementos são mais complexos.

- Os *assessores* são um cruzamento especial entre um método e uma propriedade. Durante a escrita do código para definir a classe, você escreve o assessor como um método. Você pode realizar várias ações em vez de apenas ler ou atribuir um valor, que é tudo o que você pode fazer ao definir uma propriedade. Entretanto, na criação de uma ocorrência da classe, você trata o assessor como uma propriedade e use apenas o nome para ler ou atribuir o valor.
- Os eventos no ActionScript não são definidos usando uma sintaxe específica. Em vez disso, você define os eventos em sua classe usando a funcionalidade da classe EventDispatcher.

Mais tópicos da Ajuda

Manipulação de eventos

Exemplo: Criação de um aplicativo básico

O ActionScript 3.0 pode ser usado em uma série de ambientes de desenvolvimento de aplicativos, incluindo as ferramentas Flash Professional e Flash Builder ou qualquer editor de texto.

Este exemplo apresenta as etapas de criação e de aprimoramento de um aplicativo ActionScript 3.0 simples usando o Flash Professional ou Flash Builder. O aplicativo que você criar apresentará um padrão simples para usar arquivos externos de classe do ActionScript 3.0 no Flash Professional e Flex.

Criação do seu aplicativo do ActionScript

Esse exemplo de aplicativo ActionScript é um aplicativo padrão do tipo "Hello World", por isso seu design é simples:

- O aplicativo é chamado de HelloWorld.
- Ele exibe um único campo de texto contendo as palavras “Hello World!”.
- O aplicativo usa uma classe única orientada a objetos chamadas Greeter. Esse projeto permite que a classe seja usada de dentro de um projeto do Flash Professional ou Flex.
- Neste exemplo, você primeiro cria uma versão básica do aplicativo. Em seguida, você aumenta a funcionalidade para fazer com que o usuário insira um nome de usuário e o aplicativo verifique o nome em uma lista de usuários conhecidos.

Com essa definição concisa estabelecida, você pode começar a criar o aplicativo em si.

Criação do projeto HelloWorld e da classe Greeter

A instrução do design para o aplicativo Hello World diz que seu código é fácil de reutilizar. Para atingir esse objetivo, o aplicativo usa uma única classe orientada a objetos chamadas Greeter. Você usa essa classe de dentro de um aplicativo que você cria no Flash Builder ou no Flash Professional.

Para criar o projeto HelloWorld e a classe Greeter no Flex:

1. No Flash Builder, selecione File > New> Flex Project.
2. Digite HelloWorld como o nome do projeto. Certifique-se de que o tipo de aplicativo esteja definido como "Web (é executado no Adobe Flash Player)" e depois clique em Concluir.

   O Flash Builder cria seu projeto e o exige no Package Explorer. Por padrão, o projeto já contém um arquivo chamado HelloWorld.mxml, e esse arquivo é aberto no painel Editor.
3. Agora, para criar um arquivo de classe do ActionScript personalizado no Flash Builder, selecione Arquivo > Novo > Classe ActionScript.
4. Na caixa de diálogo New ActionScript Class, no campo Name, digite **Greeter** para o nome da classe e, em seguida, clique em Finish.

   Uma nova janela de edição do ActionScript será exibida.

   Continue com Adição de código à classe Greeter.

Para criar a classe Greeter no Flash Professional:

1. No Flash Professional, selecione File > New.
2. Na caixa de diálogo Novo documento, selecione Arquivo ActionScript e clique em OK.

   Uma nova janela de edição do ActionScript será exibida.
3. Selecione Arquivo > Salvar. Selecione uma pasta para conter o aplicativo, chame o arquivo do ActionScript de **Greeter.as** e clique em OK.

   Continue com Adição de código à classe Greeter.

Adição de código à classe Greeter

A classe Greeter define um objeto, `Greeter`, que você usa no aplicativo HelloWorld.

Para adicionar código à classe Greeter:

1. Digite o seguinte código no novo arquivo (parte do código pode já ter sido adicionada):

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

A classe Greeter inclui um único método `sayHello()` , que retorna uma seqüência de caracteres que diz “Hello World!”.

2 Clique em Arquivo > Salvar para salvar esse arquivo do ActionScript.

A classe Greeter agora está pronta para ser usada em um aplicativo.

Criação de um aplicativo que usa o código do ActionScript

A classe Greeter que você criou define um conjunto independente de funções de software, mas não representa um aplicativo completo. Para usar a classe, você crie um documento do Flash Professional ou um projeto do Flex.

O código precisa de uma ocorrência da classe Greeter. Veja como usar a classe Greeter ao seu aplicativo.

Para criar um aplicativo do ActionScript usando o Flash Professional:

1 Selecione Arquivo > Novo.

2 Na caixa de diálogo Novo documento, selecione Arquivo Flash (ActionScript 3.0) e clique em OK.

É exibida uma nova janela de documento.

3 Selecione Arquivo > Salvar. Selecione uma pasta que contenha o arquivo de classe Greeter.as, chame o documento Flash de **HelloWorld.fla** e clique em OK.

4 Na paleta de ferramentas do Flash Professional, selecione a ferramenta Text. Arraste no Palco para definir um novo campo de texto com aproximadamente 300 pixels de largura e 100 pixels de altura.

5 No painel Propriedades, com o campo de texto ainda selecionado no Palco, defina o tipo de texto como "Texto dinâmico". Digite **mainText** como o nome de ocorrência do campo de texto.

6 Clique no primeiro quadro da linha de tempo principal. Abra o painel Ações escolhendo Janela > Ações.

7 No painel Ações, digite o seguinte script:

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

8 Salve o arquivo.

Continue com Publicação e teste do aplicativo do ActionScript.

Para criar um aplicativo do ActionScript usando o Flash Builder:

1 Abra o arquivo HelloWorld.mxml e adicione código correspondente à seguinte listagem:

```
<?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 projeto do Flex inclui quatro tags MXML:

- Uma tag `<s:Application>`, que define o contêiner Application
- Uma tag `<s:layout>`, que define o estilo de layout (layout vertical) da tag Application
- Uma tag `<fx:Script>`, que inclui código do ActionScript
- Uma tag `<s:TextArea>`, que define um campo para exibir mensagens de texto ao usuário

O código na tag `<fx:Script>` define um método `initApp()` que é chamado quando o aplicativo é carregado. O método `initApp()` define o valor de texto da TextArea `mainTxt` para a seqüência de caracteres "Hello World!" retornada pelo método `sayHello()` da classe Greeter personalizada, que você acabou de escrever.

2 Selecione Arquivo > Salvar para salvar o aplicativo.

Continue com Publicação e teste do aplicativo do ActionScript.

Publicação e teste do aplicativo do ActionScript

O desenvolvimento de software é um processo interativo. Você escreve um código, tenta compilá-lo e o edita até obter uma compilação limpa. Você executa o aplicativo compilado e testa-o para ver se representa o design pretendido. Se não representar, você edita o código novamente até que represente. Os ambientes de desenvolvimento do Flash Professional e do Flash Builder oferecem vários meios de publicar, testar e depurar seus aplicativos.

Veja as etapas básicas para testar o aplicativo HelloWorld em cada ambiente.

Para publicar e testar um aplicativo do ActionScript usando o Flash Professional:

1 Publique seu aplicativo e observe se há erros de compilação. No Flash Professional, selecione Controlar > Testar filme para compilar o código do ActionScript e executar o aplicativo HelloWorld.

2 Se forem exibidos erros ou advertências na janela Saída quando você testar seu aplicativo, conserte esses erros nos arquivos HelloWorld.fla ou HelloWorld.as. Tente testar o aplicativo novamente.

3 Se não houver nenhum erro de compilação, você vê uma janela do Flash Player mostrando o aplicativo Hello World.

Você criou um aplicativo orientado a objetos simples, mas completo, que usa o ActionScript 3.0. Continue com Aprimoramento do aplicativo HelloWorld.

Para publicar e testar um aplicativo do ActionScript usando o Flash Builder:

1 Selecione Executar > Executar HelloWorld.

2 O aplicativo HelloWorld é iniciado.

- Se forem exibidos erros ou advertências na janela Saída quando você testar seu aplicativo, conserte os erros nos arquivos HelloWorld.mxml ou Greeter.as. Tente testar o aplicativo novamente.
- Se não houver nenhum erro de compilação, uma janela de navegador é aberta mostrando o aplicativo Hello World. O texto "Hello World!" aparece.

Você criou um aplicativo orientado a objetos simples, mas completo, que usa o ActionScript 3.0. Continue com Aprimoramento do aplicativo HelloWorld.

Aprimoramento do aplicativo HelloWorld

Para tornar o aplicativo um pouco mais interessante, agora você o fará solicitar e validar um nome de usuário em relação a uma lista de nomes predefinida.

Primeiro, você atualiza a classe Greeter para adicionar nova funcionalidade. Depois, você atualiza o aplicativo para usar a nova funcionalidade.

Para atualizar o arquivo Greeter.as:

1 Abra o arquivo Greeter.as.

2 Altere o conteúdo do arquivo com o seguinte (as linhas novas e alteradas são mostradas em negrito):

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

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                           + "the Enter key.";
            }
            else if (validName(userName))
            {
```

```
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

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

A classe Greeter agora tem vários novos recursos:

- A matriz `validNames` lista os nomes de usuário válidos. A matriz é inicializada com uma lista de três nomes quando a classe Greeter é carregada.
- O método `sayHello()` agora aceita um nome de usuário e altera a saudação com base em algumas condições. Se `userName` for uma seqüência de caracteres vazia (`""`), a propriedade `greeting` será definida para solicitar um nome ao usuário. Se o nome do usuário for válido, a saudação se tornará `"Hello, `*userName*`"`. Finalmente, se as duas condições não forem atendidas, a variável `greeting` será definida como `"Sorry `*userName*`, you are not on the list"`.
- O método `validName()` retornará `true` se `inputName` for encontrado na matriz `validNames` e `false` se não for encontrado. A instrução `validNames.indexOf(inputName)` verifica cada seqüência de caracteres na matriz `validNames` em relação à seqüência de caracteres `inputName`. O método `Array.indexOf()` retorna a posição de índice da primeira ocorrência de um objeto em uma matriz. Ele retorna o valor -1 se o objeto não for encontrado na array.

Em seguida, você edita o arquivo do aplicativo que faz referência a esta classe do ActionScript.

Para modificar o aplicativo usando o Flash Professional:

1. Abra o arquivo HelloWorld.fla.
2. Modifique o script no Quadro 1 para que uma seqüência de caracteres (`""`) seja transmitida ao método `sayHello()` da classe Greeter:

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

3. Selecione a ferramenta Text na paleta de ferramentas. Crie dois novos campos de texto no Palco. Coloque-os lado a lado diretamente no campo de texto atual `mainText`.
4. No primeiro campo novo de texto, que é o rótulo, digite o texto **User Name:**.

5 No outro campo de texto novo, e no inspetor de propriedades, selecione InputText como o tipo de campo de texto. Selecione Linha única como o Tipo de linha. Digite **textIn** como o nome de ocorrência.

6 Clique no primeiro quadro da linha de tempo principal.

7 No painel Ações, adicione as seguintes linhas no final do 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);
    }
}
```

O novo código adiciona a seguinte funcionalidade:

- As primeiras duas linhas simplesmente definem bordas para os dois campos de texto.
- Um campo de texto de entrada, como o campo `textIn`, possui um conjunto de eventos que ele pode despachar. O método `addEventListener()` permite definir uma função que é executada quando um tipo de evento ocorre. Neste caso, o evento é o pressionamento de uma tecla no teclado.
- A função personalizada `keyPressed()` verifica se a tecla que foi pressionada é a tecla Enter. Caso afirmativo, ela chama o método `sayHello()` do objeto `myGreeter` , transmitindo o texto do campo de texto `textIn` como um parâmetro. Esse método retorna uma seqüência de caracteres de saudação com base no valor transmitido. A seqüência de caracteres retornada é atribuída à propriedade `text` do campo de texto `mainText`.

O script completo para o Quadro 1 é o seguinte:

```
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 Salve o arquivo.

9 Selecione Controlar > Testar filme para executar o aplicativo.

Quando você executa o aplicativo, ele lhe pede para inserir um nome de usuário. Se for válido (Sammy, Frank ou Dean), o aplicativo exibirá a mensagem de confirmação "hello".

Para modificar o aplicativo usando o Flash Builder:

1 Abra o arquivo HelloWorld.mxml.

2 Em seguida, modifique a tag `<mx:TextArea>` para indicar para o usuário que é apenas para exibição. Altere a cor de plano de fundo para um cinza claro e defina o atributo `editable` como `false`:

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

3 Agora, adicione as seguintes linhas logo depois da tag de fechamento `<s:TextArea>`. Essas linhas criam um componente TextInput que permite que o usuário insira um valor de nome de usuário:

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

O atributo `enter` define o que acontece quando o usuário pressiona a tecla Enter no campo `userNameTxt`. Neste exemplo, o código passa o texto no campo para o método `Greeter.sayHello()`. A saudação no campo `mainTxt` muda da forma correspondente.

O arquivo HelloWorld.mxml file se parece com o seguinte:

```
<?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 Salve o arquivo HelloWorld.mxml editado. Selecione Executar > Executar HelloWorld para executar o aplicativo.

Quando você executa o aplicativo, o aplicativo lhe pede para inserir um nome de usuário. Se for válido (Sammy, Frank ou Dean), o aplicativo exibirá a mensagem de confirmação "`Hello,` *userName*".

Capítulo 3: Linguagem e sintaxe do ActionScript

O ActionScript 3.0 inclui a linguagem central do ActionScript e na API (Application Programming Interface) da Plataforma Adobe Flash. A linguagem principal é a parte do ActionScript que define a sintaxe da linguagem, assim como os tipos de dados de nível superior. O ActionScript 3.0 fornece um acesso programático aos tempos de execução da Plataforma Adobe Flash: Adobe Flash Player e Adobe AIR.

Visão geral da linguagem

Os objetos são a base da linguagem do ActionScript 3.0, seus os blocos de construção fundamentais. Cada variável declarada, cada função escrita e cada ocorrência de classe criada é um objeto. Pense em um programa do ActionScript 3.0 como um grupo de objetos que realizam tarefas, respondem a eventos e se comunicam.

Os programadores acostumados à OOP (Programação orientada a objetos) em Java ou C++ podem pensar nos objetos como módulos que contêm dois tipos de membros: dados armazenados em variáveis ou propriedades de membros e comportamento acessível por meio de métodos. O ActionScript 3.0 define objetos de modo similar, com pequenas particularidades. No ActionScript 3.0, os objetos são apenas conjuntos de propriedades. Essas propriedades são contêineres que podem manter não apenas dados, mas também funções ou outros objetos. Se uma função for anexada a um objeto dessa forma, ela será chamada de método.

Embora a definição do ActionScript 3.0 possa parecer um pouco estranha aos programadores com experiência em Java ou C++, na prática, a definição dos tipos de objeto com classes do ActionScript 3.0 é bastante semelhante à forma como as classes são definidas em Java ou C++. A distinção entre as duas definições de objeto é importante ao discutir o modelo de objeto do ActionScript e outros tópicos avançados, mas, na maioria das situações, o termo *propriedades* significa variáveis de membro de classe, e não métodos. O Referência do ActionScript® 3.0 para Adobe® Flash® Platform, por exemplo, usa o termo *properties* para indicar as variáveis ou as propriedades getter-setter. Ela usa o termo *métodos* para se referir às funções que fazem parte de uma classe.

Uma diferença sutil entre as classes no ActionScript e as classes em Java ou C++ é que, no ActionScript, as classes não são apenas entidades abstratas. As classes do ActionScript são representadas por *objetos de classe* que armazenam propriedades e métodos da classe. Isso permite o uso de técnicas que podem parecer estranhas aos programadores de Java e C++, como incluir instruções ou código executável no nível superior de uma classe ou um pacote.

Outra diferença entre as classes do ActionScript e as de Java ou C++ é que toda classe do ActionScript tem o que chamamos de *objeto deprotótipo*. Nas versões anteriores do ActionScript, os objetos de protótipo, vinculados em *cadeias de protótipos*, serviam coletivamente como a base de toda a hierarquia de herança de classes. No ActionScript 3.0, contudo, os objetos de protótipo desempenham um papel secundário no sistema de herança. Apesar disso, eles poderão ser úteis como uma alternativa às propriedades e aos métodos estáticos se você quiser compartilhar uma propriedade e seu valor com todas as ocorrências de uma classe.

Anteriormente, os programadores de ActionScript avançados podiam manipular diretamente a cadeia de protótipos com elementos de linguagem embutidos especiais. Agora que essa linguagem fornece uma implementação mais madura de uma interface de programação baseada em classes, muitos desses elementos de linguagem especiais, como `__proto__` e `__resolve`, não fazem mais parte da linguagem. Além disso, as otimizações do mecanismo de herança interno que fornece melhorias de desempenho significativas impedem o acesso direto ao mecanismo de herança.

Objetos e classes

No ActionScript 3.0, cada objeto é definido por uma classe. Uma classe pode ser entendida como um modelo ou uma cópia de um tipo de objeto. As definições de classe podem incluir variáveis e constantes, que mantêm valores de dados, e métodos, que são funções de encapsulamento de comportamento vinculadas à classe. Os valores armazenados em propriedades podem ser *valores primitivos* ou outros objetos. Os valores primitivos são números, seqüências de caracteres ou valores booleanos.

O ActionScript contém diversas classes embutidas que fazem parte da linguagem central. Algumas delas, como Number, Boolean e String, representam os valores primitivos disponíveis no ActionScript. Outras classes, como Array, Math e XML, definem objetos mais complexos.

Todas as classes, incorporadas ou definidas pelo usuário, derivam da classe Object. Para os programadores com experiência no ActionScript, é importante observar que o tipo de dados Object não é mais o tipo de dados padrão, muito embora todas as outras classes ainda derivem dessa. No ActionScript 2.0, as duas linhas de código a seguir eram equivalentes porque a falta de uma anotação de tipo significava que uma variável seria do tipo Object:

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

O ActionScript 3.0, porém, apresenta o conceito de variáveis sem tipo, que podem ser designadas destas duas formas:

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

Uma variável sem tipo não é igual a uma variável do tipo Object. A principal diferença é que as variáveis sem tipo podem manter o valor especial `undefined`, enquanto que uma variável do tipo Object não pode.

Você pode definir suas próprias classes, usando a palavra-chave `class`. As propriedades de classe podem ser declaradas de três formas: as constantes podem ser definidas com a palavra-chave `const`, as variáveis são definidas com a palavra-chave `var` e as propriedades getter e setter são definidas usando os atributos `get` e `set` em uma declaração de método. Os métodos podem ser declarados com a palavra-chave `function`.

Uma ocorrência de uma classe é criada usando o operador `new`. O exemplo a seguir cria uma ocorrência da classe Date chamada `myBirthday`.

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

Pacotes e namespaces

Os pacotes e espaços para nomes são conceitos relacionados. Os pacotes permitem compactar definições de classe juntas de uma forma que facilita o compartilhamento de dados e minimiza conflitos de nome. Os espaços para nomes permitem controlar a visibilidade de identificadores, como nomes de propriedades e métodos, e podem ser aplicados ao código quer ele resida dentro ou fora de um pacote. Os pacotes permitem organizar os arquivos de classe, e os espaços para nomes permitem gerenciar a visibilidade de propriedades e métodos individuais.

Pacotes

Os pacotes no ActionScript 3.0 são implementados com espaços para nomes, mas eles não são sinônimos. Ao declarar um pacote, você cria implicitamente um tipo especial de espaço para nomes que será conhecido em tempo de compilação. Os espaços para nomes, quando criados explicitamente, não são necessariamente conhecidos em tempo de compilação.

O seguinte exemplo usa a diretiva `package` para criar um pacote simples contendo uma classe:

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

O nome da classe neste exemplo é SampleCode. Como a classe está dentro do pacote de amostras, o compilador automaticamente qualifica o nome da classe em tempo de compilação em seu nome totalmente qualificado: samples.SampleCode. O compilador também qualifica os nomes de quaisquer propriedades e métodos para que `sampleGreeting` e `sampleFunction()` se tornem `samples.SampleCode.sampleGreeting` e `samples.SampleCode.sampleFunction()`, respectivamente.

Muitos desenvolvedores, especialmente aqueles com experiência em programação Java, podem optar por colocar apenas classes no nível superior de um pacote. O ActionScript 3.0, no entanto, oferece suporte não apenas a classes no nível superior de um pacote, mas também a variáveis, funções e até mesmo instruções. Uma utilização avançada desse recurso é definir um espaço para nomes no nível superior de um pacote de forma que fique disponível para todas as classes desse pacote. Observe, porém, que somente dois especificadores de acesso, `public` e `internal`, são permitidos no nível superior de um pacote. Diferentemente de Java, que permite declarar classes aninhadas como particulares, o ActionScript 3.0 não oferece suporte a classes aninhadas nem a particulares.

Entretanto, de muitas outras formas, os pacotes do ActionScript 3.0 são semelhantes aos pacotes na linguagem de programação Java. Como você pode ver no exemplo anterior, as referências aos pacotes totalmente qualificados são expressas usando o operador dot (.), da mesma forma que em Java. Você pode usar pacotes para organizar seu código em uma estrutura hierárquica intuitiva para ser usada por outros programadores. Isso facilita o compartilhamento de código, permitindo criar seu próprio pacote para compartilhar com outros e usar pacotes criados por outros em seu código.

O uso de pacotes também ajuda a garantir que os nomes de identificador usados sejam exclusivos e não entrem em conflito com outros. Na verdade, alguns acham que essa é a maior vantagem dos pacotes. Por exemplo, dois programadores que desejam compartilhar código entre si criam uma classe chamada SampleCode. Sem pacotes, ela cria um conflito de nome e a única solução seria renomear uma das classes. Com pacotes, porém, o conflito de nomes é facilmente evitado colocando uma classe, ou de preferência as duas, em pacotes com nomes exclusivos.

Também é possível incluir pontos incorporados no nome do pacote para criar pacotes aninhados. Isso permite criar uma organização hierárquica de pacotes. Um bom exemplo disso é o pacote flash.display fornecido pelo ActionScript 3.0. O pacote flash.display é aninhado dentro do pacote flash.

Grande parte do ActionScript 3.0 é organizado com base no pacote flash. Por exemplo, o pacote flash.display contém a API de lista de exibição, e o pacote flash.events contém o novo modelo de eventos.

Criação de pacotes

O ActionScript 3.0 fornece uma flexibilidade significativa na forma de organizar pacotes, classes e arquivos de origem. As versões anteriores do ActionScript permitiam somente uma classe por arquivo de origem e exigiam que o nome do arquivo de origem correspondesse ao nome da classe. O ActionScript 3.0 permite incluir diversas classes em um único arquivo de origem, mas somente uma classe em cada arquivo pode ser disponibilizada para um código externo ao arquivo. Em outras palavras, somente uma classe em cada arquivo pode ser declarada dentro de uma declaração de pacote. As classes adicionais devem ser declaradas fora da definição do pacote, o que torna as classes invisíveis ao código fora do arquivo de origem. O nome da classe declarada dentro da definição do pacote deve corresponder ao nome do arquivo de origem.

O ActionScript 3.0 também oferece mais flexibilidade na forma de declarar pacotes. Nas versões anteriores do ActionScript, os pacotes simplesmente representavam diretórios nos quais os arquivos de origem eram colocados e os pacotes não eram declarados com a instrução `package`, mas incluíam o nome do pacote como parte do nome da classe totalmente qualificada na sua declaração de classe. Embora ainda representem diretórios no ActionScript 3.0, os pacotes podem conter mais do que apenas classes. No ActionScript 3.0, a instrução `package` é usada para declarar um pacote, o que significa que você também pode declarar variáveis, funções e espaços para nomes no nível superior de um pacote. Também é possível incluir instruções executáveis no nível superior de um pacote. Se você declarar variáveis, funções ou espaços para nomes no nível superior de um pacote, os únicos atributos disponíveis nesse nível serão `public` e `internal`, e somente uma declaração de nível de pacote por arquivo poderá usar o atributo `public`, quer a declaração seja uma classe, variável, função ou um espaço para nomes.

Os pacotes são úteis para organizar o código e evitar conflitos de nome. Não confunda o conceito de pacotes com o conceito não relacionado de herança de classe. Duas classes que residem no mesmo pacote têm um espaço para nomes em comum, mas não estão necessariamente relacionadas de outra forma. Da mesma forma, um pacote aninhado pode não ter nenhuma relação semântica com o pacote pai.

importação de pacotes

Para usar uma classe que está dentro de um pacote, você deve importar o pacote ou a classe específica. Isso difere do ActionScript 2.0, em que a importação de classes era opcional.

Por exemplo, pense no exemplo de classe SampleCode apresentado anteriormente. Se a classe residir em um pacote chamado sample, você deverá usar uma das seguintes instruções de importação usando a classe SampleCode:

```
import samples.*;
```

ou

```
import samples.SampleCode;
```

Em geral, as instruções `import` devem ser tão específicas quanto possível. Se você pretende usar apenas a classe SampleCode do pacote samples, deverá importar somente a classe SampleCode e não o pacote inteiro ao qual ela pertence. A importação de pacotes inteiros pode gerar conflitos de nome inesperados.

