ACTIONSCRIPT® 3.0 の学習

法律上の注意

法律上の注意については、http://help.adobe.com/ja_JP/legalnotices/index.html を参照してください。

目次

第 1 章：ActionScript 3.0 の概要

ActionScript について

ActionScript は Adobe® Flash® Player および Adobe® AIR™ のランタイム環境用のプログラム言語で、Flash、Flex および AIR コンテンツおよびアプリケーションでのユーザー操作やデータ処理などを可能にします。

ActionScript は、Flash Player および AIR の一部である ActionScript 仮想マシン（AVM）により実行されます。ActionScript コードは、通常、コンパイラーによってバイトコード形式に変換されます。（バイトコードは、コンピューターによって書き込まれ、認識されるプログラミング言語の一種です。）コンパイラーの例として、Adobe® Flash® Professional や Adobe® Flash® Builder™ に内蔵されているもの、Adobe® Flex™ SDK で使用可能なものがあります。バイトコードは、Flash Player と AIR によって実行される SWF ファイルに埋め込まれます。

ActionScript 3.0 は、オブジェクト指向プログラミングの基礎知識を持つ開発者にはなじみのある堅牢なプログラミングモデルを提供します。ActionScript の以前のバージョンから改善された ActionScript 3.0 の一部の主要な機能は次のとおりです。

- AVM2 という新しい ActionScript 仮想マシン。新しいバイトコード命令セットを使用し、パフォーマンスを大幅に向上させます。
- コンパイラーの以前のバージョンより詳細な最適化を実行するより現代的なコンパイラーコードベース
- 拡張および改良されたアプリケーションプログラミングインターフェイス（API）。オブジェクトの低レベルコントロールと真のオブジェクト指向モデルを備えています。
- ECMAScript for XML（E4X）仕様（ECMA-357 edition 2）に準拠した XML API。E4X は、言語のネイティブデータ型として XML を追加する ECMAScript 言語拡張です。
- ドキュメントオブジェクトモデル（DOM）Level 3 Events 仕様に準拠したイベントモデル。

ActionScript 3.0 の利点

ActionScript 3.0 は、旧バージョンの ActionScript のスクリプト機能を上回っています。大容量のデータセットおよびオブジェクト指向の再利用可能なコードベースを使用する非常に複雑なアプリケーションを容易に作成できるように設計されています。ActionScript 3.0 は、Adobe Flash Player で実行されるコンテンツには必要ありませんが、新しい仮想マシン AVM2（ActionScript 3.0 仮想マシン）でのみ可能なパフォーマンスの向上を実現します。ActionScript 3.0 コードは、従来の ActionScript コードより最大で 10 倍高速に実行できます。

古いバージョンの ActionScript 仮想マシン AVM1 は、ActionScript 1.0 および ActionScript 2.0 のコードを実行します。Flash Player 9 および 10 では、後方互換性を維持するために AVM1 がサポートされています。

ActionScript 3.0 の新機能

ActionScript 3.0 には ActionScript 1.0 と 2.0 と同様のクラスと機能が多く含まれていますが、旧バージョンの ActionScript とはそのアーキテクチャや概念が異なります。ActionScript 3.0 の強化点には、コア言語の新機能および低レベルオブジェクトの制御を高める改良された API があります。

コア言語機能

コア言語は、ステートメント、式、条件、ループ、型などのプログラミング言語の基本的な要素を定義します。ActionScript 3.0 には、開発プロセスの効率化を促す多くの新機能が備わっています。

ランタイム例外

ActionScript 3.0 では、旧バージョンの ActionScript より多くのエラー状態が報告されます。ランタイム例外を一般的なエラー状態に使用すると、デバッグの操作性が向上し、エラーを確実に処理するアプリケーションを開発することができます。ランタイムエラーには、ソースファイルおよび行番号情報で注釈を付けたスタックトレースがあり、エラーをすばやく特定できます。

ランタイム型

ActionScript 3.0 では、型情報は実行時に維持されます。型情報はランタイム型チェックの実行に使用され、システムの型の安全性を向上させます。また、型情報はネイティブのマシン表現で変数を表すためにも使用され、パフォーマンスを向上させ、メモリの使用量を削減します。これに対して、ActionScript 2.0 では型注釈は主に開発者を補助するもので、実行時にすべての値が動的に入力されました。

sealed クラス

ActionScript 3.0 には、sealed クラスの概念が導入されています。sealed クラスには、コンパイル時に定義された一定のプロパティとメソッドだけが含まれ、その他のプロパティやメソッドを追加することはできません。実行時にクラスを変更できないため、より厳密にコンパイル時のチェックを行うことができ、堅牢性の高いプログラムを作成できます。また、各オブジェクトインスタンスの内部ハッシュテーブルが不要になり、メモリの使用量を削減できます。dynamic キーワードを使用してダイナミッククラスも使用できます。ActionScript 3.0 のクラスはすべてデフォルトで sealed クラスになりますが、dynamic キーワードを使用してダイナミッククラスとして宣言することもできます。

メソッドクロージャ

ActionScript 3.0 では、元のオブジェクトインスタンスを自動的に記憶するメソッドクロージャが有効です。この機能はイベント処理に役立ちます。ActionScript 2.0 では、メソッドクロージャに抽出元のオブジェクトインスタンスが記憶されず、メソッドクロージャが呼び出されたときに予期しない動作が発生しました。

ECMAScript for XML（E4X）

ActionScript 3.0 は、最近 ECMA-357 として標準化された ECMAScript for XML（E4X）を実装しています。E4X は、XML を操作するための自然で滑らかな言語コンストラクトを提供します。従来の XML 解析 API とは異なり、E4X の XML は、その言語のネイティブデータ型のように動作します。E4X は、必要なコード量を大幅に削減して、XML を操作するアプリケーションの開発を簡略化します。