Você também deve colocar o código-fonte que define o pacote ou a classe no *caminho de classe*. O caminho de classe é uma lista definida pelo usuário de caminhos de diretório locais que determina onde o compilador pesquisa as classes e os pacotes importados. O caminho de classe, às vezes, é chamado de *caminho de criação* ou *caminho de origem*.

Depois de importar adequadamente a classe ou o pacote, você pode usar o nome totalmente qualificado da classe (samples.SampleCode) ou apenas o nome da classe em si (SampleCode).

Os nomes totalmente qualificados são úteis quando classes, métodos ou propriedades com nomes idênticos geram código ambíguo, mas podem ser difíceis de gerenciar se usados para todos os identificadores. Por exemplo, o uso do nome totalmente qualificado gera um código detalhado ao instanciar uma ocorrência da classe SampleCode:

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

Conforme os níveis de pacotes aninhados crescem, a legibilidade do código diminui. Nas situações em que certamente não há identificadores ambíguos, você pode tornar seu código mais fácil de ler usando identificadores simples. Por exemplo, a instanciação de uma nova ocorrência da classe SampleCode será bem menos detalhada se você usar somente o identificador de classe:

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

Se você tentar usar os nomes de identificador sem primeiro importar o pacote ou a classe apropriados, o compilador não consegue encontrar as definições de classe. Entretanto, se você importar um pacote ou uma classe, qualquer tentativa de definir um nome que entre em conflito com um nome importado gera um erro.

Durante a criação de um pacote, o especificador de acesso padrão para todos os seus membros é `internal`, o que significa que, por padrão, os membros do pacote são visíveis apenas por outros membros do mesmo pacote. Para que uma classe fique disponível para o código fora do pacote, é necessário declará-la como `public`. Por exemplo, o seguinte pacote contém duas classes, SampleCode e CodeFormatter:

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

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

A classe SampleCode é visível fora do pacote porque é declarada como uma classe `public`. A classe CodeFormatter, porém, é visível somente dentro do próprio pacote de amostras. Se você tentar acessar a classe CodeFormatter fora do pacote de amostras, será gerado um erro, como mostra o exemplo a seguir:

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

Para que as duas classes fiquem disponíveis fora do pacote, é necessário declará-las como `public`. Você não pode aplicar o atributo `public` à declaração do pacote.

Os nomes totalmente qualificados são úteis para resolver conflitos de nome que podem ocorrer durante o uso de pacotes. Esse cenário pode surgir na importação de dois pacotes que definem classes com o mesmo identificador. Por exemplo, considere o seguinte pacote, que também tem uma classe chamada SampleCode:

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

Se você importar as duas classes, como a seguir, há um conflito de nomes ao fazer referência à classe SampleCode:

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

O compilador não tem como saber qual classe SampleCode deve usar. Para resolver o conflito, você deve usar o nome totalmente qualificado de cada classe, como a seguir:

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

Nota: *Os programadores com experiência em C++ costumam confundir a instrução* `import` *com* `#include`*. A diretiva* `#include` *é necessária em C++ porque os compiladores de C++ processam um arquivo por vez e não pesquisam definições de classes em outros arquivos, a menos que um arquivo de cabeçalho seja incluído explicitamente. O ActionScript 3.0 tem uma diretiva* `include`*, mas não foi criado para importar classes e pacotes. Para importar classes ou pacotes no ActionScript 3.0, é necessário usar a instrução* `import` *e colocar o arquivo de origem que contém o pacote no caminho da classe.*

Espaços para nomes

Os espaços para nomes fornecem controle sobre a visibilidade das propriedades e dos métodos criados. Pense nos especificadores de controle de acesso `public`, `private`, `protected` e `internal` como espaços para nomes embutidos. Se esses especificadores de controle de acesso predefinidos não atenderem às suas necessidades, você poderá definir seus próprios espaços para nomes.

Se você está familiarizado com espaços para nomes XML, boa parte desta discussão não será novidade, embora a sintaxe e os detalhes da implementação do ActionScript sejam ligeiramente diferentes do XML. Se nunca trabalhou com espaços para nomes antes, o conceito em si é simples, mas a implementação tem uma terminologia específica que você deverá aprender.

Para entender como os espaços para nomes funcionam, é bom saber que o nome de uma propriedade ou método sempre contém duas partes: um identificador e um espaço para nomes. O identificador é o que normalmente entendemos como um nome. Por exemplo, os identificadores na seguinte definição de classe são `sampleGreeting` e `sampleFunction()`:

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

Sempre que as definições não forem precedidas por um atributo de espaço para nomes, seus nomes serão qualificados pelo espaço para nomes `internal` padrão, o que significa que ficam visíveis apenas para os chamadores no mesmo pacote. Se o compilador estiver definido no modo estrito, ele emitirá um aviso de que o espaço para nomes `internal` se aplica a qualquer identificador sem um atributo de espaço para nomes. Para garantir que um identificador fique disponível em todo lugar, é necessário que seu nome seja precedido especificamente pelo atributo `public`. No código anterior, `sampleGreeting` e `sampleFunction()` têm um valor de espaço para nomes `internal`.

Há três etapas básicas que devem ser seguidas ao usar espaços para nomes: Em primeiro lugar, defina o espaço para nomes usando a palavra-chave `namespace`. Por exemplo, o código a seguir define o espaço para nomes `version1`:

```
namespace version1;
```

Em segundo lugar, você deve aplicar o espaço para nomes usando-o no lugar de um especificador de controle de acesso em uma declaração de propriedade ou método. O exemplo a seguir coloca uma função chamada `myFunction()` no espaço para nomes `version1`:

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

Por último, depois de aplicar o espaço para nomes, você pode fazer referência a ele com a diretiva `use` ou qualificando o nome de um identificador com um espaço para nomes. O exemplo a seguir faz referência à função `myFunction()` por meio da diretiva `use`:

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

Também é possível usar um nome qualificado para fazer referência à função `myFunction()`, como mostra o exemplo a seguir:

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

Definição de espaços para nomes

Os espaços para nomes contêm um valor, o URI (localizador uniforme de recursos), que às vezes é chamado de *nome do espaço para nomes*. Um URI permite garantir que a definição do espaço para nomes seja exclusiva.

Você cria um espaço para nomes declarando uma definição para ele de duas formas. Você pode definir um espaço para nomes com um URI explícito, assim como definiria um espaço para nomes XML, ou pode omitir o URI. O exemplo a seguir mostra como um espaço para nomes pode ser definido usando um URI:

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

O URI funciona como uma seqüência de caracteres de identificação exclusiva para o espaço para nomes. Se você omitir o URI, como no exemplo a seguir, o compilador cria uma sequência de caracteres de identificação interna exclusiva no lugar do URI. Você não possui acesso a essa seqüência de caracteres de identificação interna.

```
namespace flash_proxy;
```

Depois de definido, com ou sem um URI, o espaço para nomes não poderá ser redefinido no mesmo escopo. A tentativa de definir um espaço para nomes definido anteriormente no mesmo escopo irá gerar um erro de compilação.

Se for definido dentro de um pacote ou uma classe, talvez o espaço para nomes não fique visível para o código fora do pacote ou da classe, a menos que seja usado o especificador de controle de acesso apropriado. Por exemplo, o seguinte código mostra o espaço para nomes `flash_proxy` definido com o pacote flash.utils. No exemplo a seguir, a falta de um especificador de controle de acesso significa que o espaço para nomes `flash_proxy` deve ser visível apenas para o código dentro do pacote flash.utils e não para qualquer outro código fora do pacote:

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

O código a seguir usa o atributo `public` para tornar o espaço para nomes `flash_proxy` visível para o código fora do pacote:

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

Aplicação de espaços para nomes

Aplicar um espaço para nomes significa colocar uma definição em um espaço para nomes. As definições que podem ser colocadas em espaços para nomes incluem funções, variáveis e constantes (não é possível colocar uma classe em um espaço para nomes personalizado).

Considere, por exemplo, uma função declarada usando o espaço para nomes de controle de acesso `public`. O uso do atributo `public` em uma definição de função coloca a função no espaço público para nomes, tornando-a disponível para todo o código. Depois de definir um espaço para nomes, você pode usá-lo da mesma forma que usa o atributo `public`, e a definição fica disponível para o código que pode referenciar o seu espaço para nomes personalizado. Por exemplo, se você definir um espaço para nomes `example1`, poderá adicionar um método chamado `myFunction()` usando `example1` como um atributo, como mostra este exemplo:

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

A declaração do método `myFunction()` usando o espaço para nomes `example1` como um atributo significa que o método pertence ao espaço para nomes `example1`.

Tenha em mente o seguinte durante a aplicação de espaços para nomes:

- Você só pode aplicar um espaço para nomes por declaração.
- Não há como aplicar um atributo de espaço para nomes a mais de uma definição por vez. Em outras palavras, se você quiser aplicar seu espaço para nomes a dez funções diferentes, deverá adicioná-lo como um atributo a cada uma das dez definições de função.
- Se aplicar um espaço para nomes, também não será possível definir um especificador de controle de acesso porque espaços para nomes e especificadores de acesso são mutuamente exclusivos. Ou seja, não é possível declarar uma função ou propriedade como `public`, `private`, `protected` ou `internal` e aplicar o espaço para nomes.

Referência a espaços para nomes

Não é necessário fazer referência a um espaço para nomes explicitamente durante o uso de um método ou uma propriedade declarados com qualquer um dos espaços para nomes de controle de acesso, como `public`, `private`, `protected` e `internal`. Isso porque o acesso a esses espaços para nomes especiais é controlado por contexto. Por exemplo, as definições colocadas no espaço para nomes `private` ficam disponíveis automaticamente para o código dentro da mesma classe. Para os espaços para nomes que você definir, porém, essa diferenciação de contexto não existe. Para usar um método ou uma propriedade colocados em um espaço para nomes personalizado, é necessário fazer referência ao espaço para nomes.

Você pode fazer referência a espaços para nomes com a diretiva `use namespace` ou pode qualificar o nome com o espaço para nomes usando o pontuador do qualificador de nome (`::`). A referência a um espaço para nomes com a diretiva `use namespace` "abre" o espaço para nomes, para que ele possa ser aplicado a quaisquer identificadores não qualificados. Por exemplo, se definir o espaço para nomes `example1`, você poderá acessar seus nomes usando `use namespace example1`:

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

É possível abrir mais de um espaço para nomes por vez. Quando aberto com `use namespace`, o espaço para nomes permanecerá aberto em todo o bloco de código no qual se encontra. Não há como fechar explicitamente um espaço para nomes.

O uso de mais de um espaço para nomes, contudo, aumenta a probabilidade de conflitos de nome. Se preferir não abrir um espaço para nomes, você poderá evitar a diretiva `use namespace` qualificando o nome do método ou da propriedade com o espaço para nomes e o pontuador do qualificador de nome. Por exemplo, o seguinte código mostra como qualificar o nome `myFunction()` com o espaço para nomes `example1`:

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

Uso de espaços para nomes

Um exemplo real de um espaço para nomes usado para evitar conflitos de nome é a classe flash.utils.Proxy que faz parte do ActionScript 3.0. A classe Proxy, que é a substituição para a propriedade `Object.__resolve` do ActionScript 2.0, permite interceptar diferenças em propriedades ou métodos não definidos antes da ocorrência de um erro. Todos os métodos da classe Proxy residem no espaço para nomes `flash_proxy` para evitar conflitos de nome.

Para entender melhor como o espaço para nomes `flash_proxy` é usado, é preciso entender como usar a classe Proxy. A funcionalidade da classe Proxy está disponível somente para suas classes herdadas. Em outras palavras, se quiser usar os métodos da classe Proxy em um objeto, a definição de classe do objeto deve estender a classe Proxy. Por exemplo, para interceptar as tentativas de chamar um método não definido, é necessário estender a classe Proxy e substituir seu método `callProperty()`.

Você deve se lembrar de que a implementação de espaços para nomes, em geral, é um processo de três etapas: definir, aplicar e referenciar um espaço para nomes. Como os métodos da classe Proxy nunca são chamados explicitamente, o espaço para nomes `flash_proxy` é definido e aplicado, mas não referenciado. O ActionScript 3.0 define o espaço para nomes `flash_proxy` e o aplica na classe Proxy. O código precisa apenas aplicar o espaço para nomes `flash_proxy` às classes que estendem a classe Proxy.

O espaço para nomes `flash_proxy` é definido no pacote flash.utils de forma semelhante à seguinte:

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

O espaço para nomes é aplicado aos métodos da classe Proxy como mostrado no seguinte trecho dessa classe:

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

Como mostra o código a seguir, primeiro você deve importar a classe Proxy e o espaço para nomes `flash_proxy`. Depois, deve declarar sua classe de forma que estenda a classe Proxy (você também deve adicionar o atributo `dynamic` se estiver compilando no modo estrito). Ao substituir o método `callProperty()`, o espaço para nomes `flash_proxy` deve ser usado.

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

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

Se criar uma ocorrência da classe MyProxy e chamar um método não definido, tal como o método `testing()` chamado no exemplo a seguir, seu objeto Proxy irá interceptar a chamada de método e executar as instruções dentro do método `callProperty()` (neste caso, uma instrução `trace()` simples).

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

Há duas vantagens em ter os métodos da classe Proxy dentro do espaço para nomes `flash_proxy`. Primeiro, ter um espaço para nomes separado reduz a desordem na interface pública de qualquer classe que estende a classe Proxy. (Há aproximadamente uma dúzia de métodos na classe Proxy que podem ser substituídos e nenhum foi criado para ser chamado diretamente. Colocá-los no espaço para nomes público poderia gerar confusão.) Em segundo lugar, o uso do espaço para nomes `flash_proxy` evitará os conflitos de nome caso a subclasse Proxy contenha métodos de ocorrência com nomes que correspondem a qualquer método da classe Proxy. Por exemplo, você pode querer chamar um de seus métodos de `callProperty()`. O código a seguir é aceitável, porque sua versão do método `callProperty()` está em um espaço para nomes diferente:

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

Os espaços para nomes também podem ser úteis para fornecer acesso a métodos ou propriedades de uma forma que não seria possível com os quatro especificadores de controle de acesso (`public`, `private`, `internal` e `protected`). Por exemplo, você pode ter alguns métodos utilitários espalhados em vários pacotes. Você quer disponibilizar esses métodos para todos os pacotes, mas não quer que sejam públicos. Para fazer isso, você pode criar um espaço para nomes e usá-lo como seu próprio especificador de controle de acesso especial.

O exemplo a seguir usa um espaço para nomes definido pelo usuário para agrupar duas funções que residem em pacotes diferentes. Ao agrupá-las no mesmo espaço para nomes, você pode tornar as duas funções visíveis para uma classe ou um pacote por meio de uma única instrução `use namespace`.

Este exemplo usa quatro arquivos para demonstrar a técnica. Todos os arquivos devem estar no caminho de classe. O primeiro deles, myInternal.as, é usado para definir o espaço para nomes `myInternal`. Como o arquivo está em um pacote chamado example, você deve colocá-lo em uma pasta chamada example. O espaço para nomes é marcado como `public` para que possa ser importado para outros pacotes.

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

O segundo e terceiro arquivos, Utility.as e Helper.as, definem as classes que contêm os métodos que devem estar disponíveis para outros pacotes. A classe Utility é um pacote example.alpha, o que significa que o arquivo deve ser colocado dentro de uma pasta chamada alpha, que é uma subpasta da pasta example. A classe Helper é um pacote example.beta, o que significa que o arquivo deve ser colocado dentro de uma pasta chamada beta, que também é uma subpasta da pasta example. Os dois pacotes, example.alpha e example.beta, devem importar o espaço para nomes antes de usá-lo.

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

O quarto arquivo, NamespaceUseCase.as, é a classe de aplicativo principal e deve ser uma irmã da pasta example. No Flash Professional, essa classe deveria ser usada como a classe de documento para o FLA. A classe NamespaceUseCase também importa o espaço para nomes `myInternal` e o usa para chamar os dois métodos estáticos que residem nos outros pacotes. O exemplo usa métodos estáticos apenas para simplificar o código. Os métodos estáticos e de ocorrência podem ser colocados no espaço para nomes `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]
        }
    }
}
```

Variáveis

As variáveis permitem armazenar valores usados no programa. Para declarar uma variável, você deve usar a instrução `var` com o nome da variável. No ActionScript 3.0, o uso da instrução `var` é sempre necessário. Por exemplo, a seguinte linha do ActionScript declara uma variável chamada `i`:

```
var i;
```

Se omitir a instrução `var` ao declarar uma variável, você obtém um erro de compilador no modo estrito e um erro de tempo de execução no modo padrão. Por exemplo, a seguinte linha de código resulta em um erro se a variável `i` não for definida antes:

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

A associação de uma variável a um tipo de dados deve ser feita durante a declaração da variável. A declaração de uma variável sem designar seu tipo é legal, mas gera um aviso do compilador no modo estrito. Um tipo de variável é designado anexando o nome da variável ao caractere dois-pontos (:) seguido do tipo da variável. Por exemplo, o seguinte código declara uma variável `i` que é do tipo int:

```
var i:int;
```

Você atribui um valor à variável usando o operador de atribuição (=). Por exemplo, o seguinte código declara uma variável `i` e lhe atribui o valor 20:

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

Pode ser mais conveniente atribuir um valor a uma variável ao mesmo que em que ela é declarada, como no exemplo a seguir:

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

A técnica de atribuir um valor a uma variável no momento em que ela é declarada é comumente usado não apenas para atribuir valores primitivos, como inteiros e seqüências de caracteres, mas também para criar uma matriz ou instanciação de uma ocorrência de uma classe. O exemplo a seguir mostra a declaração e a atribuição de um valor a uma matriz usando uma única linha de código:

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

É possível criar uma ocorrência de uma classe usando o operador `new`. O exemplo a seguir cria uma ocorrência de uma classe chamada `CustomClass` e atribui uma referência para a ocorrência de classe recém-criada à variável chamada `customItem`:

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

Se tiver mais de uma variável a declarar, você poderá declará-las em uma única linha de código usando o operador vírgula (`,`) para separar as variáveis. Por exemplo, o seguinte código declara três variáveis em uma única linha de código:

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

Você também pode atribuir valores a cada variável na mesma linha de código. Por exemplo, o seguinte código declara três variáveis (`a`, `b` e `c`) e atribui um valor a cada uma:

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

Embora você possa usar o operador vírgula para agrupar declarações de variáveis em uma instrução, isso pode reduzir a legibilidade do código.

Noções básicas sobre o escopo de variáveis

O *escopo* de uma variável é a área do código em que a variável pode ser acessada por uma referência léxica. Uma variável *global* é aquela definida em todas as áreas do seu código, enquanto que uma variável *local* é aquela definida apenas em uma parte dele. No ActionScript 3.0, às variáveis é sempre atribuído o escopo da função ou classe em que elas são declaradas. Uma variável global é aquela especificada fora de qualquer definição de função ou classe. Por exemplo, o seguinte código cria uma variável global `i` declarando-a fora de qualquer função: O exemplo mostra que uma variável global está disponível tanto dentro quanto fora da definição da função.

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

A variável local é declarada dentro de uma definição de função. A menor área de código para a qual é possível definir uma variável local é uma definição de função. Uma variável local declarada dentro de uma função existe somente nessa função. Por exemplo, se você declarar uma variável chamada `str2` dentro de uma função chamada `localScope()`, essa variável não fica disponível fora da função.

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

Se o nome de variável usado para a variável local já estiver declarado como uma variável global, a definição local ocultará (ou obscurecerá) a definição global enquanto a variável local estiver no escopo. A variável global ainda existe fora da função. Por exemplo, o código a seguir cria uma variável de seqüência de caracteres global chamada `str1` e uma variável local de mesmo nome dentro da função `scopeTest()`. A instrução `trace` dentro da função gera o valor local da variável, mas a instrução `trace` fora da função gera o valor global da variável.

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

As variáveis do ActionScript, diferentemente de C++ e Java, não possuem escopo em nível de bloqueio. Um código de bloqueio é qualquer grupo de instruções entre um colchete de abertura ({) e um de fechamento (}). Em algumas linguagens de programação, como C++ e Java, as variáveis declaradas dentro de um bloco de código não ficam disponíveis fora dele. Essa restrição de escopo é chamada de escopo em nível de bloqueio e não existe no ActionScript. Se você declarar uma variável dentro de um bloco de código, ela fica disponível não apenas nesse bloco, mas também em outras partes da função à qual o bloco pertence. Por exemplo, a seguinte função contém variáveis que são definidas em vários escopos de bloco. Todas as variáveis estão disponíveis na função.

```
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"]);
```

Uma implicação interessante da falta de escopo em nível de bloco é que você pode ler ou gravar em uma variável antes que ela seja declarada, contanto que ela seja declarada antes que a função termine. Isso é possível por causa de uma técnica chamada *içamento*, que significa que o compilador move todas as declarações de variável para o início da função. Por exemplo, o código a seguir é compilado muito embora a função inicial `trace()` para a variável `num` ocorra antes que a variável `num` seja declarada:

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

O compilador, porém, não içará nenhuma instrução de atribuição. Isso explica por que o `trace()` inicial de `num` resulta em `NaN` (e não um número), que é o valor padrão para as variáveis do tipo de dados Number. Isso significa que você pode atribuir valores a variáveis mesmo antes que elas sejam declaradas, como mostra o seguinte exemplo:

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

Valores padrão

Um *valor padrão* é o valor que uma variável contém antes que seu valor seja definido. Uma variável será *inicializada* quando seu valor for definido pela primeira vez. Se você declarar uma variável, mas não definir seu valor, ela será uma variável *não inicializada*. O valor de uma variável não inicializada depende de seu tipo de dados. A tabela a seguir descreve os valores padrão de variáveis, organizados por tipo de dados:

Tipo de dados	**Valor padrão**
Booleano	`false`
int	0
Number	`NaN`
Objeto	`null`
String	`null`
uint	0
Não declarado (equivalente à anotação de tipo `*`)	`undefined`
Todas as outras classes, inclusive classes definidas pelo usuário.	`null`

Para variáveis do tipo Number, o valor padrão é `NaN` (e não um número), que é um valor especial definido pelo padrão IEEE-754 para indicar um valor que não representa um número.

Se você declarar uma variável, mas não seu tipo de dados, o tipo de dados padrão `*` é aplicado, o que significa que, na verdade, a variável é sem tipo. Se você também não inicializar uma variável sem tipo com um valor, seu valor padrão será `undefined`.

Para tipos de dados que não forem Boolean, Number, int e uint, o valor padrão de qualquer variável não inicializada será `null`. Isso se aplica a todas as classes definidas pelo ActionScript 3.0, bem como a quaisquer classes personalizadas que você criar.

O valor `null` não é um valor válido para variáveis do tipo Boolean, Number, int ou uint. Se você tentar atribuir um valor `null` a esse tipo de variável, o valor será convertido no valor padrão para esse tipo de dados. Para variáveis do tipo Object, é possível atribuir um valor `null`. Se você tentar atribuir um valor `undefined` a uma variável do tipo Object, o valor será convertido em `null`.

Para variáveis do tipo Number, existe uma função especial de nível superior chamada `isNaN()` que retornará o valor booleano `true` se a variável não for um número e `false` se for.

Tipos de dados

Um *tipo de dados* define um conjunto de valores. Por exemplo, o tipo de dados Boolean é o conjunto de exatamente dois valores: `true` e `false`. Além do tipo de dados Boolean, o ActionScript 3.0 define vários tipos de dados mais comumente usados, como String, Number e Array. Você pode escolher seus próprios tipos de dados usando classes ou interfaces para definir um conjunto de valores personalizado. Todos os valores no ActionScript 3.0, sejam primitivos ou complexos, são objetos.

Um *valor primitivo* é aquele que pertence a um dos seguintes tipos de dados: Boolean, int, Number, String e uint. Trabalhar com valores primitivos, em geral, é mais rápido do que trabalhar com valores complexos, porque o ActionScript armazena valores primitivos de uma forma especial que torna as otimizações de memória e velocidade possíveis.

Nota: *Para os leitores interessados nos detalhes técnicos, o ActionScript armazena valores primitivos internamente como objetos imutáveis. O fato de serem armazenados como objetos imutáveis significa que transmitir por referência, na prática, é o mesmo que transmitir por valor. Isso reduz o uso de memória e aumenta a velocidade de execução, porque as referências são significativamente menores do que os valores em si.*

Um *valor complexo* é um valor que não é primitivo. Os tipos de dados que definem conjuntos de valores complexos incluem Array, Date, Error, Function, RegExp, XML e XMLList.

Muitas linguagens de programação distinguem os valores primitivos dos objetos delimitadores. Java, por exemplo, tem um primitivo int e a classe java.lang.Integer que o delimita. Os primitivos Java não são objetos, mas seus delimitadores são, o que torna os primitivos úteis para algumas operações e os objetos delimitadores mais adequados para outras operações. No ActionScript 3.0, os valores primitivos e seus objetos delimitadores são, na prática, indistinguíveis. Todos os valores, mesmo os primitivos, são objetos. O tempo de execução trata esses tipos primitivos como casos especiais que se comportam como objetos, mas não exige a sobrecarga normal associada à criação de objetos. Isso significa que as duas linhas de código a seguir são equivalentes:

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

Todos os tipos de dados primitivos e complexos listados acima são definidos pelas classes centrais do ActionScript 3.0. As classes centrais permitem criar objetos usando os valores literais em vez do operador `new`. Por exemplo, você pode criar uma matriz usando um valor literal ou o construtor de classe Array, como a seguir:

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

Verificação de tipos

A verificação de tipos pode ocorrer em tempo de compilação ou de execução. Linguagens tipificadas estatisticamente, como C++ e Java, executam a verificação de tipos em tempo de compilação. Linguagens tipificadas dinamicamente, como Smalltalk e Python, manipulam a verificação de tipos em tempo de execução. Como uma linguagem tipificada dinamicamente, o ActionScript 3.0 tem uma verificação de tipos em tempo de execução, mas também oferece suporte à verificação de tipos em tempo de compilação com um modo de compilador especial chamado *modo estrito*. No modo estrito, a verificação de tipos ocorre em tempo de compilação e de execução, mas no modo padrão, ela ocorre apenas em tempo de execução.

Linguagens tipificadas dinamicamente oferecem grande flexibilidade na estruturação do código, mas às custas de permitir que erros de tipo se manifestem em tempo de execução. Linguagens tipificadas estatisticamente relatam erros de tipo em tempo de compilação, mas exigem que as informações de tipo sejam conhecidas em tempo de compilação.

Verificação de tipos em tempo de compilação

A verificação de tipos em tempo de compilação é mais vantajosa em projetos grandes porque, conforme o tamanho de um projeto aumenta, a flexibilidade do tipo de dados se torna menos importante do que a rápida detecção de erros de tipo. É por isso que, por padrão, o compilador do ActionScript no Flash Professional e no Flash Builder é definido para ser executado no modo restrito.

Adobe Flash Builder
Você pode desabilitar o modo restrito no Flash Builder com as configurações do compilador do ActionScript na caixa de diálogo Propriedades do projeto.

Para fornecer a verificação de tipos em tempo de compilação, o compilador precisa conhecer as informações de tipo de dados para as variáveis ou expressões no seu código. Para declarar explicitamente um tipo de dados para uma variável, adicione o operador dois-pontos (:) seguido do tipo de dados como um sufixo para o nome da variável. Para associar um tipo de dados a um parâmetro, use o operador dois-pontos seguido do tipo de dados. Por exemplo, o seguinte código adiciona informações de tipo de dados ao parâmetro `xParam` e declara uma variável `myParam` com um tipo de dados explícito:

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

No modo estrito, o compilador do ActionScript relata incompatibilidades de tipos como erros do compilador. Por exemplo, o código a seguir declara um parâmetro de função `xParam`, do tipo Object, mas depois tenta atribuir valores do tipo String e Number ao parâmetro. Isso gera um erro do compilador no modo estrito.

```
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);
    }
}
```

Mesmo no modo estrito, porém, é possível optar seletivamente pela verificação de tipos em tempo de compilação deixando o lado direito de uma instrução de atribuição sem tipos. Você pode marcar uma variável ou expressão como sem tipo omitindo uma anotação de tipo ou usando a anotação de tipo especial de asterisco (`*`). Por exemplo, se o parâmetro `xParam` no exemplo anterior for modificado de forma que não tenha mais uma anotação de tipo, o código é compilado no modo estrito:

```
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");
```

Verificação de tipos em tempo de execução

A verificação de tipos em tempo de execução ocorrerá no ActionScript 3.0 se você compilar em modo restrito ou em modo de padrão. Considere uma situação em que o valor 3 é transmitido como um argumento para uma função que espera uma matriz. No modo estrito, o compilador irá gerar um erro, porque o valor 3 não é compatível com o tipo de dados Array. Se você desabilitar o modo estrito e executar no modo padrão, o compilador não reclamará sobre incompatibilidade de tipos, mas a verificação em tempo de execução resulta em um erro em tempo de execução.

O exemplo a seguir mostra uma função chamada `typeTest()` que espera um argumento Array mas tem um valor transmitido de 3. Isso gera um erro em tempo de execução no modo padrão, porque o valor 3 não é um membro do tipo de dados (Array) declarado do parâmetro.

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

Também pode haver situações em que há um erro de tipo em tempo de execução mesmo durante a operação no modo estrito. Isso é possível se você usar o modo estrito, mas optar pela verificação de tipos em tempo de compilação, usando uma variável sem tipo. O uso de uma variável sem tipo não elimina a verificação de tipos, mas a suspende até o tempo de execução. Por exemplo, se a variável `myNum` do exemplo anterior não tiver um tipo de dados declarado, o compilador não pode detectar a incompatibilidade de tipos, mas o código vai gerar um erro de tempo de execução porque compara o valor do tempo de execução de `myNum`, que está definido como 3 como resultado da instrução de atribuição, com o tipo de `xParam`, que é definido com o tipo de dados Array.

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

A verificação de tipos em tempo de execução também permite um uso mais flexível de herança que a verificação em tempo de compilação. Com a suspensão da verificação de tipos para o tempo de execução, o modo padrão permite referenciar as propriedades de uma subclasse mesmo que você a *eleve*. Uma elevação ocorrerá quando você usar uma classe base para declarar o tipo de uma ocorrência de classe, mas usar uma subclasse para instanciá-la. Por exemplo, você pode criar uma classe chamada ClassBase que pode ser estendida (classes com o atributo `final` não podem ser estendidas):

```
class ClassBase
{
}
```

Depois, você pode criar uma subclasse de uma ClassBase chamada ClassExtender, que tem uma propriedade chamada `someString`, como a seguir:

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

Usando as duas classes, é possível criar uma ocorrência de classe que é declarada usando o tipo de dados ClassBase, mas instanciada usando o construtor ClassExtender. Uma elevação é considerada uma operação segura, porque a classe base não contém nenhuma propriedade ou método que não esteja na subclasse.

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

Uma subclasse, porém, contém propriedades ou métodos que sua classe base não contém. Por exemplo, a classe ClassExtender contém a propriedade `someString`, que não existe na classe ClassBase. No modo padrão do ActionScript 3.0, é possível referenciar essa propriedade usando a ocorrência `myClass` sem gerar um erro de tempo de compilação, como mostra o seguinte exemplo:

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

O operador is

O operador `is` permite testar se uma variável ou expressão é um membro de um determinado tipo de dados. Nas versões anteriores do ActionScript, o operador `instanceof` fornecia essa funcionalidade, mas, no ActionScript 3.0, o operador `instanceof` não deve ser usado para testar a associação de tipo de dados. O operador `is` deve ser usado no lugar do operador `instanceof` para verificação de tipos manual, porque a expressão `x instanceof y` apenas verifica a existência de `x` na cadeia de protótipos `y` (e, no ActionScript 3.0, a cadeia de protótipos não fornece um retrato completo da hierarquia de herança).

O operador `is` examina a hierarquia de herança apropriada e pode ser usado para verificar não apenas se um objeto é uma ocorrência de uma classe específica, mas também de uma classe que implementa uma determinada interface. O exemplo a seguir cria uma ocorrência da classe Sprite, chamada `mySprite` e usa o operador `is` para testar se `mySprite` é uma ocorrência das classes Sprite e DisplayObject e se implementa a interface IEventDispatcher:

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

O operador `is` verifica a hierarquia de herança e relata adequadamente que `mySprite` é compatível com as classes Sprite e DisplayObject (a classe Sprite é uma subclasse da classe DisplayObject). O operador `is` também verifica se `mySprite` é herdada de alguma classe que implementa a interface IEventDispatcher. Como a classe Sprite é herdada da classe EventDispatcher, que implementa a interface IEventDispatcher, o operador `is` relata corretamente que `mySprite` implementa a mesma interface.

O exemplo a seguir mostra os mesmos testes do exemplo anterior, mas com `instanceof` em vez do operador `is`. O operador `instanceof` identificará corretamente que `mySprite` é uma ocorrência de Sprite ou DisplayObject, mas retornará `false` quando usado para testar se `mySprite` implementa a interface IEventDispatcher.

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

O operador as

O operador `as` também permite verificar se uma expressão é um membro de um determinado tipo de dados. Diferentemente do operador `is`, porém, o operador `as` não retorna um valor booleano. Em vez disso, o operador `as` retorna o valor da expressão em vez de `true` e `null` em vez de `false`. O exemplo a seguir mostra os resultados do uso do operador `as` em vez de `is` no caso simples de verificar se uma ocorrência de Sprite é um membro dos tipos de dados DisplayObject, IEventDispatcher e Number.

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

Durante o uso do operador `as`, o operando à direita deve ser um tipo de dados. Uma tentativa de usar uma expressão diferente de um tipo de dados como operando à direita resultará em um erro.

Classes dinâmicas

Uma classe *dynamic* define um objeto que pode ser alterado em tempo de execução adicionando ou alterando propriedades e métodos. Uma classe que não é dinâmica, como a classe String, é uma classe *selada*. Não é possível adicionar propriedades ou métodos a uma classe selada em tempo de execução.

As classes dinâmicas são criadas com o uso do atributo `dynamic` ao declarar uma classe. Por exemplo, o código a seguir cria uma classe dinâmica chamada `Protean`:

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

Se, posteriormente, você instanciar uma ocorrência da classe `Protean`, você pode adicionar propriedades ou métodos a ela fora da definição da classe. Por exemplo, o código a seguir cria uma ocorrência da classe `Protean` e adiciona uma propriedade chamada `aString` e outra chamada `aNumber` à ocorrência:

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

As propriedades adicionadas a uma ocorrência de uma classe dinâmica são entidades de tempo de execução, por isso qualquer tipo de verificação é feito em tempo de execução. Não é possível adicionar uma anotação de tipo a uma propriedade adicionada dessa forma.

Você também pode adicionar um método à ocorrência `myProtean` definindo uma função e anexando-a a uma propriedade da ocorrência `myProtean`. O código a seguir move a instrução de rastreamento para um método chamado `traceProtean()`:

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

Os métodos criados dessa forma, entretanto, não têm acesso a qualquer propriedade ou método particular da classe Protean. Além disso, mesmo as referências às propriedades ou métodos públicos da classe `Protean` devem ser qualificados com a palavra-chave `this` ou o nome da classe. O exemplo a seguir mostra a tentativa do método `traceProtean()` de acessar as variáveis particulares e públicas da classe `Protean`.

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

Descrições de tipos de dados

Os tipos nativos primitivos incluem Boolean, int, Null, Number, String, uint e void. As classes base do ActionScript também definem os seguintes tipos de dados complexos: Object, Array, Date, Error, Function, RegExp, XML e XMLList.

Tipo de dados Boolean

O tipo de dados Boolean inclui dois valores: `true` e `false`. Nenhum outro valor é válido para variáveis do tipo Boolean. O valor padrão de uma variável Boolean que foi declarada mas não inicializada é `false`.

Tipo de dados int

O tipo de dados int é armazenado internamente como um inteiro de 32 bits e inclui o conjunto de inteiros de

-2.147.483.648 (-2^{31}) a 2.147.483.647 ($2^{31} - 1$), inclusive. As versões anteriores do ActionScript ofereciam apenas o tipo de dados Number, que era usado para números inteiros e de ponto flutuante. No ActionScript 3.0, agora você tem acesso a tipos de computador de nível baixo para inteiros de 32 bits assinados e não assinados. Se a sua variável não tiver números de ponto flutuante, o uso do tipo de dados int em vez do tipo de dados Number é mais rápido e eficiente.

Para valores inteiros fora do intervalo dos valores int mínimo e máximo, use o tipo de dados Number, que pode manipular valores entre positivo e negativo 9.007.199.254.740.992 (valores inteiros de 53 bits). O valor padrão para variáveis do tipo de dados int é 0.

Tipo de dados Null

O tipo de dados Null contém apenas um valor, `null`. É o valor padrão para o tipo de dados String e todas as classes que definem tipos de dados complexos, inclusive a classe Object. Nenhum outro tipo de dados primitivo, como Boolean, Number, int e uint, contém o valor `null`. No tempo de execução, o valor `null` é convertido para o valor padrão apropriado se tentar atribuir `null` às variáveis do tipo Boolean, Number, int ou uint. Você não pode usar esse tipo de dados como uma anotação de tipo.

Tipo de dados Number

No ActionScript 3.0, o tipo de dados Number pode representar inteiros, inteiros não assinados e números de ponto flutuante. Entretanto, para maximizar o desempenho, você deve usar o tipo de dados Number somente para valores inteiros maiores do que os tipos `int` e `uint`de 32 bits podem armazenar ou para números de ponto flutuante. Para armazenar um número de ponto flutuante, inclua um ponto decimal no número. Se você omitir um ponto decimal, o número é armazenado como um inteiro.

O tipo de dados Number usa o formato de precisão dupla de 64 bits conforme especificado pelo Padrão IEEE para Aritmética de Ponto Flutuante Binário (IEEE-754). Esse padrão determina como os números de ponto flutuante são armazenados usando os 64 bits disponíveis. Um bit é usado para designar se o número é positivo ou negativo. Onze bits são usados para o expoente, que é armazenado como base 2. Os 52 bits restantes são usados para armazenar o *significando* (também chamado de *mantissa*), que é o número elevado à potência indicada pelo expoente.

Com o uso de alguns bits para armazenar um expoente, o tipo de dados Number pode armazenar números de ponto flutuante significativamente maiores do que se usasse todos os bits para o significando. Por exemplo, se o tipo de dados Number usasse os 64 bits para armazenar o significando, ele armazenaria um número tão grande quanto $2^{65} - 1$. Com o uso de 11 bits para armazenar um expoente, o tipo de dados Number pode elevar seu significando à potência de 2^{1023}.

Os valores máximo e mínimo que o tipo Number pode representar são armazenados em propriedades estáticas da classe Number chamadas `Number.MAX_VALUE` e `Number.MIN_VALUE`.

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

Embora esse intervalo de números seja enorme, seu custo é a precisão. O tipo de dados Number usa 52 bits para armazenar o significando, por isso os números que exigem mais de 52 bits para fazer uma representação precisa, como a fração 1/3, são apenas aproximações. Se o seu aplicativo exibir precisão absoluta com números decimais, será necessário usar um software que implemente a aritmética de ponto flutuante decimal em vez da aritmética de ponto flutuante binário.

Durante o armazenamento de valore inteiros com o tipo de dados Number, somente os 52 bits do significando são usados. O tipo de dados Number usa esses 52 bits e um bit oculto especial para representar inteiros de -9.007.199.254.740.992 (-2^{53}) a 9.007.199.254.740.992 (2^{53}).

O valor `NaN` é usado não apenas como o valor padrão para variáveis do tipo `Number`, mas também como resultado de qualquer operação que deve retornar um número e não retorna. Por exemplo, se você tentar calcular a raiz quadrada de um número negativo, o resultado é `NaN`. Outros valores Number especiais incluem *infinito positivo* e *infinito negativo*.

Nota: *O resultado da divisão por `0` será apenas `NaN` se o divisor também for `0`. A divisão por `0` produzirá `infinity` quando o dividendo for positivo ou `-infinity` quando o dividendo for negativo.*

Tipo de dados String

O tipo de dados String representa uma seqüência de caracteres de 16 bits. Os Strings são armazenados internamente como caracteres Unicode, usando o formato UTF-16, Eles são valores imutáveis, assim como na linguagem de programação Java. Uma operação sobre um valor String retorna uma nova ocorrência da seqüência de caracteres. O valor padrão para uma variável declarada com o tipo de dados String é `null`. O valor `null` não é igual ao string vazio (`""`). O valor `null` significa que a variável não tem valor armazenado nela, enquanto a sequência vazia de caracteres indica que a variável tem um valor que é um String que não contém caracteres.

Tipo de dados uint

O tipo de dados int é armazenado internamente como um inteiro não assinado de 32 bits e inclui o conjunto de inteiros de 0 a 4.294.967.295 (2^{32} - 1), inclusive. Use o tipo de dados uint para circunstâncias especiais que exigem inteiros não negativos. Por exemplo, você deve usar o tipo de dados uint para representar os valores de cor de pixel, porque o tipo de dados int tem um bit de sinal interno que não é apropriado para manipular valores de cor. Para valores inteiros maiores do que o valor uint máximo, use o tipo de dados Number, que pode manipular valores inteiros de 53 bits. O valor padrão para variáveis do tipo de dados uint é 0.

Tipo de dados void

O tipo de dados void contém apenas um valor, `undefined`. Nas versões anteriores do ActionScript, `undefined` era o valor padrão para ocorrências da classe Object. No ActionScript 3.0, o valor padrão para ocorrências de Object é `null`. Se você tentar atribuir um valor `undefined` a uma ocorrência da classe Object, o valor é convertido para `null`. É possível atribuir apenas um valor de `undefined` a variáveis sem tipo. Variáveis sem tipo são aquelas que não possuem nenhuma anotação de tipo ou usam o símbolo asterisco (`*`) para a anotação de tipo. Você pode usar `void` apenas como uma anotação de tipo de retorno.

Tipo de dados Object

O tipo de dados Object é definido pela classe Object. A classe Object serve de classe base para todas as definições de classe no ActionScript. A versão do ActionScript 3.0 do tipo de dados Object difere das versões anteriores de três formas. Primeiro, o tipo de dados Object não é mais o tipo de dados padrão atribuído a variáveis sem nenhuma anotação de tipo. Em segundo lugar, o tipo de dados Object não inclui mais o valor `undefined`, que costumava ser o valor padrão das ocorrências Object. Em terceiro lugar, no ActionScript 3.0, o valor padrão para ocorrências da classe Object é `null`.

Nas versões anteriores do ActionScript, uma variável sem nenhuma anotação de tipo era automaticamente atribuída ao tipo de dados Object. Isso não acontece mais no ActionScript 3.0, que agora inclui a idéia de uma variável realmente sem tipo. As variáveis sem nenhuma anotação de tipo agora são consideradas sem tipo. Se preferir deixar mais claro para os leitores do código que sua intenção é deixar uma variável sem tipo, você pode usar o símbolo de asterisco (`*`) para a anotação de tipo, que é equivalente a omitir uma anotação de tipo. O exemplo a seguir mostra duas instruções equivalentes, que declaram uma variável sem tipo `x`:

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

Somente variáveis sem tipo podem manter o valor `undefined`. Se você tentar atribuir o valor `undefined` a uma variável que possui um tipo de dados, o tempo de execução converte o valor `undefined` no valor padrão desse tipo de dados. Para as ocorrências do tipo de dados Object, o valor padrão é `null`, que significa que, se você tentar atribuir `undefined` a uma ocorrência de Object, o valor é convertido para `null`.

Conversões de tipo

Uma conversão de tipo ocorre quando um valor é transformado em um valor de um tipo de dados diferente. As conversões de tipo podem ser *implícitas* ou *explícitas*. A conversão implícita, que também é chamada de *coerção*, é realizada às vezes em tempo de execução. Por exemplo, se o valor 2 for atribuído a uma variável do tipo de dados Boolean, o valor 2 é convertido para o valor booleano `true` antes de atribuir o valor à variável. A conversão explícita, também chamada de *projeção*, ocorrerá quando o código instruir o compilador a tratar uma variável de um tipo de dados como se pertencesse a um tipo de dados diferente. Quando os valores primitivos estão envolvidos, a projeção realmente converte os valores de um tipo de dados para outro. Para projetar um objeto em um tipo diferente, use parênteses para delimitar o nome do objeto e preceda-o com o nome do novo tipo. Por exemplo, o seguinte código usa um valor booleano e projeta-o em um inteiro:

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

Conversões implícitas

As conversões implícitas acontecem em tempo de execução em diversos contextos:

- Em instruções de atribuição
- Quando os valores são transmitidos como argumentos de função
- Quando os valores são retornados de funções
- Em expressões que usam determinados operadores, como o operador de adição (+)

  Para tipos definidos pelo usuário, as conversões implícitas são bem-sucedidas quando o valor a ser convertido é uma ocorrência da classe de destino ou de uma classe derivada dela. Se uma conversão implícita não for bem-sucedida, ocorrerá um erro. Por exemplo, o seguinte código contém uma conversão implícita bem-sucedida e outra malsucedida:

  ```
  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 os tipos primitivos, as conversões implícitas são tratadas chamando os mesmos algoritmos de conversão internos que são chamados pelas funções de conversão explícitas.

Conversões explícitas

É útil usar conversões explícitas, ou projeção, ao compilar no modo estrito, porque pode haver situações em que você não deseja que uma incompatibilidade de tipos gere um erro em tempo de compilação. Pode ser o caso de quando você sabe que a coerção converterá os valores corretamente em tempo de execução. Por exemplo, ao trabalhar com dados recebidos de um formulário, você pode querer contar com a coerção para converter determinados valores de seqüência de caracteres em valores numéricos. O código a seguir gera um erro em tempo de compilação, muito embora o código seja executado corretamente no modo padrão:

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

Se quiser continuar a usar o modo estrito, mas preferir converter a seqüência de caracteres em um inteiro, você pode usar a conversão explícita da seguinte forma:

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

Projeção para int, uint e Number

É possível projetar qualquer tipo de dados em um dos três tipos de números: int, uint e Number. Se o número não puder ser convertido por alguma razão, o valor padrão de 0 é atribuído para os tipos de dados int e uint, e o valor padrão de `NaN` é atribuído para o tipo de dados Number. Se você converter um valor Boolean em um número, `true` se tornará o valor 1 e `false` se tornará o 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
```

Os valores String que contêm apenas dígitos pode ser convertidos com êxito em um dos tipos de número. Os tipos de número também podem converter seqüências de caracteres que parecem números negativos ou que representam um valor hexadecimal (por exemplo, `0x1A`). O processo de conversão ignora os caracteres de espaço em branco à esquerda e à direita no valor da seqüência de caracteres. Também é possível projetar seqüências de caracteres que se parecem com números de ponto flutuante usando `Number()`. A inclusão de um ponto decimal faz com que `uint()` e `int()` retornem um inteiro, truncando o decimal e os caracteres seguintes. Por exemplo, os seguintes valores de seqüência de caracteres podem ser projetados em 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
```

Os valores String que contêm caracteres não numéricos retornarão 0 quando projetados com `int()` ou `uint()` e `NaN` quando projetados com `Number()`. O processo de conversão ignorará espaço em branco à esquerda e à direita, mas retornará 0 ou `NaN` se uma seqüência de caracteres tiver espaço em branco separando dois números.

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

No ActionScript 3.0, a função `Number()` não suporta mais números octais ou de base 8. Se você fornecer uma seqüência de caracteres com um zero à esquerda para a função `Number()` do ActionScript 2.0, o número será interpretado como um número octal e convertido em seu equivalente decimal. Isso não acontece com a função `Number()` no ActionScript 3.0, que em vez disso ignora o zero à esquerda. Por exemplo, o seguinte código gera uma saída diferente quando compilado usando versões diferentes do ActionScript:

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

A projeção não é necessária quando um valor de um tipo numérico é atribuído a uma variável de um tipo numérico diferente. Mesmo no modo estrito, os tipos numéricos são implicitamente convertidos em outros tipos numéricos. Isso significa que, em alguns casos, quando o intervalo de um tipo for excedido, o resultado poderá gerar valores inesperados. Os seguintes exemplos compilam no modo estrito, embora alguns gerem 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
```

A tabela a seguir resume os resultados da projeção para os tipos de dados Number, int ou uint a partir de outros tipos de dados.

Tipo de dados ou valor	Resultado da conversão em Number, int ou uint
Booleano	Se o valor for `true`, 1; caso contrário, 0.
Date	A representação interna do objeto Date, que é o número de milésimos de segundo desde a meia-noite de 1º de janeiro de 1970, hora universal.
`null`	0
Objeto	Se a ocorrência for `null` e convertida para Number, `NaN`; caso contrário, 0.
String	Um número se a sequência de caracteres puder ser convertido para número; do contrário, `NaN` se convertido para Number, ou 0, se convertido para int ou uint.
`undefined`	Se convertido em Number, `NaN`; se convertido em int ou uint, 0.

Projeção para Boolean

A projeção para Boolean a partir de qualquer tipo de dados numérico (uint, int e Number) resultará em `false` se o valor numérico for 0 e em `true` se não for. Para o tipo de dados Number, o valor `NaN` também resulta em `false`. O exemplo a seguir mostra os resultados da projeção dos números em -1, 0 e 1:

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

A saída do exemplo mostra que, dos três números, somente 0 retorna um valor `false`:

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

A projeção para Boolean de um valor String retornará `false` se a seqüência de caracteres for `null` ou vazia (`""`). Do contrário, retornará `null`.