ECMA の E4X 仕様を確認するには、www.ecma-international.org を参照してください。

正規表現

ActionScript 3.0 では正規表現がネイティブにサポートされているので、ストリングをすばやく検索し、操作できます。ActionScript 3.0 は、ECMAScript Edition 3 言語仕様（ECMA-262）で定義されている正規表現のサポートを実装しています。

名前空間

名前空間は、宣言（public、private、protected）の可視性を制御するために使用する通常のアクセス指定子に似ています。名前空間は、カスタムアクセス指定子として動作し、任意の名前を指定できます。名前空間には、競合を回避するために URI（Universal Resource Identifier）が割り当てられています。また、E4X を使用するときに XML 名前空間を表すためにも使用します。

新しいプリミティブ型

ActionScript 3.0 には、Number、int、uint という 3 つの数値型があります。Number 型は、倍精度浮動小数点数を表します。int 型は、ActionScript コードが CPU の高速整数演算機能を活用できる 32 ビット符号付き整数で、整数が使用されるループカウンターや変数に便利です。uint 型は符号なしの 32 ビット整数型で、RGB カラー値、バイトカウントなどに便利です。ActionScript 2.0 では、使用できる数値型は Number 型のみでした。

API 機能

ActionScript 3.0 の API には、オブジェクトを低いレベルで制御できる多数のクラスが追加されています。この言語のアーキテクチャは、以前のバージョンより直観的に設計されています。新しく追加されたクラスが多いため、すべてを詳しく説明はしませんが、大きく変更されたいくつかについて次の節で取り上げます。

DOM3 イベントモデル

DOM3（Document Object Model Level 3）イベントモデルは、イベントメッセージを生成および処理するための標準的な方法を提供します。このイベントモデルは、アプリケーション内のオブジェクトが互いにやり取りし、その状態を保持し、変更に応答できるように設計されています。ActionScript 3.0 イベントモデルは World Wide Web Consortium DOM Level 3 イベント仕様に従って作成され、以前のバージョンの ActionScript のイベントシステムよりも、わかりやすく効率的なメカニズムを備えています。

イベントおよびエラーイベントは、flash.events パッケージにあります。Flash Professional コンポーネントと Flex フレームワークで同じイベントモデルを使用するので、イベントシステムが Flash Platform 全体で統一されます。

表示リスト API

表示リスト、つまりアプリケーションのビジュアルエレメントを含むツリーにアクセスするための API は、ビジュアルプリミティブを操作するためのクラスで構成されます。

Sprite クラスは、軽量の構築ブロックで、ユーザーインターフェイスコンポーネントなどのビジュアルエレメントの基本クラスとして適しています。Shape クラスは、未処理のベクターシェイプを表します。これらのクラスは、new 演算子で自然にインスタンス化され、いつでも動的に親に戻すことができます。

深度の管理は、自動化されました。オブジェクトの重ね順を指定し、管理するための新しいメソッドが用意されています。

動的なデータおよびコンテンツの処理

ActionScript 3.0 は、アプリケーションでアセットおよびデータをロードし処理するための直観的で API 全体で一貫したメカニズムを備えています。Loader クラスは、SWF ファイルおよびイメージアセットをロードするためのメカニズムを備え、ロードされたコンテンツに関する詳細情報にアクセスするための手段を提供します。URLLoader クラスには、データ駆動アプリケーションでテキストおよびバイナリデータをロードするための個別のメカニズムがあります。Socket クラスは、サーバーソケットに対して任意の形式でバイナリデータを読み取り、書き込む手段を提供します。

低レベルでのデータアクセス

様々な API を使用して、データに低レベルでアクセスできるようになりました。ダウンロード中のデータの場合、URLStream クラスを使用すると、データをダウンロード中に未処理のバイナリデータとしてデータにアクセスできます。ByteArray クラスを使用すると、バイナリデータの読み取り、書き込み、および操作を最適化できます。新しいサウンド API では、SoundChannel クラスと SoundMixer クラスからサウンドを詳細に制御できます。セキュリティ API は、SWF ファイルまたはロードされたコンテンツのセキュリティ権限に関する情報を提供し、セキュリティエラーをより適切に処理できるようにします。

テキストの操作

ActionScript 3.0 には、すべてのテキスト関連 API の flash.text パッケージが含まれています。TextLineMetrics クラスは、テキストフィールド内のテキスト行の詳細なメトリックを提供します。これは、ActionScript 2.0 の TextFormat.getTextExtent() メソッドに代わるものです。TextField クラスには、テキストフィールド内のテキスト 1 行また

は 1 文字に関する情報を提供する低レベルメソッドが含まれています。例えば、getCharBoundaries() メソッドは、文字の境界ボックスを表す矩形を返します。getCharIndexAtPoint() メソッドは、指定されたポイントにある文字のインデックスを返します。getFirstCharInParagraph() メソッドは、段落の最初の文字のインデックスを返します。行レベルのメソッドには、特定のテキスト行の文字数を返す getLineLength()、特定の行のテキストを返す getLineText() などがあります。Font クラスを使用すると、SWF ファイルの埋め込みフォントを管理できます。

さらに低いレベルでのテキスト制御のために、flash.text.engine パッケージのクラスは Flash Text Engine を作成します。このクラスのセットは、テキストに対する低レベルでの制御を提供し、テキストフレームワークとコンポーネントを作成するように設計されています。

第 2 章：ActionScript の使用について

プログラミングの基礎

ActionScript はプログラミング言語です。このため、ActionScript を学習する際に、コンピュータープログラミングの一般的な概念をいくつか理解しておくと役に立ちます。

コンピュータープログラムの内容

最初に、コンピュータープログラムとは何か、またどのような作業を実行するかについての概念を頭に描くことが大切です。コンピュータープログラムには、2 つの重要な側面があります。

- プログラムとは、コンピューターを実行するための一連の命令または手順です。
- 各手順では最終的に一部の情報またはデータの操作が伴います。

一般的な意味では、コンピュータープログラムは単にステップバイステップ形式の命令のリストであり、これをコンピューターに発行して、コンピューターはこれを 1 つずつ実行します。個々の命令はステートメントと呼ばれます。ActionScript では、ステートメントの最後にセミコロンを記述します。

実際には、プログラム内の所定の命令は、コンピューターのメモリ内に格納された一部のデータビットを操作するにすぎません。分かりやすい例として、2 つの数値を加算して、その結果をメモリに格納するようにコンピューターに命令することがあります。これよりも複雑な例としては、画面に描画された矩形を、画面上の別の場所に移動するようにプログラムを記述することがあります。コンピューターはその矩形に固有の情報、つまり、矩形が配置されている x 座標と y 座標や、矩形の幅と高さ、矩形の色などを記憶します。情報の各ビットはコンピューターのメモリ内に保存されます。矩形を別の場所に移動するプログラムは、「x 座標を 200 に変更し、y 座標を 150 に変更する」といった手順を含むことになります。つまり、x 座標と y 座標に別の値を指定します。内部的には、コンピューターがこのデータに対して何らかの処理を行うことで、これらの数値をコンピューター画面上に実際に表示されるイメージに変換しています。ただし、基本的には、「画面上の矩形の移動」プロセスが、実際にはコンピューターのメモリ内のデータビットを変更する作業に過ぎないことを理解するだけで十分です。

変数と定数

プログラミングとは、主に、コンピューターのメモリ内の情報を変更する作業です。したがって、プログラム内で個々の情報を表す手段を持つことが重要です。変数はコンピューターのメモリ内の値を表す名前です。値を操作するステートメントを記述する場合、値の代わりに変数名を記述します。コンピューターは、プログラム内に変数名を検出した場合は必ずメモリ内を検索し、そこで見つかった値を使用します。例えば、value1 と value2 の 2 つの変数があり、それぞれに数値が 1 つずつ格納されている場合、2 つの数値を加算するステートメントは次のように記述できます。

```
value1 + value2
```

コンピューターが手順を実際に実行するときには、それぞれの変数に格納された値を検出し、それらを加算します。

ActionScript 3.0 では、実際には変数は 3 つの異なるパートから構成されます。

- 変数の名前
- 変数に格納できるデータ型
- コンピューターのメモリに格納される実際の値

これで、コンピューターがどのようにして名前を値のプレースホルダーとして使用するかがわかりました。データ型も重要です。ActionScript で変数を作成するときには、変数が格納する特定のデータ型を指定します。この時点から、プログラムの命令は、指定されたデータ型のみを変数に格納できるようになります。また、そのデータ型に関連付けられた特定の特性を使用して値を操作できます。ActionScript では、変数を作成する場合（変数の宣言と呼ばれる）、var ステートメントを使用します。

```
var value1:Number;
```

この例は、Number 型のデータのみを格納する value1 という変数の作成をコンピューターに命令しています（「Number」は ActionScript で定義されている固有のデータ型です）。この直後に値を変数に格納することも可能です。

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

Adobe Flash Professional

Flash Professional では、変数を宣言する方法がもう 1 つあります。ステージにムービークリップシンボル、ボタンシンボル、またはテキストフィールドを配置した場合、プロパティインスペクターでインスタンス名を指定できます。内部的には、指定したインスタンス名と同じ名前の変数が、Flash Professional によって作成されます。この変数を ActionScript コードで使用することで、そのステージアイテムを表すことができます。例えば、ステージにムービークリップシンボルを配置し、そのシンボルに rocketShip というインスタンス名を指定するとします。ActionScript コードで変数 rocketShip を使用すると、いつでも必ずそのムービークリップを操作することになります。

定数は、変数とよく似ています。定数とは、指定されたデータ型でコンピューターのメモリ内の値を表す名前ですが、異なる点は、ActionScript アプリケーションで定数には一度に 1 つの値のみを割り当てることができることです。割り当てた定数値は、アプリケーション内で一貫して同じ値になります。定数を宣言するためのシンタックスは変数の宣言と同じですが、var キーワードの代わりに const キーワードを使用する点が異なります。

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

定数は、プロジェクト内の複数の場所で使用され、通常の状況下では変化しない値を定義するのに便利です。リテラル値の代わりに定数を使用することで、コードがさらに読みやすくなります。例えば、同じ処理を行うコードの 2 とおりの記述方法を比較してみましょう。一方のコードでは、定価に SALES_TAX_RATE を掛けます。もう一方のコードでは、定価に 0.07 を掛けます。SALES_TAX_RATE を使った記述の方がコードの目的を理解しやすいことがわかります。また、定数で定義されている値を変更する必要が生じた場合、プロジェクト全体で、定数を使ってその値を表しているのであれば、1 箇所（定数の宣言）で値を変更するだけで済みます。対照的に、ハードコードされたリテラル値を使用している場合は、多くの箇所を変更する必要が生じます。

データ型

ActionScript では、作成する変数のデータ型として使用できる多くのデータ型があります。これらの中には、「単純」な、つまり「基本的」なデータ型と見なすことができるものがあります。

- String：名前または本の章の文字に似たテキスト値
- Numeric：ActionScript 3.0 では数値データに 3 種類のデータ型を使用します。
  - Number：小数点付きまたは小数点なしの値を含むすべての数値
  - int：整数（小数点のない自然数）
  - uint：「符号なし」整数（負でない自然数）
- Boolean：スイッチのオンかオフ、2 つの値が等しいか否かなどの true または false の値