```
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
```

A projeção para Boolean de uma ocorrência da classe Object retornará `false` se a ocorrência for `null`; do contrário, retornará `true`:

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

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

As variáveis Boolean obtém tratamento especial no modo estrito no que se refere a atribuir valores de qualquer tipo de dados a uma variável Boolean sem projeção. A coerção implícita de todos os tipos de dados para o tipo de dados Boolean ocorre mesmo no modo estrito. Em outras palavras, diferentemente de quase todos os outros tipos de dados, a projeção para Boolean não é necessária para evitar erros no modo estrito. Os seguintes exemplos compilam no modo estrito e se comportam conforme o esperado em tempo de execução:

```
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
```

A tabela a seguir resume os resultados da projeção para o tipo de dados Boolean a partir de outros tipos de dados:

Tipo de dados ou valor	Resultado da conversão em Boolean
String	`false` se o valor for `null` ou uma seqüência de caracteres vazia (`""`); caso contrário, `true`.
`null`	`false`
Number, int ou uint	`false` se o valor for `NaN` ou 0; caso contrário, `true`.
Objeto	`false` se a ocorrência for `null`; caso contrário, `true`.

Projeção para String

A projeção para o tipo de dados String de qualquer tipo de dados numérico retorna uma representação de seqüência de caracteres do número. A projeção para o tipo de dados String de um valor Boolean retornará a seqüência de caracteres `"true"` se o valor for `true` e retornará a seqüência de caracteres `"false"` se o valor for `false`.

A projeção para String de uma ocorrência da classe Object retornará a seqüência de caracteres `"null"` se a ocorrência for `null`. Caso contrário, a projeção para o tipo String da classe Object retornará a seqüência de caracteres `"[object Object]"` .

A projeção para String de uma ocorrência da classe Array retorna uma sequência de caracteres composta por uma lista delimitada por vírgula de todos os elementos da matriz. Por exemplo, a seguinte projeção para o tipo de dados String retorna uma seqüência de caracteres contendo os três elementos da matriz:

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

A projeção para String de uma ocorrência da classe Date retorna uma representação da seqüência de caracteres da data que a ocorrência contém. Por exemplo, o seguinte exemplo retorna uma representação da seqüência de caracteres da ocorrência da classe Date (a saída mostra o resultado para o Horário de Verão do Pacífico):

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

A tabela a seguir resume os resultados da projeção para o tipo de dados String a partir de outros tipos de dados:

Tipo de dados ou valor	Resultado da conversão em String
Matriz	Uma sequência de caracteres composta por todos os elementos da matriz.
Booleano	`"true"` ou `"false"`
Date	Uma representação da seqüência de caracteres do objeto Date.
`null`	`"null"`
Number, int ou uint	Uma representação da seqüência de caracteres do número.
Objeto	Se a ocorrência for null, `"null"`; caso contrário, `"[object Object]"`.

Sintaxe

A sintaxe de uma linguagem define um conjunto de regras que deve ser seguido durante a escrita de código executável.

Diferenciação entre maiúsculas e minúsculas

O ActionScript 3.0 é uma linguagem que diferencia maiúsculas e minúsculas. Os identificadores que diferem somente em maiúsculas e minúsculas são considerados identificadores diferentes. Por exemplo, o código a seguir cria duas variáveis diferentes:

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

Sintaxe de pontos

O operador dot (.) fornece uma maneira de acessar as propriedades e os métodos de um objeto. Com o uso da sintaxe de pontos, é possível fazer referência a uma propriedade ou um método de classe usando um nome de ocorrência, seguido do operador dot e do nome da propriedade ou do método. Por exemplo, considere a seguinte definição de classe:

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

Com o uso da sintaxe de pontos, é possível acessar a propriedade `prop1` e o método `method1()` usando o nome da instância criado no seguinte código:

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

É possível usar a sintaxe de pontos para definir pacotes. O operador dot é usado para fazer referência a pacotes aninhados. Por exemplo, a classe EventDispatcher reside em um pacote chamado eventos que é aninhado dentro do pacote chamado flash. Você pode fazer referência aos pacotes de eventos usando a seguinte expressão:

```
flash.events
```

Você também pode fazer referência à classe EventDispatcher usando esta expressão:

```
flash.events.EventDispatcher
```

Sintaxe de barras

A sintaxe de barras não é suportada no ActionScript 3.0. Ela foi usada nas versões anteriores do ActionScript para indicar o caminho de um clipe de filme ou variável.

Literais

Um *literal* é um valor que aparece diretamente em seu código. Os seguintes exemplos são de literais:

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

Os literais também podem ser agrupados para formar literais compostos. Os literais de matriz são colocados entre colchetes (`[]`) e usam a vírgula para separar elementos de matriz.

Um literal de matriz pode ser usado para inicializar uma matriz. Os exemplos a seguir mostram duas matrizes que são inicializadas usando literais de matriz. É possível usar a instrução `new` e transmitir o literal composto como um parâmetro para o construtor de classe Array, mas você também pode atribuir valores literais diretamente ao instanciar ocorrências das seguintes classes centrais do ActionScript: Object, Array, String, Number, int, uint, XML, XMLList e Boolean.

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

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

Os literais também podem ser usado para inicializar um objeto genérico. Um objeto genérico é uma ocorrência da classe Object. Os literais Object são colocados entre chaves (`{}`) e usam a vírgula para separar propriedades de objetos. Cada propriedade é declarada com o caractere dois-pontos (`:`), que separa o nome da propriedade do valor da propriedade.

É possível criar um objeto genérico usando a instrução `new` e transmitir o literal de objeto como um parâmetro para o construtor de classe Objeto ou atribuir o literal de objeto diretamente à ocorrência que você está declarando. O exemplo a seguir demonstra duas formas alternativas de criar um novo objeto genérico e inicializar o objeto com três propriedades (`propA`, `propB` e `propC`), cada uma com valores definidos como 1, 2, e 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};
```

Mais tópicos da Ajuda

Trabalho com sequências de caracteres

Uso de expressões regulares

Inicialização de variáveis XML

Ponto-e-vírgula

Você pode usar o caractere ponto-e-vírgula (`;`) para encerrar uma instrução. Opcionalmente, se omitir o caractere ponto-e-vírgula, o compilador presume que cada linha de código representa uma única instrução. Como muitos programadores estão acostumados a usar o ponto-e-vírgula para denotar o fim de uma instrução, seu código poderá ser mais fácil de ler se você usar consistentemente ponto-e-vírgula para encerrar as instruções.

O uso de um ponto-e-vírgula para encerrar uma instrução permite colocar mais de uma instrução em uma única linha, mas isso pode tornar o código mais difícil de ler.

Parênteses

Você pode usar parênteses (`()`) de três formas no ActionScript 3.0. Em primeiro lugar, você pode usar parênteses para alterar a ordem das operações em uma expressão. As operações que são agrupadas dentro de parênteses são sempre executadas primeiro. Por exemplo, os parênteses são usados para alterar a ordem das operações no seguinte código:

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

Em segundo lugar, você pode usar parênteses com o operador vírgula (`,`) para avaliar uma série de expressões e retornar o resultado da expressão final, como mostra o seguinte exemplo:

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

Em terceiro lugar, você pode usar parênteses para transmitir um ou mais parâmetros para funções ou métodos, como mostra o exemplo a seguir, que transmite um valor String para a função `trace()`:

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

Comentários

O código do ActionScript 3.0 oferece suporte a dois tipos de comentários: comentários de uma única linha e de várias linhas. Esses mecanismos de comentários são semelhantes aos do C++ e Java. O compilador ignora o texto marcado com um comentário.

Os comentários de uma única linha começam com dois caracteres de barra inclinada (`//`) e continuam até o fim da linha. Por exemplo, o seguinte código contém um comentário de uma única linha:

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

Os comentários de várias linhas começam com uma barra inclinada e um asterisco (`/*`) e terminam com um asterisco e uma barra inclinada (`*/`).

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

Palavras-chave e palavras reservadas

As *palavras reservadas* são palavras que não podem ser usadas como identificadores no código porque são para uso do ActionScript. Elas incluem *palavras-chave léxicas*, que são removidas do espaço para nomes do programa pelo compilador. O compilador relata um erro se você usar uma palavra-chave léxica como identificador. A seguinte tabela lista as palavras-chave léxicas do ActionScript 3.0.

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

Há um pequeno conjunto de palavras-chave, chamado *palavras-chave sintáticas*, que pode ser usado como identificador, mas têm um significado especial em determinados contextos. A tabela a seguir lista as palavras-chave sintáticas do ActionScript 3.0.

each	get	set	namespace
incluir	dinâmicos	final	native
override	estáticos		

Também há vários identificadores que, às vezes, são referidos como *palavras reservadas futuras*. Eles não são reservados pelo ActionScript 3.0, embora alguns sejam tratados como palavras-chave pelo software que incorpora o ActionScript 3.0. Você pode usar vários desses identificadores no seu código, mas a Adobe não recomenda essa prática porque eles podem aparecer como palavras-chave em uma versão subseqüente da linguagem.

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

long	prototype	short	synchronized
throws	to	transient	tipo
virtual	volatile		

Constantes

O ActionScript 3.0 oferece suporte à instrução `const`, que você pode usar para criar constantes. As constantes são propriedades com um valor fixo que não pode ser alterado. Você pode atribuir um valor a uma constante apenas uma vez, e a atribuição deve ocorrer próxima à declaração da constante. Por exemplo, se uma constante for declarada como um membro de uma classe, você poderá atribuir-lhe um valor somente como parte da declaração ou dentro do construtor de classe.

O seguinte código declara duas constantes. A primeira, `MINIMUM`, tem um valor atribuído como parte da instrução de declaração. A segunda, `MAXIMUM`, tem um valor atribuído no construtor. Observe que este exemplo é compilado apenas no modo padrão porque o modo estrito só permite que um valor de constante seja atribuído em tempo de inicialização.

```
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
```

Um erro será gerado se você tentar atribuir um valor inicial a uma constante de qualquer outra forma. Por exemplo, se você tentar definir o valor inicial de `MAXIMUM` fora da classe, ocorre um erro de tempo de execução.

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

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

O ActionScript 3.0 define uma ampla gama de constantes para sua conveniência. Por convenção, as constantes do ActionScript usam tudo em letras maiúsculas, com as palavras separadas pelo caractere de sublinhado (_). Por exemplo, a definição de classe MouseEvent usa essa convenção de nomenclatura para suas constantes, cada uma representando um evento relacionado à entrada do mouse:

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

Operadores

Os operadores são funções especiais que usam um ou mais operandos e retornam um valor. Um *operando* é um valor, normalmente um literal, uma variável ou uma expressão, que um operador usa como entrada. Por exemplo, no código a seguir, os operadores de adição (`+`) e de multiplicação (`*`) são usados com três operandos literais (2, 3, e 4) para retornar um valor. Esse valor é usado pelo operador de atribuição (`=`) para atribuir o valor retornado, 14, para a variável `sumNumber`.

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

Os operadores podem ser unários, binários ou ternários. Um operador *unário* usa um operando. Por exemplo, o operador de incremento (`++`) é um operador unário, porque usa apenas um operando. Um operador *binário* usa dois operandos. Por exemplo, o operador de divisão (`/`) usa dois operandos. Um operador *ternário* usa três operandos. Por exemplo, o operador condicional (`/`) usa três operandos.

Alguns operadores são *sobrecarregados*, o que significa que se comportam de forma diferente dependendo do tipo ou da quantidade de operandos que lhes são transmitidos. O operador de adição (`+`) é um exemplo de um operador sobrecarregado que se comporta de maneira diferente dependendo do tipo de dados dos operandos. Se os dois operandos forem números, o operador de adição retornará a soma dos valores. Se os dois operandos forem seqüências de caracteres, o operador de adição retornará a concatenação dos dois operandos. O código de exemplo a seguir mostra como o operador se comporta de forma diferente dependendo dos operandos:

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

Os operadores também podem se comportar de forma diferente com base no número de operandos fornecidos. O operador de subtração (`-`) é um operador unário e binário. Quando fornecido com apenas um operando, o operador de subtração nega o operando e retorna o resultado. Quando fornecido com dois operandos, o operador de subtração retorna a diferença entre os operandos. O exemplo a seguir mostra o operador de subtração usado primeiro como um operador unário e depois como binário.

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

Precedência e associatividade de operadores

A precedência e a associatividade de operadores determina a ordem na qual os operadores são processados. Embora possa parecer natural aos que estão familiarizados com aritmética que o compilador processe o operador de multiplicação (`*`) antes do operador de adição (`+`), o compilador precisa de instruções explícitas sobre os operadores que deve processar primeiro. Essas instruções são denominadas coletivamente de *precedência de operador*. O ActionScript define a precedência de um operador padrão que você pode alterar usando o operador parênteses (`()`). Por exemplo, o seguinte código altera a precedência padrão no exemplo anterior para forçar o compilador a processar o operador de adição antes do operador de multiplicação:

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

Você pode encontrar situações em que dois ou mais operadores da mesma precedência aparecem na mesma expressão. Nesses casos, o compilador usa as regras de *associatividade* para determinar qual operador será processado primeiro. Todos os operadores binários, exceto os de atribuição, são *associativos à esquerda*, o que significa que os operadores à esquerda são processados antes dos que estão à direita. Todos os operadores de atribuição e o operador condicional (`?:`) são *associativos à direita*, o que significa que os operadores à direita são processados antes dos que estão à esquerda.

Por exemplo, considere os operadores "menor do que" (`<`) e "maior do que" (`>`), que têm a mesma precedência. Se os dois forem usados na mesma expressão, o operador à esquerda será processado primeiro porque os dois operadores são associativos à esquerda. Isso significa que as duas instruções a seguir produzem a mesma saída:

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

O operador maior do que é processado primeiro, o que resulta em um valor `true`, porque o operando 3 é maior do que o operando 2. O valor `true` é transmitido para o operador menor do que junto com o operando 1. O seguinte código representa esse estado intermediário:

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

O operador menor do que converte o valor `true` no valor numérico 1 e compara esse valor numérico com o segundo operando 1 para retornar o valor `false` (o valor 1 não é menor que 1).

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

É possível alterar a associatividade à esquerda padrão com o operador parênteses. Você pode instruir o compilador a processar o operador menor do que primeiro, colocando esse operador e seus operandos entre parênteses. O exemplo a seguir usa o operador parênteses para produzir uma saída diferente usando os mesmos números que o exemplo anterior:

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

O operador menor do que é processado primeiro, o que resulta em um valor `false`, porque o operando 2 não é menor que o operando 1. O valor `false` é transmitido para o operador maior do que junto com o operando 3. O seguinte código representa esse estado intermediário:

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

O operador maior do que converte o valor `false` no valor numérico 0 e compara esse valor numérico com o outro operando 3 para retornar o valor `true` (o valor 3 é maior que 0).

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

A tabela a seguir lista os operadores para o ActionScript 3.0 em ordem decrescente de precedência. Cada linha da tabela contém operadores de mesma precedência. Cada linha de operadores tem precedência sobre a linha que aparece abaixo dela na tabela.

Grupo	Operadores
Primário	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Sufixo	`x++ x--`
Unário	`++x --x + - ~ ! delete typeof void`
Multiplicativo	`* / %`
Aditivo	`+ -`
Desvio em nível de bits	`<< >> >>>`
Relacional	`< > <= >= as in instanceof is`
Igualdade	`== != === !==`
AND em nível de bits	`&`
XOR em nível de bits	`^`
OR em nível de bits	`|`
AND lógico	`&&`
OR lógico	`||`
Condicional	`?:`
Atribuição	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Vírgula	`,`

Operadores primários

Os operadores primários incluem aqueles usados para criar literais Array e Object, agrupar expressões, chamar funções, instanciar ocorrências de classes e acessar propriedades.

Todos os operadores primários, conforme listados na tabela a seguir, têm a mesma precedência. Os operadores que fazem parte da especificação E4X são indicados pela notação (E4X).

Operador	Operação executada
`[]`	Inicializa uma matriz
`{x:y}`	Inicializa um objeto
`()`	Agrupa expressões
`f(x)`	Chama uma função
`new`	Chama um construtor
`x.y x[y]`	Acessa uma propriedade
`<></>`	Inicializa um objeto XMLList (E4X)
`@`	Acessa um atributo (E4X)
`::`	Qualifica um nome (E4X)
`..`	Acessa um elemento XML descendente (E4X)

Operadores de sufixo

Os operadores de sufixo usam um operador e incrementam ou decrementam o valor. Embora esses operadores sejam unários, eles são classificados separadamente do resto dos operadores unários por causa de sua maior precedência e seu comportamento especial. Quando um operador de sufixo é usado como parte de uma expressão maior, o valor da expressão é retornado antes que o operador de sufixo seja processado. Por exemplo, o seguinte código mostra como o valor da expressão `xNum++` é retornado antes de ser incrementado:

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

Todos os operadores de sufixo, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
`++`	Incrementos (sufixo)
`--`	Decrementos (sufixo)

Operadores unários

Os operadores unários usam um operando. Os operadores de incremento (`++`) e de decremento (`--`) deste grupo são *operadores deprefixo*, o que significa que aparecem antes do operando em uma expressão. Os operadores de prefixo diferem dos de sufixo na operação de incremento ou decremento que é executada antes que o valor da expressão geral seja retornado. Por exemplo, o seguinte código mostra como o valor da expressão `++xNum` é retornado depois de ser incrementado:

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

Todos os operadores unários, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
`++`	Incrementos (prefixo)
`--`	Decrementos (prefixo)
`+`	Unário +
`-`	Unário - (negativa)
`!`	NOT lógico
`~`	NOT em nível de bits
`delete`	Exclui uma propriedade
`typeof`	Retorna informações de tipo
`void`	Retorna um valor indefinido

Operadores multiplicativos

Os operadores multiplicativos usam dois operandos e executam cálculos de multiplicação, divisão ou módulo.

Todos os operadores multiplicativos, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
*	Multiplicação
/	Divisão
%	Módulo

Operadores aditivos

Os operadores aditivos usam dois operandos e executam cálculos de adição ou subtração. Todos os operadores aditivos, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
+	Adição
-	Subtração

Operadores de desvio em nível de bits

Os operadores de desvio em nível de bits usam dois operandos e desviam os bits do primeiro até o ponto especificado pelo segundo operando. Todos os operadores de desvio em nível de bits, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
<<	Desvio à esquerda em nível de bits
>>	Desvio à direita em nível de bits
>>>	Desvio à direita não assinado em nível de bits

Operadores relacionais

Os operadores relacionais usam dois operandos, comparam seus valores e retornam um valor booleano. Todos os operadores relacionais, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
<	Menor do que
>	Maior do que
<=	Menor do que ou igual a
>=	Maior do que ou igual a
as	Verifica tipo de dados
in	Verifica propriedades de objeto
instanceof	Verifica cadeia de protótipos
is	Verifica tipo de dados

Operadores de igualdade

Os operadores de igualdade usam dois operandos, comparam seus valores e retornam um valor booleano. Todos os operadores de igualdade, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
`==`	Igualdade
`!=`	Desigualdade
`===`	Igualdade estrita
`!==`	Desigualdade estrita

Operadores lógicos em nível de bits

Os operadores lógicos em nível de bits usam dois operandos e executam operações lógicas em nível de bits. Os operadores lógicos em nível de bits diferem na precedência e são listados na tabela a seguir em ordem decrescente de precedência:

Operador	Operação executada
`&`	AND em nível de bits
`^`	XOR em nível de bits
`|`	OR em nível de bits

Operadores lógicos

Os operadores lógicos usam dois operandos e retornam um valor booleano. Os operadores lógicos diferem em precedência e são listados na tabela a seguir em ordem decrescente de precedência:

Operador	Operação executada
`&&`	AND lógico
`||`	OR lógico

Operador condicional

O operador condicional é um operador ternário, o que significa que usa três operandos. Ele é um método útil para aplicar a instrução condicional `if..else .`

Operador	Operação executada
`?:`	Condicional

Operadores de atribuição

Os operadores de atribuição usam dois operandos e atribuem um valor a um deles, com base no valor do outro. Todos os operadores de atribuição, conforme listados na tabela a seguir, têm a mesma precedência:

Operador	Operação executada
=	Atribuição
*=	Atribuição de multiplicação
/=	Atribuição de divisão
%=	Atribuição de módulo
+=	Atribuição de adição
-=	Atribuição de subtração
<<=	Atribuição de desvio à esquerda em nível de bits
>>=	Atribuição de desvio à direita em nível de bits
>>>=	Atribuição de desvio à direita não assinado em nível de bits
&=	Atribuição AND em nível de bits
^=	Atribuição XOR em nível de bits
|=	Atribuição OR em nível de bits

Condicionais

O ActionScript 3.0 fornece três instruções condicionais básicas que você pode usar para controlar o fluxo de programa.

if..else

A instrução condicional `if..else` permitirá testar uma condição e executar um bloco de código se essa condição existir ou executar um bloco de código alternativo se ela não existir. Por exemplo, o seguinte código testa se o valor `x` excede 20, gera uma função `trace()` em caso afirmativo ou gera uma função `trace()` diferente em caso negativo:

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

Se não quiser executar um bloco de código alternativo, você poderá usar a instrução `if` sem a instrução `else`.

if..else if

É possível testar mais de uma condição usando a instrução condicional `if..else if`. Por exemplo, o código a seguir não apenas testa se o valor `x` excede 20, mas também se o valor `x` é negativo:

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

Se uma instrução `if` ou `else` for seguida de apenas uma instrução, a instrução não precisa ficar entre chaves. Por exemplo, o código a seguir não usa chaves:

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

Entretanto, a Adobe recomenda que você use chaves, porque poderá ocorrer um comportamento inesperado se instruções forem adicionadas posteriormente a uma instrução condicional sem chaves. Por exemplo, no código a seguir, o valor `positiveNums` é aumentado em 1 quer a condição seja ou não avaliada como `true`:

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

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

trace(positiveNums); // 1
```

switch

A instrução `switch` será útil se você tiver vários caminhos de execução que dependam da mesma expressão de condição. Ela fornece uma funcionalidade semelhante a uma longa série de instruções `if..else if`, mas é mais fácil de ler. Em vez de testar uma condição quanto a um valor booleano, a instrução `switch` avalia uma expressão e usa o resultado para determinar qual bloco de código será executado. Os blocos de código começam com uma instrução `case` e terminam com uma instrução `break`. Por exemplo, a seguinte instrução `switch` imprime o dia da semana, com base no número de dias retornado pelo 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;
}
```

Repetição

As instruções de repetição permitem executar um bloco específico de código repetidamente usando uma série de valores ou variáveis. A Adobe recomenda que o bloco de código seja sempre colocado entre chaves ({}). Embora você possa omitir as chaves caso o bloco de código contenha apenas uma instrução, essa prática não é recomendada pelo mesmo motivo que para as condicionais: ela aumenta as chances de que as instruções adicionadas em um momento posterior sejam excluídas inadvertidamente do bloco de código. Se, mais tarde, você adicionar uma instrução que deseja incluir no bloco de código, mas esquecer de adicionar as chaves necessárias, a instrução não é executada como parte da repetição.

for

A repetição `for` permite fazer a iteração por meio de uma variável para um intervalo específico de valores. Você deve fornecer três expressões em uma instrução `for`: uma variável que é definida com um valor inicial, uma instrução condicional que determina quando a repetição termina e uma expressão que altera o valor da variável a cada repetição. Por exemplo, o código a seguir é repetido cinco vezes. O valor da variável `i` começa com 0 e termina com 4, e a saída são os números de 0 a 4, cada um em sua própria linha.

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

for..in

A instrução `for..in` itera por meio das propriedades de um objeto ou dos elementos de uma matriz. Por exemplo, você pode usar uma repetição `for..in` para iterar por meio das propriedades de um objeto genérico (as propriedades do objeto não são mantidas em uma ordem específica, por isso elas podem aparecer em uma ordem aparentemente aleatória):

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

Também é possível iterar por meio dos elementos de uma matriz:

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

O que você não pode fazer é fazer a iteração pelas propriedades de um objeto se ele for uma ocorrência de uma classe selada (incluindo classes incorporadas e classes definidas pelo usuário). Você só pode fazer a iteração pelas propriedades de uma classe dinâmica. Mesmo com as ocorrências de classes dinâmicas, você só pode fazer iterações pelas propriedades que são acrescentadas dinamicamente.

for each..in

A instrução `for each..in` itera por meio dos itens de um conjunto, que podem ser tags em um objeto XML ou XMLList, os valores mantidos pelas propriedades do objeto ou os elementos de uma matriz. Por exemplo, como mostra o trecho a seguir, você pode usar uma repetição `for each..in` para iterar por meio das propriedades de um objeto genérico, mas diferentemente da repetição `for..in`, a variável do iterador em uma repetição `for each..in` contém o valor mantido pela propriedade em vez do nome da propriedade:

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

Você pode iterar por meio de um objeto XML ou XMLList, como mostra o seguinte exemplo:

```
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
*/
```

Também é possível iterar por meio dos elementos de uma matriz, como mostra este exemplo:

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

Você não poderá iterar por meio das propriedades de um objeto se o objeto for uma ocorrência de uma classe selada. Mesmo para ocorrências de classes dinâmicas, não é possível iterar por meio de uma propriedade fixa, que é a propriedade especificada como parte da definição de classe.

while

A repetição `while` é como uma instrução `if` que é repetida desde que a condição seja `true`. Por exemplo, o código a seguir produz a mesma saída que o exemplo da repetição `for`:

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

Uma desvantagem do uso de uma repetição `while` em vez de `for` é que repetições infinitas são mais fáceis de escrever com repetições `while`. O exemplo da repetição `for` não será compilado se você omitir a expressão que incrementa a variável do contador, mas o exemplo da repetição `while` será compilado se essa etapa for omitida. Sem a expressão que incrementa `i`, a repetição se torna infinita.

do..while

A repetição `do..while` é uma repetição `while` que garante que o bloco de código seja executado pelo menos uma vez, porque a condição é verificada depois que o bloco é executado. O código a seguir mostra um exemplo simples de uma repetição `do..while` que gera saída mesmo que a condição não seja atendida:

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

Funções

As *funções* são blocos de código que executam tarefas específicas e podem ser reutilizados no seu programa. Há dois tipos de funções no ActionScript 3.0: *métodos* e *fechamentos de função*. O fato de uma função ser uma chamada a um método ou um fechamento de função depende do contexto na qual ela é definida. Uma função é chamada de método quando é especificada como parte de uma definição de classe ou anexada a uma ocorrência de um objeto. Uma função é chamada de fechamento de função quando é definida de qualquer outra forma.