単純なデータ型は、1 つの情報を表します。例えば、単一の数字またはテキストの 1 文などです。ただし、ActionScript で定義されているデータ型の大半は、複雑なデータ型です。それらのデータ型は、1 つのコンテナ内の値セットを表します。例えば、データ型 Date の変数は、1 つの値、すなわち特定の時点を表します。しかし、実際の日付の値は、日、月、年、時

間、分、秒などの複数の値で表され、これらすべては個別の数値です。一般に、日付は 1 つの値と考える方が自然であり、Date 変数を作成することで日付を 1 つの値として扱うことができます。ただし、コンピューターの内部では、日付は、1 つの日付を定義する複数の値から成るグループと見なされます。

プログラマーが定義するデータ型以外にビルトインデータ型の大半は、複雑なデータ型です。複雑なデータ型と認識できるものは次のとおりです。

- MovieClip：ムービークリップシンボル
- TextField：ダイナミックテキストフィールドまたはテキスト入力フィールド
- SimpleButton：ボタンシンボル
- Date：特定の時点に関する情報（日付と時間）

データ型と同じ意味で多く使用される用語に、クラスとオブジェクトがあります。クラスは、データ型の定義です。データ型のオブジェクトすべてのテンプレートと考えることができ、「Example データ型のすべての変数には A、B、C の 3 つの特性があります」と宣言することに似ています。これに対して、オブジェクトは、クラスの実際のインスタンスです。例えば、データ型が MovieClip である変数は、MovieClip オブジェクトとして記述できます。したがって、次のように同じ内容を様々な表現で表すことができます。

- 変数 myVariable のデータ型は Number
- 変数 myVariable は Number インスタンス
- 変数 myVariable は Number オブジェクト
- 変数 myVariable は Number クラスのインスタンス

オブジェクトの操作

ActionScript は、オブジェクト指向のプログラミング言語として知られています。オブジェクト指向のプログラミングは、プログラミングアプローチの 1 つです。このアプローチでは、オブジェクトを使用してプログラムのコードを構成します。

このドキュメントの前半で、「コンピュータープログラム」という用語を、コンピューターが実行する一連の手順（命令）と定義しました。概念上は、コンピュータープログラムは、多数の命令が記述された 1 つの長いリストと想定できます。ただし、オブジェクト指向プログラミングでは、プログラム命令は異なるオブジェクト間で分割されます。コードは機能ごとにまとめられるため、機能に関連するデータ型または関連する情報は、1 つのコンテナにグループ化されます。

Adobe Flash Professional

Flash Professional でシンボルを使用したことのあるデザイナーは、実はオブジェクトの取り扱いを既に知っているといえます。例えば、矩形などのムービークリップシンボルを定義して、そのコピーをステージに配置したとします。そのムービークリップシンボルは、実際には ActionScript のオブジェクトでもあり、MovieClip クラスのインスタンスです。

ムービークリップには、変更可能な様々な特性があります。ムービークリップが選択されている場合、そのムービークリップの x 座標や幅などの値をプロパティインスペクターで変更できます。さらに、アルファ（透明度）を変更したり、ドロップシャドウフィルターをムービークリップに適用することで、様々なカラー調整を行うこともできます。他の Flash Professional ツールを使用すると、自由変形ツールを使って矩形を回転するなど、さらに多くの変更が可能になります。Flash Professional でムービークリップシンボルを変更するこれらの方法はすべて、ActionScript でも使用できます。ActionScript では、MovieClip オブジェクトと呼ばれる 1 つのバンドルにまとめられたデータを変更することで、ムービークリップの変更を行います。

ActionScript のオブジェクト指向プログラミングでは、クラスに 3 種類の特性を指定できます。

- プロパティ

- メソッド
- イベント

これらの要素は、プログラムで使用されるデータの管理と、どのアクションをどの順序で実行するかの決定に使用されます。

プロパティ

プロパティは、オブジェクト内でまとめてバンドルされるデータグループの 1 つを表します。例えば、Song オブジェクトには、artist や title という名前のプロパティを指定できます。MovieClip クラスには、rotation、x、width、alpha などのプロパティを指定できます。プロパティは、個々の変数と同じように扱うことができます。つまり、プロパティは、オブジェクト内に格納された「子」変数と見なすことができます。

次に示すのは、プロパティを使用する ActionScript コードの例です。次のコード行は、square という名前の MovieClip を x 座標で 100 ピクセル移動します。

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

次のコードでは、square MovieClip を triangle MovieClip の回転に合わせて回転するために、rotation プロパティが使用されています。

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

次のコードは、square MovieClip の水平スケールを、従来の 1.5 倍の幅になるように変更します。

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

共通の構造として、オブジェクトの名前として変数（square、triangle）を使用し、ピリオド（.）、プロパティの名前（x、rotation、scaleX）が続きます。ドット演算子と呼ばれるピリオドは、オブジェクトの子要素のいずれかにアクセスしていることを表すために使用します。全体の構造「変数名 - ドット - プロパティ名」は、コンピューターのメモリ内の単一の値の名前として、単一の変数と同様に使用されます。

メソッド

メソッドは、オブジェクトで実行可能なアクションです。例えば、Flash Professional で、タイムラインに複数のキーフレームとアニメーションが配置されたムービークリップシンボルを作成するとします。作成したムービークリップは再生、停止したり、特定のフレームに再生ヘッドを移動するように指示することもできます。

次のコードは、shortFilm という名前の MovieClip に再生を開始するよう指示します。

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

次の行により、shortFilm という名前の MovieClip は再生を停止します（再生ヘッドは、ビデオの一時停止のように、所定の位置で停止します）。

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

次のコードは、shortFilm という MovieClip の再生ヘッドをフレーム 1 に移動し、再生を停止します（ビデオの巻き戻しと同じです）。

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

プロパティと同様、メソッドにアクセスするには、オブジェクト名（変数）、ピリオド、メソッド名、括弧の順で記述します。括弧は、メソッドを呼び出している、つまり、アクションの実行をオブジェクトに指示していることを示します。値（または変数）が括弧内に記述されていることもありますが、このようにすると、アクションの実行に必要な追加情報を渡すことができます。これらの値はメソッドパラメーターと呼ばれます。例えば、gotoAndStop() メソッドには、移動先のフレームの情報が必要なので、括弧内にパラメーターが 1 つ必要です。play() や stop() などのメソッドは自明であり、他の情報は不要です。ただし、この場合も括弧を使って記述されます。

プロパティ（および変数）と異なり、メソッドは値のプレースホルダーとして使用されません。ただし、計算を実行し、変数と同様に使用できる結果を返すメソッドもあります。例えば、Number クラスの toString() メソッドは、数値をそのテキスト表現に変換します。

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

例えば、画面のテキストフィールドに Number 変数の値を表示する場合は、toString() メソッドを使用します。TextField クラスの text プロパティは、String として定義されているので、テキスト値のみ格納できます（property というテキストは、画面に表示される実際のテキストコンテンツを表します）。次のコード行は、変数 numericData の数値をテキストに変換します。次に、画面上の calculatorDisplay という TextField オブジェクトに値を表示します。

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

イベント

コンピュータープログラムとは、コンピューターがステップバイステップ方式で実行する一連の命令です。単純なコンピュータープログラムの場合、コンピューターが実行するいくつかの手順のみで構成されることもあります。また手順を実行した時点でプログラムは終了します。ただし、ActionScript のプログラムはユーザー入力またはその他のイベントの発生を待機しながら、実行を継続するように設計されています。イベントは、コンピューターがどの命令をいつ実行するかを判断するメカニズムです。

基本的には、イベントは ActionScript が認識し応答できる事象です。多くのイベントはボタンのクリックやキーボードからのキー入力など、ユーザー操作に関連しています。また、他の種類のイベントもあります。例えば、ActionScript から外部イメージを読み込む場合、イメージの読み込みの終了時を知らせるイベントが起こります。概念上は、ActionScript プログラムを実行している間、ActionScript プログラムは特定の出来事が発生するのを待機していることになります。特定の出来事が発生すると、それらのイベント用に指定しておいた特定の ActionScript コードが実行されます。

基本的なイベント処理

特定のイベントに応答して実行される特定のアクションを指定する手段を「イベント処理」と呼びます。イベント処理を実行する ActionScript コードを作成する場合、特定する必要がある 3 つの重要な要素があります。

- イベントソース：イベントはどのオブジェクトに対して発生するか。例えば、どのボタンがクリックされたか、またはどの Loader オブジェクトがイメージを読み込んでいるかなどの要素です。イベントソースは、イベントターゲットとも呼ばれます。イベントソースは、コンピューターがイベントのターゲットとする（イベントが実際に発生する）オブジェクトです。
- イベント：起ころうとしていること、すなわち応答が必要な出来事は何か。多くのオブジェクトが複数のイベントをトリガーするため、イベントの特定作業は重要です。
- 応答：イベントが起こったときに、どの手順を実行するか。

イベントを処理する ActionScript コードを作成する場合は、常にコードにこれら 3 つの要素を指定する必要があります。コードは次の基本構造に従います（太字の要素は、固有の事例に応じて入力するプレースホルダーを示します）。

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

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

このコードは 2 つのことを行います。まず、関数を定義します。関数はイベントに対応して実行するアクションをユーザーが指定する方法です。次に、ソースオブジェクトの addEventListener() メソッドを呼び出します。addEventListener() を呼び出すと、指定したイベントに対して関数が「サブスクライブ」されます。そのイベントが発生すると、機能のアクションが実行されます。ここで、各動作を詳細に検討します。

関数は、いくつかのアクションを 1 つの名前でグループ化する手段を提供します。これは、アクションを実行するためのショートカット名のようなものです。関数はメソッドに似ていますが、必ずしも特定のクラスに関連付けられているわけではありません（実際には、「メソッド」を特定のクラスに関連付けられた関数として定義することはできます）。イベント処理を行う関数を作成する場合は、関数の名前を選択する必要があります（この場合は eventResponse）。またパラメーターを 1 つ（この例では eventObject）指定する必要があります。関数パラメーターの指定は、変数の宣言に似ています。このため、パラメーターのデータ型を指定する必要があります。この例ではパラメーターのデータ型は EventType です。

監視するイベントの各タイプには、ActionScript クラスが関連付けられています。関数パラメーターに指定するデータ型は常に、応答の必要な特定のイベントに関連付けられたクラスになります。例えば、click イベント（ユーザーがアイテムをマウスでクリックしたときにトリガーされる）は、MouseEvent クラスに関連付けられています。click イベントに対してリスナー関数を作成するには、MouseEvent データ型のパラメーターを使用してリスナー関数を定義します。最後に、左中括弧と右中括弧の間（{ ... }）に、イベントの発生時にコンピューターに実行を指示する命令を記述します。

以上で、イベント処理関数が作成されました。次に、イベントソースオブジェクト（ボタンなど、イベントの発生原因となるオブジェクト）に、イベントが発生したときにその関数を呼び出すように指示します。関数をイベントソースオブジェクトに登録するには、そのオブジェクトの addEventListener() メソッドを呼び出します（イベントに関連付けられたオブジェクトはすべて、addEventListener() メソッドにも関連付けられます）。addEventListener() メソッドは 2 つのパラメーターを受け取ります。