As funções sempre foram extremamente importantes no ActionScript. No ActionScript 1.0, por exemplo, a palavra-chave `class` não existir, por isso as "classes" eram definidas pelas funções do construtor. Embora a palavra-chave `class` tenha sido adicionada à linguagem, ainda é importante ter um bom entendimento das funções para aproveitar todas as vantagens que ela tem a oferecer. Isso pode ser um desafio para os programadores que esperam que as funções do ActionScript se comportem da mesma forma que as funções em linguagens como C++ ou Java. Embora a definição e a chamada de funções básicas não apresentem um desafio aos programadores experientes, alguns recursos mais avançados das funções do ActionScript exigem uma explicação.

Conceitos de funções básicas

Chamada de funções

Uma função é chamada usando seu identificador seguido de um operador parênteses (`()`). O operador parênteses delimita qualquer parâmetro de função que você deseja enviar para a função. Por exemplo, a função `trace()` é uma função de nível superior no ActionScript 3.0:

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

Se estiver chamando uma função sem nenhum parâmetro, você deverá usar parênteses vazios. Por exemplo, você pode usar o método `Math.random()`, que não usa nenhum parâmetro, para gerar um número aleatório:

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

Definição de suas próprias funções

Há duas maneiras de definir uma função no ActionScript 3.0: você pode usar uma instrução de função ou uma expressão de função. A escolha da técnica depende de sua preferência por um estilo de programação mais estático ou dinâmico. Defina suas funções com instruções de função se preferir uma programação de modo estático ou estrito. Defina as funções com expressões de função se tiver uma necessidade específica para isso. As expressões de função são usadas com mais freqüência em programação de modo dinâmico ou padrão.

Instruções de função

A instrução de função é a técnica preferida para definir funções no modo estrito. Uma instrução de função começa com a palavra-chave `function` e, em seguida, vem:

- O nome da função

- Os parâmetros, em uma lista delimitada por vírgula e entre parênteses
- O corpo da função, ou seja, o código do ActionScript a ser executado quando a função é chamada, delimitado por chaves

Por exemplo, o código a seguir cria uma função que define um parâmetro e chama a função usando a sequência de caracteres "`hello`" como valor do parâmetro:

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

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

Expressões de função

A segunda forma de declarar uma função é usar uma instrução de atribuição com uma expressão de função, que às vezes também é chamada de literal de função ou função anônima. Este método é mais detalhado e amplamente usado nas versões anteriores do ActionScript.

Uma instrução de atribuição com uma expressão de função começa com a palavra-chave `var` e, em seguida, vem:

- O nome da função
- O operador dois-pontos (`:`)
- A classe `Function` para indicar o tipo de dados
- O operador de atribuição (`=`)
- A palavra-chave `function`
- Os parâmetros, em uma lista delimitada por vírgula e entre parênteses
- O corpo da função, ou seja, o código do ActionScript a ser executado quando a função é chamada, delimitado por chaves

  Por exemplo, o seguinte código declara a função `traceParameter` usando uma expressão de função:

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

  Observe que um nome de função não é especificado da mesma forma que em uma instrução de função. Outra diferença importante entre a expressão de função e a instrução de função é que a primeira é uma expressão e não uma instrução. isso significa que uma expressão de função não é suficiente como uma instrução de função. Ela só pode ser usada como parte de uma instrução, em geral, de atribuição. O exemplo a seguir mostra uma expressão de função atribuída a um elemento de matriz:

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

Escolha entre instruções e expressões

Como regra geral, use uma instrução de função a menos que circunstâncias específicas exijam o uso de uma expressão. As instruções de função são menos detalhadas e fornecem uma experiência mais consistente entre o modo estrito e o padrão do que as expressões de função.

As instruções de função são mais fáceis de ler do que as instruções de atribuição que contêm expressões de função. As instruções de função tornam o código mais conciso; elas são menos confusas do que as expressões de função, que requerem o uso das palavras-chave `var` e `function`.

As instruções de função fornecem uma experiência mais consistente entre os dois modos de compilação já que é possível usar a sintaxe de pontos nos modos estrito e padrão para chamar um método declarado usando uma instrução de função. Isso não é necessariamente verdadeiro para métodos declarados com uma expressão de função. Por exemplo, o código a seguir define uma classe chamada Example com dois métodos: `methodExpression()`, que é declarado com uma expressão de função, e `methodStatement()`, que é declarado com uma instrução de função. No modo estrito, não é possível usar a sintaxe de pontos para chamar o 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
```

As expressões de função são consideradas mais adequadas para a programação direcionada ao comportamento de tempo de execução ou dinâmico. Se preferir usar o modo estrito, mas também precisar chamar um método declarado com uma expressão de função, você poderá usar qualquer uma destas duas técnicas. Em primeiro lugar, você pode chamar o método usando colchetes (`[]`) em vez do operador ponto (`.`). A chamada de método a seguir é bem-sucedida nos modos estrito e padrão:

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

Em segundo lugar, você pode declarar a classe inteira como uma classe dinâmica. Embora isso permita chamar o método usando o operador dot, o lado negativo é que você sacrifica algumas funcionalidades do modo estrito para todas as ocorrências dessa classe. Por exemplo, o compilador não irá gerar um erro se você tentar acessar uma propriedade indefinida em uma ocorrência de uma classe dinâmica.

Há algumas circunstâncias nas quais as expressões de função são úteis. Um uso comum de expressões de função é para funções que são usadas uma única vez e, depois, são descartadas. Outro uso menos comum é para anexar uma função a uma propriedade de protótipo. Para obter mais informações, consulte Objeto de protótipo.

Há duas diferenças sutis entre as instruções de função e as expressões de função que devem ser levadas em conta ao escolher a técnica usada. A primeira diferença é que as expressões de função não existem independentemente como objetos em relação ao gerenciamento de memória e à coleta de lixo. Em outras palavras, durante a atribuição de uma expressão de função a outro objeto, como um elemento de matriz ou uma propriedade de objeto, você cria a única referência a essa expressão de função no código. Se a matriz ou o objeto ao qual a expressão de função é anexada sair do escopo ou não estiver disponível, não é mais possível acessar a expressão de função. Se a matriz ou o objeto forem excluídos, a memória usada pela expressão de função se torna qualificada para a coleta de lixo, o que significa que a memória é qualificada para ser reivindicada e reutilizada para outros fins.

O exemplo a seguir mostra que, para uma expressão de função, assim que a propriedade com a expressão atribuída for excluída, a função não ficará mais disponível. A classe Test é dinâmica, o que significa que é possível adicionar uma propriedade chamada `functionExp` que mantém uma expressão de função. A função `functionExp()` pode ser chamada com o operador dot, mas assim que a propriedade `functionExp` for excluída, a função ficará inacessível.

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

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

Se, no entanto, for definida com uma instrução de função, a função existirá como seu próprio objeto e continuará a existir mesmo depois que a propriedade à qual está anexada for excluída. O operador `delete` funciona apenas em propriedades de objetos, por isso até mesmo uma chamada para excluir a função `stateFunc()` em si não 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
```

A segunda diferença entre as instruções de função e as expressões de função é que as primeiras existem em todo o escopo no qual são definidas, incluindo em instruções que aparecem antes da instrução de função. As expressões de função, porém, são definidas somente para instruções subseqüentes. Por exemplo, o seguinte código chama com êxito a função `scopeTest()` antes que ela seja definida:

```
statementTest(); // statementTest

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

As expressões de função não estão disponíveis antes de serem definidas, por isso o seguinte código resulta em um erro de tempo de execução:

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

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

Retorno de valores de funções

Para retornar um valor de sua função, use a instrução `return` seguida pela expressão ou pelo valor literal que deseja retornar. Por exemplo, o seguinte código retorna uma expressão representando o parâmetro:

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

Observe que a instrução `return` encerra a função, de forma que as instruções abaixo de uma instrução `return` não são executadas, como a seguir:

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

No modo estrito, você deve retornar um valor do tipo apropriado se decidir especificar um tipo de retorno. Por exemplo, o código a seguir gera um erro no modo estrito, porque não retorna um valor válido:

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

Funções aninhadas

É possível aninhar funções, o que significa que as funções podem ser declaradas dentro de outras. Uma função aninhada está disponível apenas dentro da função pai, a menos que uma referência a ela seja transmitida ao código externo. Por exemplo, o seguinte código declara duas funções aninhadas dentro da função `getNameAndVersion()`:

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

Quando funções aninhadas são transmitidas ao código externo, elas são transmitidas como fechamentos de função, o que significa que a função retém as definições que estão no escopo quando a função é definida. Para obter mais informações, consulte Escopo de funções.

Parâmetros de função

O ActionScript 3.0 fornece algumas funcionalidades para parâmetros de função que podem parecer novas aos programadores que não conhecem a linguagem. Embora a idéia de transmitir parâmetros por valor ou referência não deva ser nova para a maioria dos programadores, o objeto `arguments` e o parâmetro ... (rest) talvez sejam uma novidade para muitos.

Transmissão de argumentos por valor ou referência

Em muitas linguagens de programação, é importante entender a distinção entre transmitir argumentos por valor ou por referência, pois ela pode afetar a forma como o código é criado.

Ser transmitido por valor significa que o valor do argumento é copiado em uma variável local para uso dentro da função. Ser transmitido por referência significa que apenas uma referência ao argumento é transmitido, em vez do valor real. Não é feita nenhuma cópia do argumento real. Em vez disso, uma referência à variável transmitida como um argumento é criada e atribuída a uma variável local para uso dentro da função. Como uma referência a uma variável fora da função, a variável local fornece a capacidade de alterar o valor da variável original.

No ActionScript 3.0, todos os argumentos são transmitidos por referência, porque todos os valores são armazenados como objetos. Entretanto, os objetos que pertencem aos tipos de dados primitivos, que incluem Boolean, Number, int, uint e String, têm operadores especiais que fazem com que se comportem como se fossem transmitidos por valor. Por exemplo, o seguinte código cria uma função chamada `passPrimitives()` que define dois parâmetros chamados `xParam` e `yParam`, ambos do tipo int. Esses parâmetros são semelhantes às variáveis locais declaradas no corpo da função `passPrimitives()`. Quando a função for chamada com os argumentos `xValue` e `yValue`, os parâmetros `xParam` e `yParam` serão inicializados com referências aos objetos int representados por `xValue` e `yValue`. Como são primitivos, os argumentos se comportam como se fossem transmitidos por valor. Embora `xParam` e `yParam` inicialmente contenham apenas referências aos objetos `xValue` e `yValue`, quaisquer alterações às variáveis dentro do corpo da função gera novas cópias dos valores na memória.

```
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
```

Dentro da função `passPrimitives()`, os valores de `xParam` e `yParam` são incrementados, mas isso não afeta os valores de `xValue` e `yValue`, como mostra a última instrução `trace`. Isso seria válido mesmo que os parâmetros fossem nomeados de forma idêntica às variáveis, `xValue` e `yValue`, porque `xValue` e `yValue` dentro da função apontariam para novos locais na memória que existem separadamente das variáveis de mesmo nome fora da função.

Todos os outros objetos, ou seja, objetos que não pertencem aos tipos de dados primitivos, são transmitidos por referência, o que fornece a capacidade de alterar o valor das variáveis originais. Por exemplo, o código a seguir cria um objeto chamado `objVar` com duas propriedades, `x` e `y`. O objeto é transmitido como um argumento para a função `passByRef()`. Como não é de um tipo primitivo, o objeto não é apenas transmitido por referência, mas também permanece como uma. Isso significa que as alterações feitas aos parâmetros dentro da função afetam as propriedades do objeto fora da função.

```
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
```

O parâmetro `objParam` referencia o mesmo objeto que a variável global `objVar`. Como você pode ver nas instruções `trace` do exemplo, as alterações nas propriedades `x` e `y` do objeto `objParam` são refletidas no objeto `objVar`.

Valores de parâmetro padrão

No ActionScript 3.0, você pode declarar os *valores do parâmetro padrão* para uma função. Se uma chamada a uma função com valores de parâmetro padrão omitir um parâmetro com valores padrão, será usado o valor especificado na definição de função para esse parâmetro. Todos os parâmetros com valores padrão devem ser colocados no final da lista de parâmetros. Os valores atribuídos como padrão devem ser constantes de tempo de compilação. A existência de um valor padrão para um parâmetro efetivamente o torna um *parâmetro opcional*. Um parâmetro sem um valor padrão é considerado um *parâmetro necessário*.

Por exemplo, o seguinte código cria uma função com três parâmetros, dois dos quais têm valores padrão. Quando a função é chamada com apenas um parâmetro, os valores padrão para os parâmetros são usados.

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

O objeto arguments

Quando os parâmetros forem transmitidos a uma função, será possível usar o objeto `arguments` para acessar informações sobre esses parâmetros. Alguns aspectos importantes do objeto `arguments` incluem o seguinte:

- O objeto `arguments` é uma matriz que inclui todos os parâmetros transmitidos à função.
- A propriedade `arguments.length` relata o número de argumentos transmitidos à função.
- A propriedade `arguments.callee` fornece uma referência à função em si, que é útil para chamadas recursivas às expressões de função.

  Nota: *O objeto `arguments` não estará disponível se qualquer parâmetro for chamado de `arguments` ou se for usado o parâmetro ... (rest).*

  Se o objeto `arguments` estiver referenciado no corpo de uma função, o ActionScript 3.0 permite que as chamadas de função incluam mais parâmetros do que os especificados na definição de função, mas gera um erro no compilador no modo restrito, se o número de parâmetros não corresponder ao número de parâmetros necessários (e, opcionalmente, quaisquer parâmetros opcionais). É possível usar o aspecto de matriz do objeto `arguments` para acessar qualquer parâmetro transmitido à função, independentemente de ele ser especificado na definição de função. O exemplo a seguir, que é compilado apenas no modo padrão, usa a matriz `arguments` com a propriedade `arguments.length` para rastrear todos os parâmetros transmitidos para a função `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
  ```

A propriedade `arguments.callee`, em geral, é usada em funções anônimas para criar recursão. Você pode usá-la para adicionar flexibilidade ao seu código. Se o nome de uma função recursiva for alterado durante o ciclo de desenvolvimento, não será necessário se preocupar em alterar a chamada recursiva no corpo da função se usar `arguments.callee` em vez do nome de função. A propriedade `arguments.callee` é usada na seguinte expressão de função para permitir recursão:

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

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

Caso você use o parâmetro ... (rest) na declaração de função, o objeto `arguments` não fica disponível. Em vez disso, você terá de acessar os parâmetros usando os nomes de parâmetro declarados para eles.

Você também deve ter o cuidado de evitar o uso da seqüência de caracteres `"arguments"` como parâmetro, porque ela obscurece o objeto `arguments`. Por exemplo, se a função `traceArgArray()` for reescrita de forma que um parâmetro `arguments` seja adicionado, as referências a `arguments` no corpo da função irão se referir ao parâmetro e não ao objeto `arguments`. O seguinte código não produz nenhuma saída:

```
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
```

O objeto `arguments` nas versões anteriores do ActionScript também continham uma propriedade chamada `caller`, que é uma referência á função que chamava a função atual. A propriedade `caller` não está presente no ActionScript 3.0, mas, se uma referência à função de chamada for necessária, você poderá alterar a função de chamada para que ela transmita um parâmetro extra que faça referência a si mesmo.

O parâmetro ... (rest)

O ActionScript 3.0 apresenta uma nova declaração de parâmetro chamada de parâmetro ... (rest). Esse parâmetro permite especificar um parâmetro de matriz que aceita uma grande quantidade de argumentos delimitados por vírgula. O parâmetro pode ter qualquer nome que não seja uma palavra reservada. Essa declaração de parâmetro deve ser o último parâmetro especificado. Usar este parâmetro torna o objeto `arguments` indisponível. Embora o parâmetro ... (rest) ofereça a mesma funcionalidade que a matriz `arguments` e a propriedade `arguments.length`, ele não fornece uma funcionalidade semelhante à fornecida por `arguments.callee`. Você deve se certificar de que não será preciso usar `arguments.callee` antes de usar o parâmetro ... (rest).

O exemplo a seguir reescreve a função `traceArgArray()` usando o parâmetro ... (rest) em vez do 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
```

O parâmetro ... (rest) também pode ser usado com outros parâmetros, contanto que o último parâmetro seja listado. O exemplo a seguir modifica a função `traceArgArray()` para que seu primeiro parâmetro, `x`, seja do tipo int e o segundo use o parâmetro ... (rest). A saída ignora o primeiro valor, porque o primeiro parâmetro não faz mais parte da matriz criada pelo parâmetro ... (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
```

Funções como objetos

As funções no ActionScript 3.0 são objetos. Durante a criação de uma função, você cria um objeto que não apenas pode ser transmitido como um parâmetro para outra função, mas também possui propriedades e métodos anexados.

As funções transmitidas como argumentos para outra função são transmitidas por referência e não por valor. Durante a transmissão de uma função como um argumento, é usado apenas o identificador e não o operador parênteses usado para chamar o método. Por exemplo, o código a seguir transmite uma função chamada `clickListener()` como um argumento para o método `addEventListener()`:

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

Embora possa parecer estranho aos programadores que não conhecem o ActionScript, as funções podem ter propriedades e métodos, assim como qualquer outro objeto. Na verdade, cada função possui uma propriedade somente leitura chamada `length` que armazena o número de parâmetros definido para a função. Isso é diferente na propriedade `arguments.length`, que relata o número de argumentos enviados à função. Lembre-se de que, no ActionScript, o número de argumentos enviados a uma função pode exceder o número de parâmetros definido para ela. O exemplo a seguir, que é compilado somente no modo padrão porque o modo estrito requer uma correspondência exata entre o número de argumentos transmitidos e o número de parâmetros definido, mostra a diferença entre as duas propriedades:

```
// 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 */
```

No modo padrão, é possível especificar suas próprias propriedades de função definindo-as fora do corpo da função. As propriedades de função podem servir como propriedades quase estáticas que permitem salvar o estado de uma variável relacionada à função. Por exemplo, você pode querer controlar o número de vezes que uma função específica é chamada. Essa funcionalidade pode ser útil quando você escreve um jogo e deseja controlar o número de vezes que um usuário usa um comando especifico, embora também seja possível usar uma propriedade de classe estática para isso. O exemplo a seguir, que é compilado somente no modo padrão porque o modo estrito não permite adicionar propriedades dinâmicas às funções, cria uma propriedade de função fora da declaração de função e incrementa a propriedade sempre que a função é chamada:

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

someFunction.counter = 0;

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

Escopo da função

Um escopo de função determina não apenas o local em um programa no qual a função pode ser chamada, mas também as definições que ela pode acessar. As mesmas regras de escopo válidas para os identificadores de variável se aplicam aos identificadores de função. Uma função declarada no escopo global está disponível em todo o código. Por exemplo, o ActionScript 3.0 contém funções globais, tais como `isNaN()` e `parseInt()`, que estão disponíveis em qualquer lugar no seu código. Uma função aninhada (uma função declarada dentro de outra função) pode ser usada em qualquer lugar na função na qual ela foi declarada.

A cadeia do escopo

Sempre que uma função começa com a execução, vários objetos e propriedades são criados. Em primeiro lugar, é criado um objeto especial chamado *objeto de ativação* que armazena os parâmetros e quaisquer variáveis locais ou funções declaradas no corpo da função. Não é possível acessar o objeto de ativação diretamente, porque ele é um mecanismo interno. Em segundo lugar, é criada uma *cadeia do escopo* que contém uma lista ordenada de objetos em que o tempo de execução verifica as declarações de identificador. Cada função executada tem uma cadeia de escopo que é armazenada em uma propriedade interna. Para uma função aninhada, a cadeia do escopo começa com seu próprio objeto de ativação, seguido pelo objeto de ativação de sua função pai. A cadeia continua assim até atingir o objeto global. O objeto global é criado quando um programa do ActionScript começa e contém todas as variáveis e funções globais.

Fechamentos de função

Um *fechamento de função* é um objeto que contém um instantâneo de uma função e seu *ambiente léxico*. O ambiente léxico de uma função inclui todas as variáveis, propriedades, métodos e objetos na cadeia do escopo da função, além de seus valores. Os fechamentos de função são criados sempre que uma função é executada independentemente de um objeto ou uma classe. O fato de que os fechamentos de função mantêm o escopo no qual eles foram definidos cria resultados interessantes quando uma função é transmitida como um argumento ou um valor de retorno em um escopo diferente.

Por exemplo, o código a seguir cria duas funções: `foo()`, que retorna uma função aninhada chamada `rectArea()` que calcula a área de um retângulo, e `bar()`, que chama `foo()` e armazena o fechamento de função retornado em uma variável chamada `myProduct`. Muito embora a função `bar()` defina sua própria variável local `x` (com um valor 2), quando o fechamento de função `myProduct()` é chamado, ela mantém a variável `x` (com um valor 40) definida na função `foo()`. A função `bar()`, portanto, retorna o valor `160` em vez 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
```

Os métodos se comportam de forma semelhante pois também mantêm as informações sobre o ambiente léxico no qual são criados. Essa característica é a mais notável quando um método é extraído de sua ocorrência, que cria um método vinculado. A principal diferença entre um fechamento de função e um método vinculado é que o valor da palavra-chave `this` em um método vinculado sempre se refere à ocorrência à qual ele foi inicialmente anexado, enquanto que, em um fechamento de função, o valor da palavra-chave `this` pode ser alterado.

Capítulo 4: Programação orientada a objetos no ActionScript

Introdução à programação orientada a objetos

A programação orientada a objetos (OOP) é uma forma de organizar o código em um programa agrupando-o em objetos. O termo *objeto*, neste sentido, significa um elemento individual que inclui informações (valores de dados) e funcionalidade. Quando você usa uma abordagem orientada a objetos para organizar um programa, você agrupa várias informações em particular com funcionalidades comuns ou ações associadas com essas informações. Por exemplo, você poderia agrupar as informações musicais, como título do álbum, título da trilha ou nome do artista, com funcionalidades como "adicionar trilha à lista de reprodução" ou "reproduzir todas as músicas deste artista". Essas informações são combinadas em único item, um objeto (por exemplo, um “Album” ou “MusicTrack”). Integrar valores e funções oferece várias vantagens. Uma vantagem principal é que você só vai precisar usar uma única variável em vez de várias. Além disso, isso mantém as funcionalidades relacionadas juntas. Por fim, combinar as informações e as funcionalidades lhe permite estruturar os programas de uma forma que se aproxime mais do mundo real.

Classes

Uma classe é uma representação abstrata de um objeto. Uma classe armazena informações sobre os tipos de dados que um objeto pode manter e os comportamentos que um objeto pode exibir. A utilidade dessa abstração não é necessariamente visível quando você grava pequenos scripts que contêm apenas alguns objetos que interagem entre si. No entanto, à medida que aumenta o escopo de um programa, o número de objetos que devem ser gerenciados também sobe. Nesse caso, as classes permitem controlar melhor como os objetos são criados e como interagem uns com os outros.

Já no ActionScript 1.0, os programadores do ActionScript podiam usar objetos Function para criar construções semelhantes a classes. O ActionScript 2.0 adicionou suporte formal para classes com palavras-chave, como `class` e `extends`. O ActionScript 3.0 só não continua a oferecer suporte às palavras-chaves introduzidas no ActionScript 2.0. Ele também acrescenta novos recursos. Por exemplo, o ActionScript 3.0 inclui um controle de acesso aprimorado com os atributos `protected` e `internal`. Além disso, fornece um melhor controle sobre a herança com as palavras-chave `final` e `override`.

Para os desenvolvedores que criaram classes em linguagens de programação como Java, C++ ou C#, o ActionScript oferece uma experiência parecida. O ActionScript compartilha muitas das mesmas palavras-chaves e nomes de atributos, como `class`, `extends` e `public`.

Nota: *Na documentação do Adobe ActionScript, o termo propriedade significa qualquer membro de um objeto ou classe, incluindo variáveis, constantes e métodos. Além disso, embora os termos classe e estática sejam sempre usados alternadamente, aqui esses termos são distintos. Por exemplo, a expressão "propriedades de classe" é usada para indicar todos os membros de uma classe, em vez de apenas membros estáticos.*

Definições de classes

As definições de classes do ActionScript 3.0 usam sintaxe semelhante à sintaxe usada no ActionScript 2.0. A sintaxe apropriada para uma definição de classe requer a palavra-chave `class` seguida pelo nome da classe. O corpo da classe, que está entre chaves ({}), segue o nome da classe. Por exemplo, o código a seguir cria uma classe denominada Shape que contém uma variável denominada `visible`:

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

Uma alteração significativa na sintaxe envolve definições de classes que estão dentro de um pacote. No ActionScript 2.0, se uma classe estiver dentro de um pacote, o nome do pacote deverá estar incluído na declaração da classe. No ActionScript 3.0, que introduz a instrução `package`, o nome do pacote deve estar incluído na declaração do pacote em vez de na declaração da classe. Por exemplo, as seguintes declarações de classe mostram como a classe BitmapData, que faz parte do pacote flash.display, é definida no ActionScript 2.0 e no ActionScript 3.0:

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

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

Atributos de classes

O ActionScript 3.0 permite modificar definições de classe usando um dos quatro seguintes atributos:

Atributo	Definição
`dynamic`	Permitir que propriedades sejam adicionadas a ocorrências em tempo de execução.
`final`	Não deve ser estendido por outra classe.
`internal` (padrão)	Visível para referências dentro do pacote atual.
`public`	Visível para referências em todos os lugares.

Para cada um desses atributos, exceto para `internal`, você inclui explicitamente o atributo para obter o comportamento associado. Por exemplo, se você não incluir o atributo `dynamic` ao definir uma classe, você não pode adicionar propriedades a uma ocorrência da classe em tempo de execução. Você atribui explicitamente um atributo colocando-o no início da definição de classe, conforme demonstrado no código a seguir:

```
dynamic class Shape {}
```

Observe que a lista não inclui um atributo denominado `abstract`. As classes abstratas não têm suporte no ActionScript 3.0. Observe também que a lista não inclui atributos denominados `private` e `protected`. Esses atributos têm significado apenas dentro de uma definição de classe e não podem ser aplicados às próprias classes. Se você não desejar que uma classe seja publicamente visível fora de um pacote, coloque-a no pacote e marque-a com o atributo `internal`. Como alternativa, você pode omitir os atributos `internal` e `public` e o compilador adiciona automaticamente o atributo `internal` para você. Você também pode definir uma classe para ser apenas visível dentro do arquivo de origem no qual está definido. Coloque a classe na parte de baixo do seu arquivo de origem, abaixo da chave de fechamento da definição de pacote

Corpo da classe

O corpo da classe está dentro de chaves. Ele define as variáveis, constantes e métodos de sua classe. O exemplo a seguir mostra a declaração para a classe Accessibility no ActionScript 3.0:

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

Você também pode definir um espaço para nomes dentro de um corpo de classe. O exemplo a seguir mostra como um espaço para nomes pode ser definido dentro de um corpo de classe e usado como atributo de um método naquela classe:

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

O ActionScript 3.0 permite incluir não apenas definições em um corpo da classe, mas também instruções. As instruções que estão dentro do corpo da classe, mas fora de uma definição de método, são executadas exatamente uma vez. Essa execução acontece quando a definição de classe é encontrada primeiro e o objeto de classe associado é criado. O exemplo a seguir inclui uma chamada para uma função externa, `hello()`, e uma instrução `trace` que produz uma mensagem de confirmação quando a classe é definida:

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

No ActionScript 3.0, é permitido definir uma propriedade estática e uma propriedade de ocorrência com o mesmo nome no mesmo corpo da classe. Por exemplo, o código a seguir declara uma variável estática denominada `message` e uma variável da ocorrência do mesmo nome:

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

Atributos de propriedade de classes

Em discussões do modelo de objeto do ActionScript, o termo *property* significa qualquer coisa que possa ser um membro de uma classe, incluindo variáveis, constantes e métodos. No entanto, no Referência do Adobe ActionScript® 3.0 para Adobe® Flash® Platform, o termo é usado de forma mais estreita. Nesse contexto, o termo propriedade inclui apenas os membros de classe que são variáveis ou são definidos por um método getter ou setter. No ActionScript 3.0, há um conjunto de atributos que pode ser usado com qualquer propriedade de uma classe. A tabela a seguir lista esse conjunto de atributos.

Atributo	Definição
`internal` (padrão)	Visível para referências dentro do mesmo pacote.
`private`	Visível para referências na mesma classe.
`protected`	Visível para referências na mesma classe e em classes derivadas.
`public`	Visível para referências em todos os lugares.
`static`	Especifica que uma propriedade pertence à classe, ao contrário das ocorrências da classe.
`UserDefinedNamespace`	Nome do espaço para nomes personalizado definido pelo usuário.

Atributos de espaço para nomes de controle de acesso

O ActionScript 3.0 fornece quatro atributos especiais que controlam o acesso às propriedades definidas dentro de uma classe: `public`, `private`, `protected` e `internal`.

O atributo `public` torna a propriedade visível em qualquer lugar no script. Por exemplo, para disponibilizar um método para o código fora de seu pacote, declare o método com o atributo `public`. Isso é verdadeiro para qualquer propriedade, se ela for declarada usando as palavras-chave `var`, `const` ou `function`.

O atributo `private` torna a propriedade visível apenas para chamadores dentro da classe de definição da propriedade. Esse comportamento é diferente daquele do atributo `private` no ActionScript 2.0, que permite que uma subclasse acesse uma propriedade private em uma superclasse. Outra alteração significativa no comportamento que tem a ver com acesso em tempo de execução. No ActionScript 2.0, a palavra-chave `private` proibia o acesso apenas em tempo de compilação e era facilmente contornada em tempo de execução. No ActionScript 3.0, isso não é mais verdade. Propriedades que estão marcadas como `private` não estão disponíveis em tempo de compilação e em tempo de execução.

Por exemplo, o código a seguir cria uma classe simples denominada PrivateExample com uma variável private, e tenta acessar a variável private de fora da classe.

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this
is a run-time error.
```

No ActionScript 3.0, uma tentativa de acessar uma propriedade private usando o operador ponto (`myExample.privVar`) resultará em um erro em tempo de compilação se você estiver usando modo estrito. Caso contrário, o erro será relatado em tempo de execução, exatamente como quando você usa o operador de acesso de propriedade (`myExample["privVar"]`).

A tabela a seguir resume os resultados da tentativa de acessar uma propriedade private que pertence a uma classe selada (não dinâmica):

	modo Estrito	**modo Padrão**
operador ponto (`.`)	erro em tempo de compilação	erro em tempo de execução
operador colchete (`[]`)	erro em tempo de execução	erro em tempo de execução

Em classes declaradas com o atributo `dynamic`, as tentativas de acessar uma variável private não resulta em um erro em tempo de execução. Em vez disso, a variável não é visível, por isso o valor `undefined` é devolvido. No entanto, ocorrerá um erro em tempo de compilação, se você usar o operador ponto no modo estrito. O exemplo a seguir é o mesmo do exemplo anterior, exceto que a classe PrivateExample é declarada como uma classe dinâmica:

```
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
```

As classes dinâmicas geralmente retornam o valor `undefined` em vez de gerar um erro quando o código externo a uma classe tenta acessar uma propriedade private. A tabela a seguir mostra que um erro é gerado apenas quando o operador ponto é usado para acessar uma propriedade private no modo estrito:

	modo Estrito	**modo Padrão**
operador ponto (`.`)	erro em tempo de compilação	`undefined`
operador colchete (`[]`)	`undefined`	`undefined`

O atributo `protected`, que é novo para o ActionScript 3.0, torna uma propriedade visível a chamadores dentro de sua própria classe ou em uma subclasse. Em outras palavras, uma propriedade protected está disponível dentro de sua própria classe ou a classes que estão em qualquer lugar abaixo dela na hierarquia de heranças. Isso será verdadeiro se a subclasse estiver no mesmo pacote ou em um pacote diferente.

Para aqueles familiarizados com o ActionScript 2.0, essa funcionalidade é semelhante ao atributo `private` no ActionScript 2.0. O atributo `protected` do ActionScript 3.0 também é semelhante ao atributo `protected` em Java. Ele difere na medida em que a versão em Java também permite o acesso aos chamadores no mesmo pacote. O atributo `protected` é útil quando você tem uma variável ou método que as subclasses precisam, mas que você deseja ocultar do código que está fora da cadeia de heranças.

O atributo `internal`, que é novo para o ActionScript 3.0, torna uma propriedade visível a chamadores dentro de seu próprio pacote. Este é o atributo padrão do código dentro de um pacote e se aplica a qualquer propriedade que não tenha nenhum dos seguintes atributos:

- `public`
- `private`
- `protected`
- um espaço para nomes definido pelo usuário

O atributo `internal` é semelhante ao controle de acesso padrão no Java, embora no Java não haja nenhum nome explícito para este nível de acesso, e ele pode ser alcançado apenas por meio da omissão de qualquer outro modificador de acesso. O atributo `internal` está disponível no ActionScript 3.0 para fornecer a opção de indicar explicitamente a intenção tornar a propriedade visível apenas para chamadores dentro de seu próprio pacote.

atributo static

O atributo `static`, que pode ser usado com propriedades declaradas com as palavras-chave `var`, `const` ou `function`, permite anexar uma propriedade à classe em vez de às ocorrências da classe. O código externo à classe deve chamar propriedades estáticas usando o nome da classe em vez do nome de uma ocorrência.

As propriedades estáticas não são herdadas pelas subclasses, mas fazem parte da cadeia de escopos de uma subclasse. Isso significa que dentro do corpo de uma subclasse, uma variável ou método estático pode ser usado sem fazer referência à classe na qual ele foi definido.

Atributos de espaço para nomes definidos pelo usuário

Como alternativa aos atributos de controle de acesso predefinidos, você pode criar um espaço para nomes personalizado para uso como um atributo. Apenas um atributo de espaço para nomes pode ser usado por definição, e você não pode usar um atributo de espaço para nomes em combinação com qualquer um dos atributos de controle de acesso (`public`, `private`, `protected`, `internal`).

Variáveis

As variáveis podem ser declaradas com as palavras-chave `var` ou `const`. Variáveis declaradas com a palavra-chave `var` podem ter seus valores alterados várias vezes durante a execução de um script. Variáveis declaradas com a palavra-chave `const` são chamadas *constants* e valores podem ser atribuídos a elas apenas uma vez. Uma tentativa de atribuir um novo valor a uma constante inicializada resulta em um erro.

Variáveis estáticas

As variáveis estáticas são declaradas usando uma combinação da palavra-chave `static` e da instrução `var` ou `const`. As variáveis estáticas que são anexadas a uma classe em vez de a uma ocorrência de a uma classe são úteis para armazenar e compartilhar informações que se aplicam a uma classe inteira de objetos. Por exemplo, uma variável estática será apropriada se você desejar manter uma contagem do número de vezes que uma classe é instanciada ou se desejar armazenar o número máximo de ocorrências da classe que são permitidas.

O exemplo a seguir cria uma variável `totalCount` para rastrear o número de instanciações de classes e uma constante `MAX_NUM` para armazenar o número máximo de instanciações. As variáveis `totalCount` e `MAX_NUM` são estáticas porque contêm valores que se aplicam à classe como um todo em vez de a uma ocorrência específica.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

O código que é externo à classe StaticVars e qualquer uma de suas subclasses pode fazer referência às propriedades `totalCount` e `MAX_NUM` apenas por meio da própria classe. Por exemplo, o código a seguir funciona:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

Não é possível acessar variáveis estáticas por meio de uma ocorrência da classe, portanto, o código a seguir retorna erros:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Variáveis que são declaradas com as palavras-chave `static` e `const` devem ser inicializadas ao mesmo tempo em que a constante é declarada, como faz a classe StaticVars para `MAX_NUM`. Você não pode atribuir um valor a `MAX_NUM` dentro do construtor ou de um método da ocorrência. O código a seguir gera um erro, porque não é uma maneira válida de inicializar uma constante estática:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Variáveis de ocorrência

As variáveis de ocorrência incluem propriedades declaradas com as palavras-chave `var` e `const`, mas sem a palavra-chave `static`. As variáveis de ocorrência que são anexadas às ocorrências da classe em vez de a uma classe inteira são úteis para armazenar valores específicos a uma ocorrência. Por exemplo, a classe Array tem uma propriedade de ocorrência denominada `length` que armazena o número de elementos da matriz que é mantida por uma ocorrência específica da classe Array.

Variáveis de ocorrência, se declaradas como `var` ou `const`, não podem ser substituídas em uma subclasse. No entanto, você pode alcançar funcionalidade semelhante substituindo variáveis pelos métodos getter e setter.

Métodos

Métodos são funções que fazem parte de uma definição de classe. Depois que uma ocorrência da classe é criada, um método é ligado a essa ocorrência. Ao contrário de uma função declarada fora de uma classe, um método não pode ser usado à parte da ocorrência à qual ele está anexado.

Os métodos são definidos usando a palavra-chave `function`. Como ocorre com qualquer propriedade de classe, você pode aplicar qualquer um dos atributos de propriedade de classe aos métodos, incluindo private, protected, public, internal, static ou custom namespace. Você pode usar uma instrução de função como a seguinte:

```
public function sampleFunction():String {}
```

Ou pode usar uma variável à qual atribuir uma expressão de função, da seguinte maneira:

```
public var sampleFunction:Function = function () {}
```

Na maior parte dos casos, use uma instrução function em vez de uma expressão function pelas seguintes razões:

- As instruções function são mais concisas e mais fáceis de ler.
- As instruções function permitem usar as palavras-chave `override` e `final`.
- As instruções function criam uma ligação mais forte entre o identificador (o nome da função), e o código dentro do corpo do método. Como o valor de uma variável pode ser alterado com uma instrução de atribuição, a conexão entre uma variável e sua expressão de função pode ser desfeita a qualquer momento. Embora você possa solucionar esse problema declarando a variável com `const` em vez de `var`, essa técnica não é considerada uma prática recomendada, porque dificulta a leitura do código e impede o uso das palavras-chave `override` e `final`.

Um caso em que você deve usar uma expressão function é ao optar por anexar uma função ao objeto de protótipo.

Métodos do construtor

Os métodos de construtor, às vezes chamados de *construtores*, são funções que compartilham o mesmo nome da classe na qual eles são definidos. Qualquer código incluído em um método de construtor é executado todas as vezes que uma ocorrência da classe é criada com a palavra-chave `new`. Por exemplo, o código a seguir define uma classe simples denominada Example que contém uma única propriedade denominada `status`. O valor inicial da variável `status` é definido dentro da função de construtor.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Os métodos de construtor podem ser apenas públicos, mas o uso do atributo `public` é opcional. Você não pode usar nenhum dos outros especificadores de controle de acesso, inclusive `private`, `protected` ou `internal`, em um construtor. Você também não pode usar um espaço para nomes definido pelo usuário com um método de construtor.

Um construtor pode fazer uma chamada explícita para o construtor de sua superclasse direta usando a instrução `super()`. Se o construtor da superclasse não for chamado explicitamente, o compilador inserirá automaticamente uma chamada antes da primeira instrução no corpo do construtor. Você também pode chamar métodos da superclasse usando o prefixo `super` como uma referência à superclasse. Se você decidir usar `super()` e `super` no mesmo corpo do construtor, verifique se `super()` é chamado primeiro. Caso contrário, a referência `super` não se comporta conforme esperado. O construtor `super()` deve ser chamado também antes de qualquer instrução `throw` ou `return`.

O exemplo a seguir demonstra o que acontecerá se você tentar usar a referência `super` antes de chamar o construtor `super()`. Uma nova classe, ExampleEx, estende a classe Example. O construtor ExampleEx tenta acessar a variável status definida em sua superclasse, mas faz isso antes de chamar `super()`. A instrução `trace()` dentro do construtor ExampleEx produz o valor `null`, porque a variável `status` não está disponível até que o construtor `super()` seja executado.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Embora seja válido usar a instrução `return` dentro de um construtor, não é permitido retornar um valor. Em outras palavras, a instrução `return` não deve ter expressões ou valores associados. De forma correspondente, os métodos do construtor não têm permissão para retornar valores, o que significa que nenhum tipo de retorno pode ser especificado.

Se você não definir um método de construtor em sua classe, o compilador cria automaticamente um construtor vazio. Se a classe estender outra classe, o compilador inclui uma chamada `super()` no construtor gerado.

Métodos estáticos

Os métodos estáticos, também chamados de *métodos de classe*, são métodos que são declarados com a palavra-chave `static`. Os métodos estáticos que são anexados a uma classe e não em uma ocorrência de uma classe, são úteis para encapsular a funcionalidade que afeta algo diferente do estado de uma ocorrência individual. Como os métodos estáticos são anexados a uma classe como um todo, eles podem ser acessados apenas por meio de uma classe e não por meio de uma ocorrência da classe.

Os métodos estáticos são úteis para encapsular funcionalidade que não está limitada a afetar o estado das ocorrências da classe. Em outras palavras, um método deverá ser estático se fornecer funcionalidade que não afete diretamente o valor de uma ocorrência de classe. Por exemplo, a classe Date tem um método estático denominado `parse()`, que obtém uma string e converte-a em um número. O método é estático porque não afeta uma ocorrência individual da classe. Em vez disso, o método `parse()` obtém uma string que representa um valor de data, analisa-a e retorna um número em um formato compatível com a representação interna de um objeto Date. Esse método não é um método da ocorrência, porque não faz sentido aplicar o método a uma ocorrência da classe Date.

Compare o método estático `parse()` com um dos métodos da ocorrência da classe Date, como `getMonth()`. O método `getMonth()` é um método de ocorrência, porque ele opera diretamente no valor de uma ocorrência, recuperando um componente específico, o mês, de uma ocorrência Date.

Como os métodos estáticos não estão ligados a ocorrências individuais, você não pode usar as palavras-chave `this` ou `super` dentro do corpo de um método estático. As referências `this` e `super` têm significado apenas dentro do contexto de um método de ocorrência.

Em comparação com algumas outras linguagens de programação com base em classe, os métodos estáticos no ActionScript 3.0 não são herdados.

Métodos de ocorrência

Métodos de ocorrência são métodos declarados sem a palavra-chave `static`. Os métodos de ocorrência que são anexados a ocorrências de uma classe e não à classe como um todo, são úteis para implementar a funcionalidade que afeta ocorrências individuais de uma classe. Por exemplo, a classe Array contém um método de ocorrência denominado `sort()` que opera diretamente em ocorrências de Array.

Dentro do corpo de um método de ocorrência, as variáveis de ocorrência e estáticas estão no escopo, o que significa que as variáveis definidas na mesma classe podem ser referenciadas usando um identificador simples. Por exemplo, a seguinte classe, CustomArray, estende a classe Array. A classe CustomArray define uma variável estática denominada `arrayCountTotal` para rastrear o número total de ocorrências da classe, uma variável de ocorrência denominada `arrayNumber` que rastreia a ordem na qual as ocorrências foram criadas e o método de ocorrência denominado `getPosition()` que retorna os valores dessas variáveis.

```
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);
    }
}
```

Embora o código externo à classe deva acessar a variável estática `arrayCountTotal` por meio do objeto de classe, usando `CustomArray.arrayCountTotal`, o código que reside dentro do corpo do método `getPosition()` pode fazer referência diretamente à variável estática `arrayCountTotal`. Isso é verdadeiro mesmo para variáveis estáticas em superclasses. Apesar das propriedades estáticas não serem herdadas no ActionScript 3.0, as propriedades estáticas em superclasses estão no escopo. Por exemplo, a classe Array tem algumas variáveis estáticas, uma das quais é uma constante denominada `DESCENDING`. O código que reside em uma subclasse Array pode acessar a constante estática `DESCENDING` usando um identificador simples:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

O valor da referência `this` dentro do corpo de um método da ocorrência é uma referência à ocorrência à qual o método está anexado. O código a seguir demonstra que a referência `this` aponta para a ocorrência que contém o método:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

A herança de métodos de ocorrência pode ser controlada com as palavras-chave `override` e `final`. Você pode usar o atributo `override` para redefinir um método herdado e o atributo `final` para impedir que as subclasses substituam um método.

Métodos de acessor get e set

As funções de acessor get e set, também chamadas de *getters* e *setters*, permitem aderir aos princípios de programação de ocultação de informações e encapsulamento enquanto fornece uma interface de programação fácil de usar para as classes criadas. As funções get e set permitem manter as propriedades de classe privadas para a classe, mas permitem que usuários da classe acessem essas propriedades como se fossem acessar uma variável de classe em vez de chamar um método de classe.

A vantagem dessa abordagem é que ela permite evitar as funções de acessor tradicional com nomes inadequados, como `getPropertyName()` e `setPropertyName()`. Outra vantagem de getters e setters é que você pode evitar ter duas funções voltadas ao público para cada propriedade que permitam acesso de leitura e gravação.

O seguinte exemplo de classe, denominado GetSet, inclui funções do acessor get e set denominadas `publicAccess()` que fornecem acesso à variável 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;
    }
}
```

Se você tentar acessar a propriedade `privateProperty` diretamente, ocorre um erro, da seguinte maneira:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Em vez disso, um usuário da classe GetSet usa alguma coisa que parece ser uma propriedade denominada `publicAccess`, mas que realmente é um par de funções de acessor get e set que operam na propriedade privada denominada `privateProperty`. O exemplo a seguir instancia a classe GetSet e, em seguida, define o valor da `privateProperty` usando o acessor público denominado `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

As funções getter e setter também permitem substituir propriedades que são herdadas de uma superclasse, algo que não é possível ao usar variáveis de membros de classe normal. As variáveis de membros de classe que são declaradas usando a palavra-chave `var` não podem ser substituídas em uma subclasse. No entanto as propriedades criadas usando as funções getter e setter não têm essa restrição. É possível usar o atributo `override` nas funções getter e setter que são herdadas de uma superclasse.

Métodos vinculados

Um método vinculado, às vezes chamado *fechamento de método*, é simplesmente um método extraído de sua ocorrência. Os exemplos de métodos vinculados incluem métodos que são passados como argumentos para uma função ou retornados de uma função como valores. Novo no ActionScript 3.0, um método vinculado é semelhante a um fechamento de função já que ele retém seu ambiente léxico mesmo quando extraído de sua ocorrência. No entanto a diferença principal entre um método vinculado e um fechamento de função é que a referência `this` de um método vinculado permanece vinculada, ou ligada, à ocorrência que implementa o método. Em outras palavras, a referência `this` em um método vinculado sempre aponta para o objeto original que implementou o método. Para fechamentos de funções, a referência `this` é genérica, o que significa que ela aponta para qualquer objeto com o qual a função está associada no momento em que é chamada.

É importante compreender os métodos vinculados ao usar a palavra-chave `this`. Lembre-se de que a palavra-chave `this` fornece uma referência ao objeto pai de um método. A maioria dos programadores do ActionScript espera que a palavra-chave `this` sempre representa o objeto ou a classe que contém a definição de um método. No entanto, sem a vinculação do método, isso não é sempre verdadeiro. Em versões anteriores do ActionScript, por exemplo, a referência `this` não se referia sempre à ocorrência que implementou o método. Quando os métodos são extraídos de

uma ocorrência no ActionScript 2.0, não somente a referência `this` não está vinculada à ocorrência original, mas também os métodos as variáveis de membros da classe da ocorrência não estão disponíveis. Esse não é um problema no ActionScript 3.0 porque os métodos vinculados são criados automaticamente quando você transmite um método como um parâmetro. Os métodos vinculados garantem que a palavra-chave `this` sempre faça referência ao objeto ou à classe na qual um método está definido.

O código a seguir define uma classe denominada ThisTest que contém um método denominado `foo()` que define o método vinculado e um método denominado `bar()` que retorna o método vinculado. O código externo à classe cria uma ocorrência da classe ThisTest, chama o método `bar()` e armazena o valor de retorno em uma variável 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 */
```

As duas últimas linhas do código mostram que a referência `this` no método vinculado `foo()` ainda aponta para uma ocorrência da classe ThisTest, mesmo que a referência `this` na linha imediatamente antes dela aponte para o objeto global. Além disso, o método vinculado armazenado na variável `myFunc` ainda tem acesso às variáveis de membros da classe ThisTest. Se esse mesmo código estiver em execução no ActionScript 2.0, as referências `this` corresponderão, e a variável `num` será `undefined`.

Uma área em que a adição de métodos vinculados é mais perceptível é a de identificadores de eventos, porque o método `addEventListener()` exige que você passe uma função ou método como um argumento.

Enumerações com classes

Enumerações são tipos de dados personalizados criados para encapsular um pequeno conjunto de valores. O ActionScript 3.0 não oferece suporte a um recurso de enumeração específico, ao contrário do C++ com sua palavra-chave `enum` ou do Java com sua interface de Enumeração. No entanto, você pode criar enumerações usando classes e constantes estáticas. Por exemplo, a classe PrintJob no ActionScript 3.0 usa uma enumeração denominada PrintJobOrientation para armazenar os valores`"landscape"` e `"portrait"`, conforme mostrado no código a seguir:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Por convenção, uma classe de enumeração é declarada com o atributo `final`, porque não há nenhuma necessidade de estender a classe. A classe inclui apenas membros estáticos, o que significa que você não cria ocorrências da classe. Em vez disso, você acessa os valores de enumeração diretamente por meio do objeto de classe, conforme mostrado no trecho de código a seguir:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Todas as classes de enumeração no ActionScript 3.0 contêm apenas variáveis do tipo String, int ou uint. A vantagem de usar enumerações em vez de string literal ou valores numéricos é que os erros tipográficos são mais fáceis de encontrar com enumerações. Se você digitar incorretamente o nome de uma enumeração, o compilador do ActionScript gerará um erro. Se você usar valores literais, o compilador não reclamará se você digitar uma palavra incorretamente ou usar um número incorreto. No exemplo anterior, o compilador gerará um erro se o nome da constante de enumeração estiver incorreto, conforme mostrado no trecho a seguir:

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

No entanto o compilador não gerará um erro se você digitar de forma incorreta um valor literal da string, da seguinte maneira:

```
    if (pj.orientation == "portrai") // no compiler error
```

A segunda técnica para criar enumerações também envolve a criação de uma classe separada com propriedades estáticas para a enumeração. No entanto essa técnica difere já que cada uma das propriedades estáticas contém uma ocorrência da classe em vez de uma string ou um valor inteiro. Por exemplo, o código a seguir cria uma classe de enumeração para os dias da 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();
}
```

Essa técnica não é usada pelo ActionScript 3.0, mas é usada por muitos desenvolvedores que preferem a verificação de tipo aprimorada que a técnica fornece. Por exemplo, um método que retorna um valor de enumeração pode restringir o valor de retorno para o tipo de dados de enumeração. O código a seguir mostra não apenas uma função que retorna um dia da semana, mas também uma chamada de função que usa o tipo de enumeração como uma anotação 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();
```

Você também pode aprimorar a classe Day para que ela associe um inteiro a cada dia da semana, e forneça um método `toString()` que retorne uma representação de string do dia.