- 1 つ目は、応答の必要な個々のイベントの名前です。各イベントは特定のクラスに関連付けられています。すべてのクラスは、固有の値を持ちます。この値は、各イベント用に定義されている固有の名前のようなものです。この値は、最初のパラメーターに使用します。
- 2 つ目は、イベント応答関数の名前です。関数名をパラメーターとして渡す場合は、括弧なしで記述します。

イベント処理プロセス

次に、イベントリスナーを作成するときに発生するプロセスの詳しい手順を説明します。この例では、myButton という名前のオブジェクトをクリックしたときに呼び出されるリスナー関数を作成します。

プログラマーが作成する実際のコードは次のとおりです。

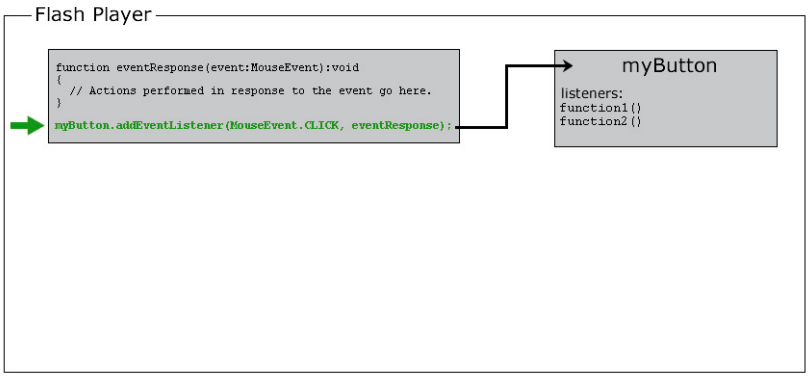
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

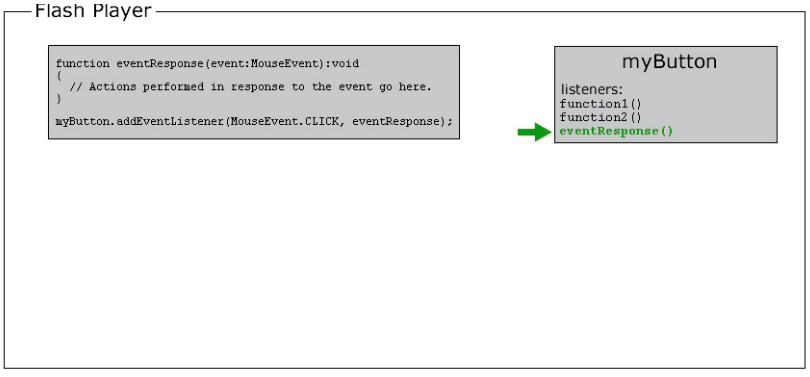
実行時のこのコードの実際の機能を次に示します。

1 SWF ファイルがロードされると、コンピューターは、eventResponse() という名前の関数があることを認識します。

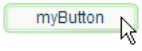
2 次に、コンピューターはコード（具体的には、関数に含まれないコード行）を実行します。この場合は、1 行だけのコードです。イベントソースオブジェクト（myButton）で addEventListener() メソッドを呼び出し、eventResponse 関数をパラメーターとして渡します。



myButton の内部には、その各イベントを監視する関数のリストがあります。その addEventListener() メソッドが呼び出されると、myButton が eventResponse() 関数をそのイベントリスナーのリストに保存します。



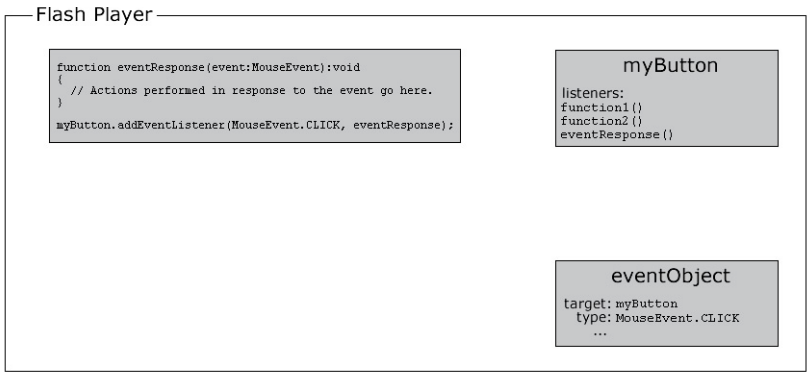
3 ある時点で、ユーザーが myButton オブジェクトをクリックし、その click イベント（コードで MouseEvent.CLICK と識別される）をトリガーします。



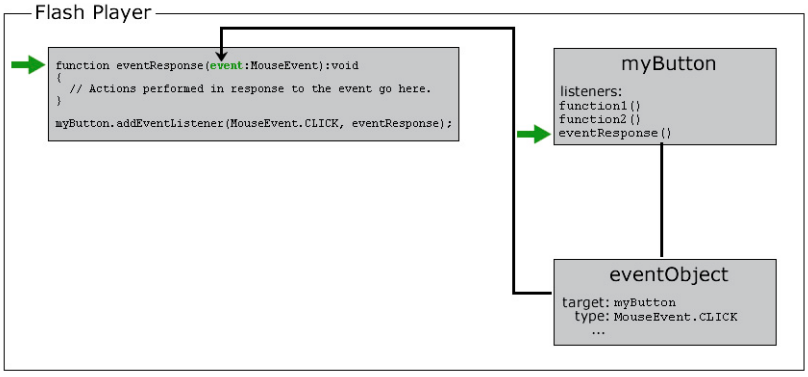
その時点で、以下の処理が行われます。

a 対象のイベントに関連付けられたクラスのインスタンスとなるオブジェクトが作成されます（この例では MouseEvent）。多くのイベントでは、このオブジェクトは Event クラスのインスタンスです。マウスイベントでは、MouseEvent インスタンスです。その他のイベントでは、そのイベントに関連付けられたクラスのインスタンスで

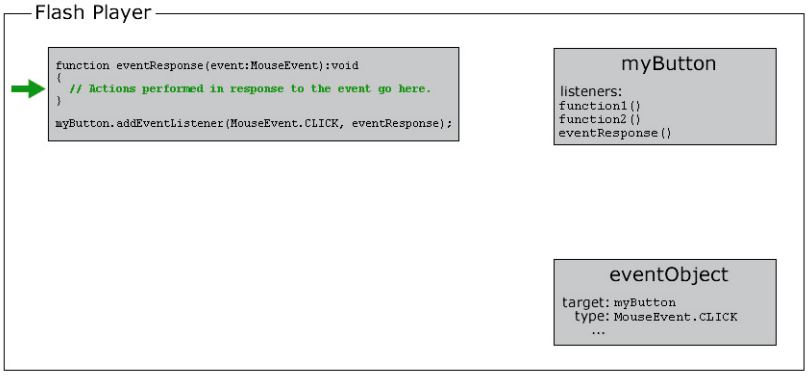
す。作成されるこのオブジェクトはイベントオブジェクトと呼ばれ、ここには、イベントのタイプ、発生場所、その他のイベント固有の情報など、発生したイベントに関する固有の情報が含まれます。



b 次に、コンピューターは myButton に格納されているイベントリスナーのリストを参照します。この関数を 1 つずつ確認し、各関数を呼び出し、イベントオブジェクトをパラメーターとして関数に渡します。eventResponse() 関数は myButton のリスナーの 1 つなので、このプロセスの一環として、コンピューターは eventResponse() 関数を呼び出します。



c eventResponse() 関数が呼び出されると、その関数のコードが実行されるので、指定したアクションが実行されます。



イベント処理の例

ここでは、イベント処理コードの具体例をいくつか紹介します。これらの例は、イベント処理コードの作成に際し、共通のイベント要素とそのバリエーションを考えるときに参考になります。

- ボタンをクリックして、現在のムービークリップの再生を開始。次の例では、playButton はボタンのインスタンス名であり、this は「現在のオブジェクト」を意味する特別な名前です。

```
this.stop();

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

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

- テキストフィールドへの入力の検出。この例では、entryText はテキスト入力フィールド、outputText はダイナミックテキストフィールドです。

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

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

- ボタンをクリックして URL に移動。この場合、linkButton はボタンのインスタンス名です。

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

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

オブジェクトインスタンスの作成

ActionScript でオブジェクトを使用するには、あらかじめそのオブジェクトが存在している必要があります。オブジェクト作成の一環として、変数の宣言がありますが、変数の宣言により作成されるのは、コンピューターメモリ内の空き領域のみです。変数には、使用または操作する前に必ず実際の値を割り当てます。それには、オブジェクトを作成し、これを変数に格納します。オブジェクト作成のプロセスは、オブジェクトのインスタンス化と呼ばれます。要するに、オブジェクトの作成とは、特定のクラスのインスタンスを作成することです。

オブジェクトのインスタンスを作成する場合、ActionScript をまったく使用しない簡単な方法もあります。Flash Professional では、ムービークリップシンボル、ボタンシンボル、またはテキストフィールドをステージに配置し、インスタンス名を割り当てます。Flash Professional は自動的に変数をそのインスタンス名で宣言し、オブジェクトインスタンスを作成し、そのオブジェクトを変数に格納します。同様に、Flex では、MXML タグをコーディングするか、Flash Builder デザインモードのエディターにコンポーネントを配置することで、MXML コンポーネントを作成できます。そのコンポーネントに ID を割り当てた場合、その ID は、コンポーネントのインスタンスを格納する ActionScript の変数名になります。

ただし、オブジェクトを視認しながら作成する必要がない場合もあります。また、不可視のオブジェクトは、視認しながら作成することができません。ActionScript のみを使用して、オブジェクトインスタンスを作成する方法もいくつかあります。

まず、ActionScript のデータ型を何種類か使用する場合、文字式、すなわち ActionScript コードに直接記述される値を使用して、インスタンスを作成できます。次に例を示します。

- Literal 数値（数字を直接入力）：

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

- Literal String 値（テキストを二重引用符で囲む）：

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

- Literal Boolean 値（リテラル値 true または false を使用）：

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

- Literal Array 値（値のコンマ区切りリストを角括弧で囲む）：

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

- Literal XML 値（XML を直接入力）：

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

ActionScript は、Array、RegExp、Object、Function の各データ型にも文字式を定義します。

任意のデータ型のインスタンスを作成する最も一般的な方法では、次のように、new 演算子をクラス名と一緒に使用します。

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

new 演算子を使用したオブジェクトの作成は、多くの場合、「クラスのコンストラクターの呼び出し」と呼ばれます。コンストラクターは、クラスのインスタンスの作成プロセスの一環として呼び出される特別なメソッドです。インスタンスをこの方法で作成する場合、クラス名の後に括弧を配置します。括弧内にパラメーター値を指定することもあります。この 2 つの作業はメソッドを呼び出す場合にも行います。

なお、リテラル値を指定してインスタンスを作成できるデータ型についても、new 演算子を使用してオブジェクトインスタンスを作成することもできます。例えば、次の 2 行のコードは、完全に同じ操作を行います。

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

new **ClassName**() を使用したオブジェクトの作成方法に慣れておく必要があります。多くの ActionScript データ型には視覚表現がありません。したがって、それらのデータ型のインスタンスは、Flash Professional のステージや Flash Builder の MXML エディターのデザインモードにアイテムを配置することによっては作成できません。これらのデータ型のインスタンスを作成するには、ActionScript で new 演算子を使用するしかありません。

Adobe Flash Professional

Flash Professional では、ライブラリで定義されているが、ステージには配置されていないムービークリップシンボルのインスタンスの作成にも、new 演算子を使用できます。

関連項目

配列の操作

正規表現の使用

ActionScript での MovieClip オブジェクトの作成

共通のプログラム要素

ActionScript プログラムの作成に使用する構築ブロックは、他にもいくつかあります。

演算子

演算子は、計算の実行に使用される特殊記号です（words と呼ばれる場合もあります）。演算子はほとんどの場合、算術演算に使用されますが、値同士の比較にも使用されます。一般的に、演算子は 1 つまたは複数の値を使用して、1 つの結果を「算出」します。次に例を示します。

- 加算演算子（+）は、2 つの値を加算し、1 つの数字を返します。

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

- 乗算演算子（*）は 1 つの値を別の値と乗算し、1 つの数字を返します。

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

- 等価演算子（==）は、2 つの値を比較し、等価であるかどうかを確認し、1 つの Boolean 値（true または false）を返します。

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

  ここで示すように、等価演算子とその他の「比較」演算子は、ほとんどの場合、特定の命令を実行すべきかどうかを決定する if ステートメントで使用されます。

コメント

ActionScript を作成するときに、注釈を含めたい場合がよくあります。例えば、コード内の行がどのように動作するかを説明したり、特定の選択を行った理由を説明する場合などです。コードコメントは、コード内のテキストのうち、コンピューターに無視されるテキストを記述する手段です。ActionScript では 2 種類のコメントが使用されます。

- 1 行コメント：1 行コメントは 2 つのスラッシュを行内の任意の場所に配置して指定します。コンピューターは、その行の、スラッシュ以降から行末の改行までのすべてのテキストを無視します。

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

- 複数行コメント：複数行コメントは開始コメントマーカー（/*）、コメント内容、終了コメントマーカー（*/）から構成されます。開始マーカーと終了マーカー間のコメントは、何行に渡っていたとしても、コンピューターから無視されます。