Classes de ativos incorporados

O ActionScript 3.0 usa classes especiais, chamadas *classes de ativos incorporados*, para representar ativos incorporados. Um *ativo incorporado* é um ativo, como um som, imagem ou fonte, que é incluído em um arquivo SWF no momento da compilação. Incorporar um ativo, em vez de carregá-lo dinamicamente, garante que ele está disponível em tempo de execução, mas ao custo do tamanho do arquivo SWF aumentado.

Uso de classes de ativos incorporados no Flash Professional

Para incorporar um ativo, coloque primeiro o ativo em uma biblioteca do arquivo FLA. Em seguida, use a propriedade de ligação do ativo para fornecer um nome para a classe de ativo incorporado. Se uma classe por esse nome não puder ser encontrada no caminho de classe, ela será gerada automaticamente para você. Portanto você pode criar uma ocorrência da classe de ativo incorporado e usar todas as propriedades e métodos definidos ou herdados por essa classe. Por exemplo, o código a seguir pode ser usado para reproduzir um som incorporado que está vinculado a uma classe de ativo incorporado denominada PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Como alternativa, você pode usar a tag de metadados `[Embed]` para incorporar os ativos a um projeto do Flash Professional, conforme descrito a seguir. Se você usar a tag de metadados `[Embed]` no seu código, o Flash Professional usa o compilador do Flex para compilar seu projeto no lugar do compilador do Flash Professional.

Uso de classes de ativos incorporadas com o compilador do Flex

Se estiver compilando seu código com o compilador do Flex, para incorporar um ativo no código do ActionScript, use a tag de metadados `[Embed]`. Coloque o ativo na pasta de origem principal ou em outra pasta que esteja no caminho de criação do projeto. Quando o compilador do Flex encontra uma tag de metadados Embed, ele cria a classe de ativos incorporada para você. É possível acessar a classe por meio de uma variável de tipo de dados Class que você declara imediatamente depois da tag de metadados `[Embed]`. Por exemplo, o código a seguir incorpora um som denominado sound1.mp3 e usa uma variável denominada `soundCls` para armazenar uma referência na classe de ativo incorporado associada a esse som. Em seguida, o exemplo cria uma ocorrência da classe de ativo incorporado e chama o método `play()` nessa ocorrência:

```
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 usar a tag de metadados `[Embed]` em um projeto ActionScript do Flex Builder, importe todas as classes necessárias da estrutura do Flex. Por exemplo, para incorporar sons, importe a classe mx.core.SoundAsset. Para usar a estrutura do Flex, inclua o arquivo framework.swc no caminho de criação do ActionScript. Isso aumenta o tamanho do arquivo SWF.

Adobe Flex

Como alternativa, no Flex você pode incorporar um ativo com a diretiva `@Embed()` em uma definição de tag MXML.

Interfaces

Uma interface é uma coleção de declarações de métodos que permite que objetos não relacionados se comuniquem. Por exemplo, o ActionScript 3.0 define a interface IEventDispatcher que contém declarações de métodos que uma classe pode usar para manipular objetos de eventos. A interface IEventDispatcher estabelece uma maneira padrão para os objetos passarem objetos de eventos entre si. O código a seguir mostra a definição da interface 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;
}
```

As interfaces são baseadas na distinção entre uma interface do método e sua implementação. Uma interface do método inclui todas as informações necessárias para chamá-lo, inclusive o nome do método, todos os seus parâmetros e seu tipo de retorno. Uma implementação do método inclui não apenas as informações da interface, mas também as instruções executáveis que executam o comportamento do método. Uma definição de interface contém apenas interfaces do método, e qualquer classe que implemente a interface é responsável por definir as implementações do método.

No ActionScript 3.0, a classe EventDispatcher implementa a interface IEventDispatcher definindo todos os métodos da interface IEventDispatcher e adicionando corpos de métodos a cada um dos métodos. O exemplo a seguir é um trecho da definição da classe EventDispatcher:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

A interface IEventDispatcher serve como um protocolo que as ocorrências do EventDispatcher usam para processar objetos de eventos e passá-los para outros objetos que também implementaram a interface IEventDispatcher.

Outra maneira de descrever uma interface é dizer que ela define um tipo de dados exatamente como faz uma classe. Conseqüentemente, uma interface pode ser usada como uma anotação de tipo, exatamente como uma classe. Como um tipo de dados, uma interface pode ser usada também com operadores, como os operadores `is` e `as` que exigem um tipo de dados. No entanto, ao contrário de uma classe, uma interface não pode ser instanciada. Essa distinção levou muitos programadores a considerar interfaces como tipos de dados abstratos e as classes como tipos de dados concretos.

Definição de uma interface

A estrutura de uma definição de interface é semelhante à da definição de uma classe, exceto que uma interface pode conter apenas métodos sem nenhum corpo de método. As interfaces não podem incluir variáveis ou constantes, mas podem incluir getters e setters. Para definir uma interface, use a palavra-chave `interface`. Por exemplo, a seguinte interface, IExternalizable, faz parte do pacote flash.utils no ActionScript 3.0. A interface IExternalizable define um protocolo para serializar um objeto, o que significa converter um objeto em um formato adequado para armazenamento em um dispositivo ou para transporte pela rede.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

A interface IExternalizable é declarada com o modificador de controle de acesso `public`. As definições de interface podem ser modificadas apenas pelos especificadores de controle de acesso `public` e `internal`. As declarações de método dentro de uma definição de interface não podem ter nenhum especificador de controle de acesso.

O ActionScript 3.0 segue uma convenção na qual os nomes de interface começam com um `I` maiúsculo, mas você pode usar qualquer identificador válido como o nome de uma interface. As definições de interface são sempre colocadas no nível superior de um pacote. As definições de interface não podem ser colocadas dentro de uma definição de classe ou dentro da definição de outra interface.

As interfaces podem estender uma ou mais interfaces. Por exemplo, a seguinte interface, IExample, estende a interface IExternalizable:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Qualquer classe que implemente a interface IExample deve incluir implementações não apenas para o método `extra()`, mas também para os métodos `writeExternal()` e `readExternal()` herdados da interface IExternalizable.

Implementação de uma interface em uma classe

Uma classe é o único elemento de linguagem do ActionScript 3.0 que pode implementar uma interface. Use a palavra-chave `implements` em uma declaração de classe para implementar uma ou mais interfaces. O exemplo a seguir define duas interfaces, IAlpha e IBeta, e uma classe, Alpha, que implementa as duas:

```
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 {}
}
```

Em uma classe que implementa uma interface, os métodos implementados devem fazer o seguinte:

- Usar o identificador de controle de acesso `public`.
- Usar o mesmo nome do método da interface.
- Ter o mesmo número de parâmetros, cada um com tipos de dados que correspondam aos tipos de dados do parâmetro do método da interface.
- Usar o mesmo tipo de retorno.

```
public function foo(param:String):String {}
```

No entanto, você realmente tem alguma flexibilidade quanto a como nomear os parâmetros de métodos implementados. Embora o número de parâmetros e o tipo de dados de cada parâmetro no método implementado devam corresponder àquele do método da interface, os nomes de parâmetros não precisam corresponder. Por exemplo, no exemplo anterior o parâmetro do método `Alpha.foo()` é denominado `param`:

Mas o parâmetro é denominado `str` no método da interface `IAlpha.foo()`:

```
function foo(str:String):String;
```

Você tem também alguma flexibilidade com valores de parâmetro padrão. Uma definição de interface pode incluir declarações de função com valores de parâmetro padrão. Um método que implementa um declaração de função desse tipo deve ter um valor de parâmetro padrão que seja membro do mesmo tipo de dados que o valor especificado na definição da interface, mas o valor real não precisa corresponder. Por exemplo, o código a seguir define uma interface que contém um método com um valor de parâmetro padrão 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

A seguinte definição de classe implementa a interface Igamma, mas usa um valor do parâmetro padrão diferente:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

O motivo dessa flexibilidade é que as regras para implementação de uma interface são projetadas especificamente para garantir a compatibilidade do tipo de dados, e exigir nomes de parâmetros idênticos e valores de parâmetros padrão não é necessário para atingir esse objetivo.

Herança

Herança é uma forma de reutilização de código que permite que programadores desenvolvam novas classes com base em classes existentes. As classes existentes são sempre conhecidas como *classes base* ou *superclasses*, enquanto as novas classes são chamadas de *subclasses*. A vantagem principal da herança é que ela permite reutilizar código de uma classe base e ainda deixar o código existente inalterado. Além disso, a herança não requer nenhuma alteração no modo como as outras classes interagem com a classe base. Em vez de modificar uma classe existente que pode ter sido completamente testada ou já estar em uso, usando herança você pode tratar essa classe como um módulo integrado que pode ser estendido com propriedades ou métodos adicionais. De forma correspondente, você usa a palavra-chave `extends` para indicar que uma classe herda de outra classe.

A herança também permite que você se beneficie com o *polimorfismo* do código. Polimorfismo é a habilidade de usar um único nome de método para um método que se comporta de maneira diferente ao ser aplicado a diferentes tipos de dados. Um exemplo simples é uma classe base denominada Shape com duas subclasses denominadas Circle e Square. A classe Shape define um método denominado `area()`, que retorna a área da forma. Se o polimorfismo estiver implementado, você poderá chamar o método `area()` em objetos do tipo Circle e Square e fazer com que os cálculos corretos sejam feitos para você. A herança ativa o polimorfismo permitindo que as subclasses sejam herdadas e redefinidas, ou *substituam*, métodos da classe base. No exemplo a seguir, o método `area()` é redefinido pelas classes Circle e Square:

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Como cada classe define um tipo de dados, o uso da herança cria um relacionamento especial entre a classe base e a classe que a estende. Uma subclasse garantidamente possui todas as propriedades de sua classe base, o que significa que uma ocorrência de uma subclasse pode ser sempre substituída por uma ocorrência da classe base. Por exemplo, se um método definir um parâmetro do tipo Shape, é válido passar um argumento do tipo Circle, porque Circle estende Shape, da seguinte maneira:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Propriedades da ocorrência e herança

Uma propriedade da ocorrência, se definida com as palavras-chave `function`, `var` ou `const`, será herdada por todas as subclasses desde que a propriedade não seja declarada com o atributo `private` na classe base. Por exemplo, a classe Event no ActionScript 3.0 tem várias subclasses que herdam propriedades comuns a todos os objetos de eventos.

Para alguns tipos de eventos, a classe Event contém todas as propriedades necessárias para definir o evento. Esses tipos de eventos não exigem propriedades da ocorrência além daquelas definidas na classe Event. Exemplos desses eventos são o evento `complete`, que ocorre quando os dados foram carregados com êxito, e o evento `connect`, que ocorre quando uma conexão de rede foi estabelecida.

O exemplo a seguir é um fragmento da classe Event que mostra algumas das propriedades e métodos herdados por subclasses. Como as propriedades são herdadas, uma ocorrência de qualquer subclasse pode acessar essas propriedades.

```
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 {}
    ...
}
```

Outros tipos de eventos exigem propriedades exclusivas não estão disponíveis na classe Event. Esses eventos são definidos usando subclasses da classe Event para que novas propriedades possam ser adicionadas às propriedades definidas na classe Event. Um exemplo dessa subclasse é a classe MouseEvent, que adiciona propriedades exclusivas a eventos associados a movimento ou a cliques do mouse, como os eventos `mouseMove` e `click`. O exemplo a seguir é um fragmento da classe MouseEvent que mostra a definição de propriedades que existem na subclasse, mas não na classe base:

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

Especificadores de controle de acesso e herança

Se uma propriedade for declarada com a palavra-chave `public`, ela será visível ao código em qualquer lugar. Isso significa que a palavra-chave `public`, ao contrário das palavras-chave `private`, `protected` e `internal`, não coloca nenhuma restrição sobre a herança da propriedade.

Se uma propriedade for declarada com a palavra-chave `private`, ela será visível apenas na classe que a define, o que significa que não será herdada por nenhuma subclasse. Esse comportamento é diferente nas versões anteriores do ActionScript, em que a palavra-chave `private` se comportava de maneira mais semelhante à palavra-chave `protected` do ActionScript 3.0.

A palavra-chave `protected` indica que uma propriedade é visível não apenas dentro da classe que a define, mas também a todas as subclasses. Ao contrário da palavra-chave `protected` na linguagem de programação Java, a palavra-chave `protected` no ActionScript 3.0 não torna a propriedade visível para todas as outras classes no mesmo pacote. No ActionScript 3.0, apenas as subclasses podem acessar uma propriedade declarada com a palavra-chave `protected`. Além disso, uma propriedade protegida será visível para uma subclasse, se a subclasse estiver no mesmo pacote da classe base ou em um pacote diferente.

Para limitar a visibilidade de uma propriedade para o pacote no qual ela está definida, use a palavra-chave `internal` ou não use nenhum especificador de controle de acesso. O especificador de controle de acesso `internal` é o especificador padrão aplicado quando um não está especificado. Uma propriedade marcada como `internal` só é herdada por uma subclasse que reside no mesmo pacote.

Você pode usar o exemplo a seguir para ver como cada um dos especificadores de controle de acesso afeta a herança entre limites de pacotes. O código a seguir define uma classe de aplicativo principal denominada AccessControl e duas outras classes denominadas Base e Extender. A classe Base está em um pacote denominado foo e a classe Extender, que é uma subclasse da classe Base, está em um pacote denominado bar. A classe AccessControl importa apenas a classe Extender e cria uma ocorrência de Extender que tenta acessar uma variável denominada `str` definida na classe Base. A variável `str` é declarada como `public` para que o código seja compilado e executado, conforme mostrado no seguinte trecho:

```
// 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 como os outros especificadores de controle de acesso afetam a compilação e a execução do exemplo anterior, altere o especificador de controle de acesso da variável `str` para `private`, `protected` ou `internal` após excluir ou comentar a linha seguinte da classe `AccessControl`:

```
trace(myExt.str);// error if str is not public
```

Substituição de variáveis não permitidas

As propriedades declaradas com as palavras-chave `var` ou `const` são herdadas, mas não podem ser substituídas. Substituir uma propriedade significa redefini-la em uma subclasse. O único tipo de propriedade que pode ser substituído são os acessadores get e set (propriedades declaradas com a palavra-chave `function`). Embora não seja possível substituir uma variável de ocorrência, você pode alcançar funcionalidade semelhante criando os métodos getter e setter para a variável de ocorrência e substituindo os métodos.

Substituição de métodos

Substituir um método significa redefinir o comportamento de um método herdado. Métodos estáticos não são herdados e não podem ser substituídos. No entanto métodos de ocorrência são herdados por subclasses e podem ser substituídos desde que os dois seguintes critérios sejam atendidos:

- O método da ocorrência não é declarado com a palavra-chave `final` na classe base. Quando usada com um método da ocorrência, a palavra-chave `final` indica a intenção do programador de impedir que as subclasses substituam o método.
- O método da ocorrência não é declarado com o especificador de controle de acesso `private` na classe base. Se um método estiver marcado como `private` na classe base, não haverá necessidade de usar a palavra-chave `override` ao definir um método nomeado de maneira idêntica na subclasse, porque o método da classe base não é visível para a subclasse.

  Para substituir um método da ocorrência que atenda a esses critérios, a definição do método na subclasse deve usar a palavra-chave `override` e deve corresponder à versão da superclasse do método das seguintes maneiras:

- O método de substituição deve ter o mesmo nível de controle de acesso do método da classe base. Métodos marcados como internos têm o mesmo nível de controle de acesso que os métodos que não têm nenhum especificador de controle de acesso.
- O método de substituição deve ter o mesmo número de parâmetros que o método da classe base.
- Os parâmetros do método de substituição devem ter as mesmas anotações de tipo de dados que os parâmetros do método da classe base.
- O método de substituição deve ter o mesmo tipo de retorno que o método da classe base.

No entanto os nomes dos parâmetros no método de substituição não precisam corresponder aos nomes dos parâmetros na classe base, desde que o número de parâmetros e o tipo de dados de cada parâmetro correspondam.

A instrução super

Ao substituir um método, os programadores sempre querem aumentar o comportamento do método da superclasse que estão substituindo, em vez de substituir completamente o comportamento. Isso requer um mecanismo que permita que um método em uma subclasse chame a versão da superclasse de si próprio. A instrução `super` fornece esse mecanismo, já que ela contém uma referência à superclasse imediata. O exemplo a seguir define uma classe denominada Base que contém um método denominado `thanks()` e uma subclasse da classe Base denominada Extender que substitui o método `thanks()`. O método `Extender.thanks()` usa a instrução `super` para chamar `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";
    }
}
```

Substituição de getters e setters

Embora não seja possível substituir variáveis definidas em uma superclasse, você pode substituir getters e setters. Por exemplo, o código a seguir substitui um getter denominado `currentLabel` que é definido na classe MovieClip no 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;
        }
    }
}
```

A saída da instrução `trace()` no construtor da classe OverrideExample é `Override: null`, que mostra que o exemplo pôde substituir a propriedade `currentLabel` herdada.

Propriedades estáticas não herdadas

Propriedades estáticas não são herdadas por subclasses. Isso significa que as propriedades estáticas não podem ser acessadas por meio de uma ocorrência de uma subclasse. Uma propriedade estática pode ser acessada apenas por meio do objeto da classe no qual ela é definida. Por exemplo, o código a seguir define uma classe base denominada Base e uma subclasse denominada Extender que estende a Base. Uma variável estática denominada `test` é definida na classe Base. O código conforme escrito no fragmento a seguir, não é compilado no modo estrito e gera um erro em tempo de execução no modo padrão.

```
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 { }
```

A única maneira de acessar a variável estática `test` é por meio do objeto da classe, conforme mostrado no código a seguir:

```
Base.test;
```

No entanto é permitido definir uma propriedade da ocorrência usando o mesmo nome de uma propriedade estática. Essa propriedade da ocorrência pode ser definida na mesma classe que a propriedade estática ou em uma subclasse. Por exemplo, a classe Base no exemplo anterior podia ter uma propriedade da ocorrência denominada `test`. O código a seguir é compilado e executado porque a propriedade da ocorrência é herdada pela classe Extender. O código também será compilado e executado, se a definição da variável da ocorrência de teste for movida, mas não copiada, para a classe Extender.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Propriedades estáticas e a cadeia de escopos

Embora as propriedades estáticas não sejam herdadas, elas estão dentro da cadeia do escopo da classe que as define e em qualquer subclasse dessa classe. Como tal, diz-se que as propriedades estáticas estão *in scope* da classe na qual elas são definidas e em qualquer subclasse. Isso significa que uma propriedade estática pode ser acessada diretamente dentro do corpo da classe que a define e em qualquer subclasse dessa classe.

O exemplo a seguir modifica as classes definidas no exemplo anterior para mostrar que a variável estática `test` definida na classe Base está no escopo da classe Extender. Em outras palavras, a classe Extender pode acessar a variável estática `test` sem prefixar a variável com o nome da classe que define `test`.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

Se for definida uma propriedade de ocorrência que usa o mesmo nome que uma propriedade estática na mesma classe ou em uma superclasse, a propriedade de ocorrência terá precedência mais alta na cadeia do escopo. Diz-se que a propriedade da ocorrência *sombreia* a propriedade estática, o que significa que o valor da propriedade da ocorrência é usado no lugar do valor da propriedade estática. Por exemplo, o código a seguir mostra que se a classe Extender definir uma variável da ocorrência denominada `test`, a instrução `trace()` usará o valor da variável da ocorrência em vez do valor da variável 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
    }

}
```

Tópicos avançados

Histórico do suporte da OOP ao ActionScript

Como o ActionScript 3.0 foi criado sobre versões anteriores do ActionScript, é útil compreender como o modelo de objeto do ActionScript evoluiu. O ActionScript começou como um mecanismo de script simples para versões anteriores do Flash Professional. Depois, os programadores começaram a criar aplicativos cada vez mais complexos com o ActionScript. Em resposta às necessidades desses programadores, cada versão subseqüente adicionou recursos de linguagem que facilitam a criação de aplicativos complexos.

ActionScript 1.0

O ActionScript 1.0 é a versão da linguagem usada no Flash Player 6 e anterior. Mesmo na primeira fase de desenvolvimento, o modelo de objeto do ActionScript era baseado no conceito do objeto como um tipo de dados fundamental. Um objeto do ActionScript é um tipo de dados composto por um grupo de *propriedades*. Ao discutir o modelo de objeto, o termo *propriedades* inclui tudo o que está conectado a um objeto, como variáveis, funções ou métodos.

Embora essa primeira geração do ActionScript não ofereça suporte à definição de classes com uma palavra-chave `class`, é possível definir uma classe usando um tipo especial de objeto chamado objeto de protótipo. Em vez de usar uma palavra-chave `class` para criar uma definição de classe abstrata que você instancia em objetos concretos, como o faz em linguagens baseadas em classe, como Java e C++, as linguagens baseadas em protótipo como o ActionScript 1.0 usam um objeto existente como um modelo (ou protótipo) para outros objetos. Enquanto objetos em uma linguagem baseada em classe podem apontar para uma classe que serve como seu modelo, objetos em uma linguagem baseada em protótipo apontam para outro objeto, seu protótipo, que serve como seu modelo.

Para criar uma classe no ActionScript 1.0, defina uma função de construtor para essa classe. No ActionScript, as funções são objetos reais, não apenas definições abstratas. A função de construtor que você cria serve como o objeto de protótipo para ocorrências dessa classe. O código a seguir cria uma classe denominada Shape e define uma propriedade denominada `visible` que, por padrão, é definida como `true`:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Essa função de construtor define uma classe Shape que você pode instanciar com o operador `new`, da seguinte maneira:

```
myShape = new Shape();
```

Do mesmo modo como o objeto de função de construtor `Shape()` serve como protótipo para ocorrências da classe Shape, ele também pode servir como o protótipo para subclasses de Shape, isto é, outras classes que estendem a classe Shape.

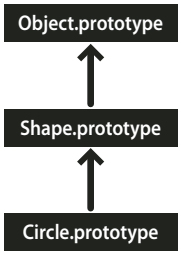
A criação de uma classe que é uma subclasse da classe Shape é um processo de duas etapas. Primeiro, crie a classe definindo uma função de construtor para a classe, da seguinte maneira:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Segundo, use o operador `new` para declarar que a classe Shape é o protótipo para a classe Circle. Por padrão, qualquer classe criada usa a classe Object como seu protótipo, o que significa que `Circle.prototype` atualmente contém um objeto genérico (uma ocorrência da classe Object). Para especificar que o protótipo de Circle é Shape em vez de Object, use o código a seguir para alterar o valor de `Circle.prototype` para que ele contenha um objeto Shape em vez de um objeto genérico:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

As classes Shape e Circle estão agora vinculadas em conjunto em um relacionamento de herança que é conhecido como a *cadeia de protótipos*. O diagrama ilustra os relacionamentos em uma cadeia de protótipos:



A classe base no final de cada cadeia de protótipos é a classe Object. A classe Object contém uma propriedade estática denominada `Object.prototype` que aponta para o objeto de protótipo base de todos os objetos criados no ActionScript 1.0. O próximo objeto no exemplo da cadeia de protótipos é o objeto Shape. Isso ocorre porque a propriedade `Shape.prototype` nunca foi definida explicitamente, portanto ela ainda mantém um objeto genérico (uma ocorrência da classe Object). O link final nessa cadeia é a classe Circle que está vinculada a seu protótipo, a classe Shape (a propriedade `Circle.prototype` mantém um objeto Shape).

Se você criar uma ocorrência da classe Circle, como no seguinte exemplo, a ocorrência herdará a cadeia de protótipos da classe Circle:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Lembre-se de que o exemplo incluía uma propriedade denominada `visible` como um membro da classe Shape. Neste exemplo, a propriedade `visible` não existe como parte do objeto `myCircle`, apenas como membro do objeto Shape, apesar da linha seguinte do código produzir `true`:

```
trace(myCircle.visible); // output: true
```

O tempo de execução pode verificar se o objeto `myCircle` herda a propriedade `visible` percorrendo a cadeia de protótipos. Ao executar este código, o tempo de execução primeiro pesquisa nas propriedades do objeto `myCircle` uma propriedade denominada `visible`, mas não encontra essa propriedade. Em seguida, ele verifica o objeto `Circle.prototype`, mas ainda não encontra uma propriedade denominada `visible`. Continuando na cadeia de protótipos, ele finalmente encontra a propriedade `visible` definida no objeto `Shape.prototype` e fornece o valor daquela propriedade.

Para fins de simplicidade, muitos dos detalhes e complicações da cadeia de protótipos são omitidos. No lugar, o objetivo é fornecer informações suficientes para ajudá-lo a entender o modelo de objetos do ActionScript 3.0.

ActionScript 2.0

O ActionScript 2.0 introduziu novas palavras-chave, como `class`, `extends`, `public`e `private`, que permitiram definir classes de uma maneira que seja familiar a qualquer pessoa que trabalhe com linguagens baseadas em classes como Java e C++. É importante compreender que o mecanismo da herança subjacente não foi alterado entre o ActionScript 1.0 e o ActionScript 2.0. O ActionScript 2.0 simplesmente adicionou uma nova sintaxe para definir classes. A cadeia de protótipos funciona da mesma maneira nas duas versões da linguagem.

A nova sintaxe introduzida pelo ActionScript 2.0, mostrada no trecho a seguir, permite definir classes de uma maneira que é considerada intuitiva por muitos programadores:

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

Observe que o ActionScript 2.0 também introduziu anotações de tipo para uso com verificação de tipos em tempo de compilação. Isso permite declarar que a propriedade `visible` no exemplo anterior deve conter apenas um valor booleano. A nova palavra-chave `extends` também simplifica o processo de criação de uma subclasse. No exemplo a seguir, o processo em duas etapas necessário no ActionScript 1.0 é executado em uma etapa com a palavra-chave `extends`:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

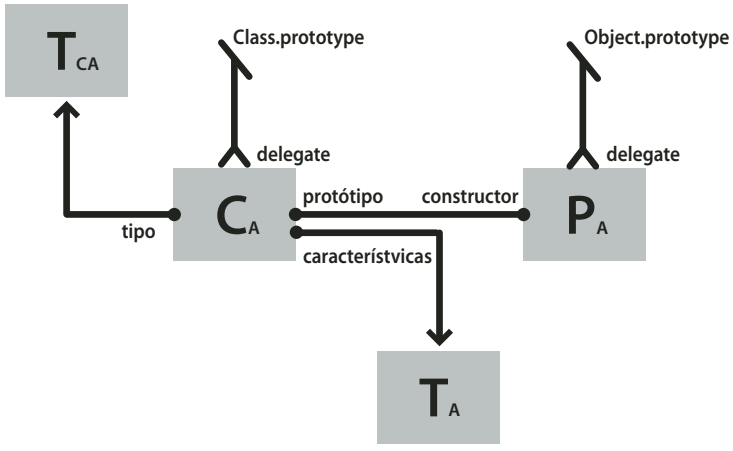
O construtor está agora declarado como parte da definição da classe e as propriedades de classes `id` e `radius` também devem ser declaradas explicitamente.

O ActionScript 2.0 também adicionou suporte para a definição de interfaces, o que permite refinar ainda mais os programas orientados a objetos com protocolos definidos formalmente para comunicação entre objetos.

Objeto de classe do ActionScript 3.0

Um paradigma comum da programação orientada a objetos, mais freqüentemente associado a Java e C++, usa classes para definir tipos de objetos. Linguagens de programação que adotam esse paradigma também tendem a usar classes para construir ocorrências do tipo de dados que a classe define. O ActionScript usa classes para duas dessas finalidades, mas suas raízes como uma linguagem com base em protótipo adicionam uma característica interessante. O ActionScript cria para cada definição de classe um objeto de classe especial que permite o compartilhamento do comportamento e do estado. No entanto, para muitos programadores do ActionScript, esta distinção não pode ter nenhuma implicação de codificação prática. O ActionScript 3.0 é projetado de forma que você possa criar aplicativos do ActionScript orientados a objetos sofisticados sem usar, ou mesmo compreender, esses objetos de classes especiais.

O diagrama a seguir mostra a estrutura de um objeto de classe que representa uma classe simples denominada A que é definida com a instrução `class A {}`:



Cada retângulo no diagrama representa um objeto. Cada objeto no diagrama tem um caractere subscrito A para representar que ele pertence à classe A. O objeto de classe (CA) contém referências a vários outros objetos importantes. Um objeto com características da ocorrência (TA) armazena as propriedades da ocorrência que são definidas dentro de uma definição de classe. Um objeto de características da classe (TCA) representa o tipo interno da classe e armazena as propriedades estáticas definidas pela classe (o caractere subscrito C representa a "classe"). O objeto de protótipo (PA) sempre significa o objeto da classe ao qual ele era originalmente anexado por meio da propriedade `constructor`.

Objeto de características

O objeto de características, novo no ActionScript 3.0, foi implementado tendo em mente o desempenho. Em versões anteriores do ActionScript, a pesquisa de nome era um processo demorado pois o Flash Player percorria a cadeia de protótipos. No ActionScript 3.0, a pesquisa de nome é muito mais eficiente e menos demorada, porque as propriedades herdadas são copiadas das superclasses no objeto de características de subclasses.

O objeto de características não pode ser acessado diretamente pelo código do programador, mas sua presença pode ser sentida pelos aprimoramentos no desempenho e no uso de memória. O objeto de características fornece ao AVM2 informações detalhadas sobre o layout e o conteúdo de uma classe. Com esse conhecimento, o AVM2 pode reduzir significativamente o tempo de execução, porque pode gerar freqüentemente instruções de máquina diretas para acessar propriedades ou chamar métodos diretamente sem uma pesquisa de nome demorada.

Graças ao objeto de características, uma superfície de memória do objeto pode ser significativamente menor do que a de um objeto semelhante em versões anteriores do ActionScript. Por exemplo, se uma classe estiver selada (isto é, a classe não está declarada dinâmica), uma ocorrência da classe não precisará de uma tabela hash para propriedades adicionadas dinamicamente, e poderá manter um pouco mais do que um ponteiro para os objetos de características e alguns slots para as propriedades fixas definidas na classe. Como resultado, um objeto que exigia 100 bytes de memória no ActionScript 2.0 pode exigir apenas 20 bytes de memória no ActionScript 3.0.

Nota: *O objeto de características é um detalhe da implementação interna, e não há nenhuma garantia de que ele não seja alterado ou mesmo que desapareça em versões futuras do ActionScript.*

Objeto de protótipo

Cada objeto de classe do ActionScript tem uma propriedade denominada `prototype`, que é uma referência ao objeto de protótipo da classe. O objeto de protótipo é um herança das raízes do ActionScript como linguagem com base em protótipo. Para obter mais informações, consulte Histórico do suporte da OOP no ActionScript.

A propriedade `prototype` é somente leitura, o que significa que não pode ser modificada para apontar para objetos diferentes. Ela é diferente da propriedade `prototype` da classe em versões anteriores do ActionScript, em que o protótipo podia ser reatribuído para que apontasse para uma classe diferente. Embora a propriedade `prototype` seja somente leitura, o objeto de protótipo ao qual ela faz referência não é. Em outras palavras, novas propriedades podem ser adicionadas ao objeto de protótipo. Propriedades adicionadas ao objeto de protótipo são compartilhadas entre todas as ocorrências da classe.

A cadeia de protótipos, que era o único mecanismo de herança em versões anteriores do ActionScript, serve apenas uma função secundária no ActionScript 3.0. O mecanismo de herança principal, herança de propriedade fixa, é manipulado internamente pelo objeto de características. Uma propriedade fixa é uma variável ou método que é definida como parte de uma definição de classe. A herança de propriedade fixa também é de chamada herança de classe, porque ela é o mecanismo de herança associado a palavras-chave, como `class`, `extends` e `override`.

A cadeia de protótipos fornece um mecanismo de herança alternativa que é mais dinâmico do que a herança de propriedade fixa. Você pode adicionar propriedades a um objeto de protótipo de classe não apenas como parte da definição da classe, mas também em tempo de execução por meio da propriedade `prototype` do objeto de classe. No entanto, observe que se você definir o compilador como modo estrito, talvez não seja possível acessar propriedades adicionadas a um objeto de protótipo, a não ser que você declare uma classe com a palavra-chave `dynamic`.

Um bom exemplo de uma classe com várias propriedades anexadas ao objeto de protótipo é a classe Object. Os métodos `toString()` e `valueOf()` da classe Object são realmente funções atribuídas às propriedades do objeto de protótipo da classe Object. A seguir está um exemplo de como pode ser a aparência da declaração desses métodos, em teoria, (a implementação real é um pouco diferente, por causa dos detalhes da implementação):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Conforme mencionado anteriormente, é possível anexar uma propriedade a um objeto de protótipo de classe fora da definição de classe. Por exemplo, o método `toString()` também pode ser definido fora da definição da classe Object, da seguinte maneira:

```
Object.prototype.toString = function()
{
    // statements
};
```

No entanto, ao contrário da herança de propriedade fixa, a herança de protótipo não exigirá a palavra-chave `override`, se você desejar redefinir um método em uma subclasse. Por exemplo. se você desejar redefinir o método `valueOf()` em uma subclasse da classe Object, terá três opções. Primeiro, você pode definir um método `valueOf()` no objeto de protótipo de subclasse dentro da definição de classe. O código a seguir cria uma subclasse de Object denominada Foo e redefine o método `valueOf()` no objeto de protótipo de Foo como parte da definição de classe. Como cada classe é herdada de Object, não é necessário usar a palavra-chave `extends`.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Segundo, você pode definir um método `valueOf()` no objeto de protótipo de Foo fora da definição de classe, conforme mostrado no código a seguir:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Terceiro, você pode definir uma propriedade fixa denominada `valueOf()` como parte da classe Foo. Essa técnica é diferente das outras já que ela mescla herança de propriedade fixa com herança de protótipo. Qualquer subclasse de Foo que precise redefinir `valueOf()` deve usar a palavra-chave `override`. O código a seguir mostra `valueOf()` definido como uma propriedade fixa em Foo:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

Espaço para nomes AS3

A existência de dois mecanismos de herança separados, herança de propriedade fixa e herança de protótipo, cria uma desafio de compatibilidade interessante em relação às propriedades e métodos das classes principais. A compatibilidade com a especificação de linguagem do ECMAScript na qual o ActionScript é baseado exige o uso de herança de protótipo, o que significa que as propriedades e métodos de uma classe principal são definidos no objeto de protótipo dessa classe. Por outro lado, a compatibilidade com o ActionScript 3.0 exige o uso de herança de propriedade fixa, o que significa que as propriedades e métodos de uma classe principal são definidos na definição da classe usando as palavras-chave `const`, `var` e `function`. Além disso, o uso de propriedades fixas, em vez das versões do protótipo pode levar a aumentos significativos no desempenho em tempo de execução.

O ActionScript 3.0 resolve esse problema usando herança de protótipo e herança de propriedade fixa para as classes principais. Cada classe principal contém dois conjuntos de propriedades e métodos. Um conjunto é definido no objeto de protótipo para compatibilidade com a especificação do ECMAScript, e o outro conjunto é definido com propriedades fixas e o espaço para nomes AS3 para compatibilidade com o ActionScript 3.0.

O espaço para nomes AS3 fornece um mecanismo conveniente para escolher entre os dois conjuntos de propriedades e métodos. Se você não usar o espaço para nomes AS3, uma ocorrência de uma classe principal herdará as propriedades e métodos definidos no objeto de protótipo da classe principal. Se você decidir usar o espaço para nomes AS3, uma ocorrência de uma classe principal herdará as versões do AS3, porque as propriedades fixas são sempre preferidas sobre as propriedades de protótipo. Em outras palavras, sempre que uma propriedade fixa estiver disponível, ela será sempre usada no lugar de uma propriedade de protótipo nomeada de forma idêntica.

Você pode usar seletivamente a versão do espaço para nomes AS3 de uma propriedade ou método qualificando-a com o espaço para nomes AS3. Por exemplo, o código a seguir usa a versão do AS3 do método `Array.pop()`:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Como alternativa, você pode usar a diretiva `use namespace` para abrir o espaço para nomes AS3 para todas as definições dentro de um bloco de código. Por exemplo, o código a seguir usa a diretiva `use namespace` para abrir o espaço para nomes AS3 para os métodos `pop()` e `push()`:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

O ActionScript 3.0 também fornece opções de compilador para cada conjunto de propriedades para que você possa aplicar o espaço para nomes AS3 ao programa inteiro. A opção de compilador `-as3` representa o espaço para nomes AS3, e a opção de compilador `-es` representa a opção de herança de protótipo (`es` representa o ECMAScript). Para abrir o espaço para nomes AS3 para o programa inteiro, defina a opção de compilador `-as3` como `true` e a opção de compilador `-es` como `false`. Para usar as versões de protótipo, defina as opções do compilador como os valores opostos. As configurações do compilador padrão do Flash Builder e do Flash Professional são `-as3 = true` e `-es = false`.

Se você planejar estender qualquer uma das classes principais e substituir qualquer método, deverá compreender como o espaço para nomes AS3 pode afetar o modo como você deve declarar um método substituído. Se você estiver usando o espaço para nomes AS3, qualquer substituição de método de um método da classe principal também deve usar o espaço para nomes AS3 juntamente com o atributo `override`. Se não estiver usando o espaço para nomes AS3 e desejar redefinir um método da classe principal em uma subclasse, você não deverá usar o espaço para nomes AS3 ou a palavra-chave `override`.

Exemplo: GeometricShapes

O aplicativo de amostra GeometricShapes mostra como vários conceitos e recursos orientados a objetos podem ser aplicados usando o ActionScript 3.0, inclusive:

- Definição de classes
- Extensão de classes
- Polimorfismo e a palavra-chave `override`
- Definição, extensão e implementação de interfaces

Ele também inclui um "método de fábrica" que cria ocorrências de classes, mostrando como declarar um valor de retorno como uma ocorrência de uma interface, e usar esse objeto retornado de uma maneira genérica.

Para obter os arquivos do aplicativo desta amostra, consulte www.adobe.com/go/learn_programmingAS3samples_flash_br. Os arquivos do aplicativo GeometricShapes podem ser encontrados na pasta Amostras/GeometricShapes. O aplicativo consiste nos seguintes arquivos:

Arquivo	Descrição
GeometricShapes.mxml ou GeometricShapes.fla	O arquivo principal do aplicativo no Flash (FLA) ou no Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	A interface base que define métodos a serem implementados por todas as classes do aplicativo GeometricShapes.
com/example/programmingas3/geometricshapes/IPolygon.as	Uma interface que define métodos a serem implementados pelas classes do aplicativo GeometricShapes que têm vários lados.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Um tipo de forma geométrica que tem lados de comprimento igual prolongados simetricamente em torno do centro da forma.
com/example/programmingas3/geometricshapes/Circle.as	Um tipo de forma geométrica que define um círculo.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Uma subclasse de RegularPolygon que define um triângulo com todos os lados com o mesmo comprimento.
com/example/programmingas3/geometricshapes/Square.as	Uma subclasse de RegularPolygon que define um retângulo com os quatro lados com o mesmo comprimento.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Uma classe que contém um método de fábrica para criar formas com tamanho e tipo especificados.

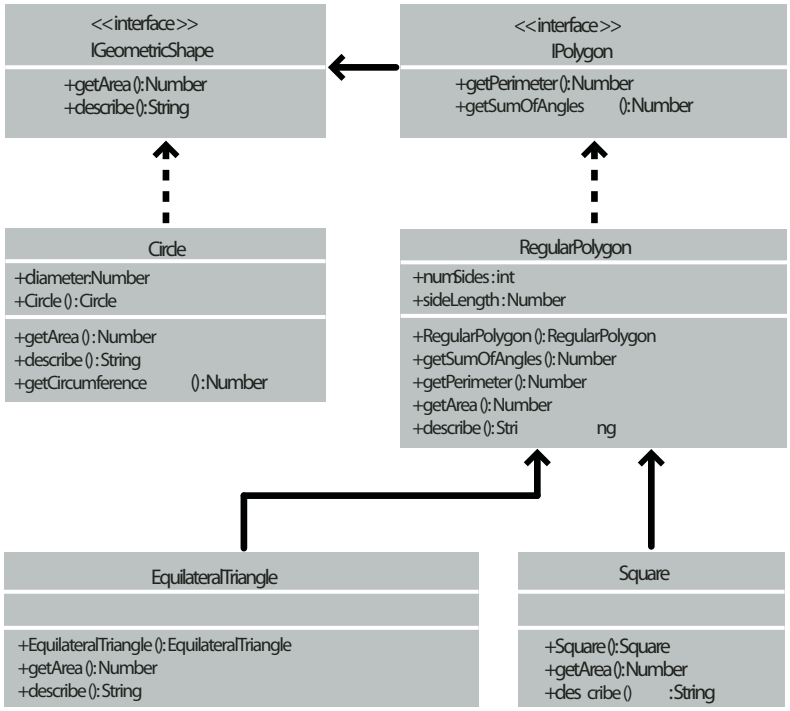
Definição das classes GeometricShapes

O aplicativo GeometricShapes permite que o usuário especifique um tipo de forma geométrica e um tamanho. Em seguida, ele responde com uma descrição da forma, sua área e a distância em torno de seu perímetro.

A interface de usuário do aplicativo é trivial, incluindo alguns controles para seleção do tipo de forma, configuração do tamanho e exibição da descrição. A parte mais interessante desse aplicativo está sob a superfície, na estrutura das classes e das próprias interfaces.

Esse aplicativo trata de formas geométricas, mas não as exibe graficamente.

As classes e interfaces que definem as formas geométricas neste exemplo são mostradas no diagrama a seguir que usa a notação UML (Linguagem de modelação unificada):



Classes de exemplo do GeometricShapes

Definição do comportamento comum com interfaces

Este aplicativo GeometricShapes trata de três tipos de formas: círculos, quadrados e triângulos eqüiláteros. A estrutura de classe GeometricShapes começa com uma interface muito simples, IGeometricShape, que lista métodos comuns para todos os três tipos de formas:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

A interface define dois métodos: o método `getArea()`, que calcula e retorna a área da forma, e o método `describe()`, que monta uma descrição de texto das propriedades da forma.

Também é desejável saber também a distância em torno do perímetro de cada forma. No entanto, o perímetro de um círculo é chamado de circunferência, e é calculado de uma maneira exclusiva, portanto o comportamento diverge daquele de um triângulo ou de um quadrado. Ainda há semelhança suficiente entre triângulos, quadrados e outros polígonos, portanto faz sentido definir uma nova classe de interface só para eles: IPolygon. A interface IPolygon também é muito simples, conforme mostrado aqui:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Essa interface define dois métodos comuns a todos os polígonos: o método `getPerimeter()` que mede a distância combinada de todos os lados e o método `getSumOfAngles()` que adiciona todos os ângulos internos.

A interface IPolygon estende a interface IGeometricShape, o que significa que qualquer classe que implemente a interface IPolygon deve declarar os quatro métodos, dois da interface IGeometricShape e dois da interface IPolygon.

Definição das classes Shape

Depois que você tiver uma boa idéia sobre os métodos comuns a cada tipo de forma, você pode definir as próprias classes. Em termos da quantidade dos métodos precisam ser implementados, a forma mais simples é a da classe Circle, mostrada aqui:

```
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;
        }
    }
}
```

A classe Circle implementa a interface IGeometricShape, portanto ela deve fornecer código para os métodos `getArea()` e `describe()`. Além disso, ela define o método `getCircumference()` que é exclusivo à classe Circle. A classe Circle também declara uma propriedade, `diameter` que não é encontrada nas outras classes de polígonos.

Os dois outros tipos de formas, quadrados e triângulos eqüiláteros, têm algumas outras coisas em comum: cada um deles têm lados com o mesmo comprimento e há fórmulas comuns que você pode usar para calcular o perímetro e a soma dos ângulos internos dos dois. Na verdade, essas fórmulas comuns são aplicadas a todos os outros polígonos regulares que você definir no futuro.

A classe RegularPolygon é a superclasse das classes Square e EquilateralTriangle. Uma superclasse permite definir métodos comuns em um lugar, portanto você não precisa defini-los separadamente em cada subclasse. Este é o código da classe RegularPolygon:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
```

```
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

Primeiro, a classe RegularPolygon declara duas propriedades comuns a todos os polígonos regulares: o comprimento de cada lado (a propriedade `sideLength`) e o número de lados (a propriedade `numSides`).

A classe RegularPolygon implementa a interface IPolygon e declara os quatro métodos da interface IPolygon. Ela implementa dois desses, os métodos `getPerimeter()` e `getSumOfAngles()`, usando fórmulas comuns.

Como a fórmula do método `getArea()` é diferente de forma para forma, a versão da classe base do método não pode incluir a lógica comum que pode ser herdada pelos métodos da subclasse. Em vez disso, ele simplesmente retorna um valor padrão 0 para indicar que a área não foi calculada. Para calcular a área de cada forma corretamente, as próprias subclasses da classe RegularPolygon precisam substituir o método `getArea()`.

O seguinte código da classe EquilateralTriangle mostra como o método `getArea()` é substituído:

```
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;
        }
    }
}
```

A palavra-chave `override` indica que o método `EquilateralTriangle.getArea()` substitui intencionalmente o método `getArea()` da superclasse RegularPolygon. Quando o método `EquilateralTriangle.getArea()` é chamado, ele calcula a área usando a fórmula do código anterior, e o código no método `RegularPolygon.getArea()` nunca é executado.

Em comparação, a classe EquilateralTriangle não define sua própria versão do método `getPerimeter()`. Quando o método `EquilateralTriangle.getPerimeter()` é chamado, a chamada percorre a cadeia de herança e executa o código no método `getPerimeter()` da superclasse RegularPolygon.

O construtor `EquilateralTriangle()` usa a instrução `super()` para chamar explicitamente o construtor `RegularPolygon()` de sua superclasse. Se os dois construtores tivessem o mesmo conjunto de parâmetros, você poderia omitir completamente o construtor `EquilateralTriangle()` e o construtor `RegularPolygon()` seria executado. No entanto, o construtor `RegularPolygon()` precisa de um parâmetro extra, `numSides`. Portanto, o construtor `EquilateralTriangle()` chama `super(len, 3)` que passa o parâmetro de entrada `len` e o valor 3 para indicar que o triângulo tem três lados.

O método `describe()` também usa a instrução `super()`, mas de forma diferente. Ele a utiliza para invocar a versão da superclasse RegularPolygon do método `describe()`. O método `EquilateralTriangle.describe()` define primeiro a variável da string `desc` como uma instrução sobre o tipo da forma. Em seguida, ele obtém os resultados do método `RegularPolygon.describe()` chamando `super.describe()` e anexa esse resultado à string `desc`.

A classe Square não é descrita em detalhes aqui, mas é semelhante à classe EquilateralTriangle, fornecendo um construtor e suas próprias implementações dos métodos `getArea()` e `describe()`.

Polimorfismo e o método de fábrica

Um conjunto de classes que faz bom uso de interfaces e herança pode ser usado de muitas maneiras interessantes. Por exemplo, todas essas classes de formas descritas até agora implementam a interface IGeometricShape ou estendem uma superclasse que o faz. Portanto, se você definir uma variável como sendo uma ocorrência de IGeometricShape, não precisará saber se ela é realmente uma ocorrência das classes Circle ou Square para chamar seu método `describe()`.

O código a seguir mostra como isso funciona:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Quando `myShape.describe()` é chamado, ele executa o método `Circle.describe()`, porque embora a variável esteja definida como uma ocorrência da interface IGeometricShape, Circle é sua classe subjacente.

Este exemplo mostra o princípio do polimorfismo em ação: a mesma chamada de método exata resulta na execução de código diferente, dependendo da classe do objeto cujo método está sendo chamado.

O aplicativo GeometricShapes aplica esse tipo de polimorfismo com base em interface usando uma versão simplificada de um padrão de design conhecido como método de fábrica. O termo *método de fábrica* significa uma função que retorna um objeto cujo tipo de dados subjacentes ou conteúdo pode ser diferente, dependendo do contexto.

A classe GeometricShapeFactory mostrada aqui define um 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();
        }
    }
}
```

O método de fábrica `createShape()` permite que construtores de subclasses de formas definam os detalhes das ocorrências que eles criam, enquanto retornam os novos objetos como ocorrências de IGeometricShape para que eles possam ser manipulados pelo aplicativo de maneira mais geral.

O método `describeShape()` no exemplo anterior mostra como um aplicativo pode usar o método de fábrica para obter uma referência genérica para um objeto mais específico. O aplicativo pode obter a descrição de um objeto Circle criado recentemente, como este:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

Em seguida, o método `describeShape()` chama o método de fábrica `createShape()` com os mesmo parâmetros, armazenando o novo objeto Circle em uma variável estática denominada `currentShape` que foi digitada como um objeto IGeometricShape. Em seguida, o método `describe()` é chamado no objeto `currentShape`, e essa chamada é resolvida automaticamente para executar o método `Circle.describe()` retornando uma descrição detalhada do círculo.

Aprimoramento do aplicativo de amostra

O poder real das interfaces e da herança torna-se aparente quando você aprimora ou altera o aplicativo.

Suponha que você deseja adicionar uma nova forma, um pentágono, a este aplicativo de amostra. Você criaria uma classe Pentagon que estende a classe RegularPolygon e define suas próprias versões dos métodos `getArea()` e `describe()`. Em seguida, adicionaria uma nova opção de Pentagon à caixa de combinação na interface de usuário do aplicativo. Mas isso é tudo. A classe Pentagon obteria automaticamente a funcionalidade dos métodos `getPerimeter()` e `getSumOfAngles()` da classe RegularPolygon por herança. Como ela é herdada de uma classe que implementa a interface IGeometricShape, uma ocorrência Pentagon também pode ser tratada como uma ocorrência IGeometricShape. Isso significa que para adicionar um novo tipo de forma, não é preciso alterar a assinatura de nenhum dos métodos na classe GeometricShapeFactory (e, conseqüentemente, também não é preciso alterar nenhum código que use a classe GeometricShapeFactory).

Talvez você queira adicionar uma classe Pentagon ao exemplo de Geometric Shapes como um exercício, para ver como as interfaces e a herança podem facilitar a carga de trabalho da adição de novos recursos a um aplicativo.