```
/*
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.
*/
```

その他に、コメントのよくある使い方としては、コードの 1 行または複数行を一時的に「無効にする」ことです。例えば、コメントを使用して、ある操作を別の方法でテストすることができます。また、特定の ActionScript コードが予想どおりに動作しない理由を解明する場合にも、コメントを使用することができます。

フロー制御

プログラムでは、特定のアクションを繰り返したり、特定のアクションのみを実行して他のアクションを実行しなかったり、特定の条件下では別のアクションを実行したりする場合が数多くあります。フロー制御は、実行されるアクションに対する制御です。ActionScript では、複数の種類のフロー制御要素を利用できます。

- 関数：関数はショートカットに似ています。関数は、一連のアクションを 1 つの名前でグループ化する手段を提供します。また計算の実行にも使用できます。関数はイベント処理に必ず使用されるほか、一連の命令をグループ化するための汎用的な手段としても使用されます。
- ループ：ループ構造を使用すると、指定した命令セットを、指定した回数、または特定の条件が変化するまでコンピューターに実行させることができます。ループは多くの場合、複数の関連性のあるアイテムの操作に使用され、コンピューターがループを操作するたびに値が変化する変数を使用します。
- 条件ステートメント：条件ステートメントは、特定の条件の下でのみ実行される特定の命令を指定する方法を提供します。さらに、異なる条件に対して別の命令セットを指定するためにも使用できます。条件ステートメントの最も一般的なタイプに、if ステートメントがあります。if ステートメントは、その括弧内の値または式を確認します。値が true であれば、中括弧内のコード行は実行されます。それ以外の場合は無視されます。次に例を示します。

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

  if ステートメントと対になる else ステートメントを使用すると、条件が true 以外の場合に実行される代替命令を指定できます。

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

例：アニメーションポートフォリオピース（Flash Professional）

この例は、ActionScript のピースをつないで、完全なアプリケーションを作成する方法を示す最初の例です。アニメーションポートフォリオピースは、既存のリニアアニメーションに小さなインタラクティブなエレメントを追加します。例えば、クライアント用に作成したアニメーションをオンラインポートフォリオに組み込むことができます。アニメーションに追加するインタラクティブビヘイビアーには、ユーザーがクリックできる 2 つのボタンが含まれます。1 つはアニメーションを開始するボタンで、もう 1 つは別の URL（ポートフォリオメニューや作者のホームページなど）に移動するボタンです。

このピースを作成するプロセスは、以下のメインセクションに分けることができます。

1 ActionScript とインタラクティブ要素を追加するための FLA ファイルを準備します。

2 ボタンを作成して追加します。

3 ActionScript コードを作成します。

4 アプリケーションをテストします。

インタラクティブ機能を追加する準備

アニメーションにインタラクティブなエレメントを追加する前に、FLA ファイルに新しいコンテンツを追加する場所を作成しておくと便利です。このタスクには、ボタンを配置する実際の場所をステージに作成する作業が含まれます。さらに、様々なアイテムを個別に保持できるように、FLA ファイル内に「スペース」を作成する作業も含まれます。

インタラクティブ要素を追加するために FLA をセットアップするには

1. シングルモーショントゥイーンやシェイプトゥイーンなどの単純なアニメーションを含む FLA ファイルを作成します。FLA ファイルが既に存在し、そのファイルにプロジェクトに表示するアニメーションが含まれている場合は、そのファイルを開いて、別の名前を付けて保存します。
2. 2 つのボタンを表示する画面上の位置を決めます。1 つはアニメーションを開始するボタンで、もう 1 つは作者のポートフォリオやホームページにリンクするボタンです。必要な場合は、ステージ上にこの新しいコンテンツのためのスペースを作ります。アニメーションに起動画面がない場合は、最初のフレームに起動画面を作成することもできます。その場合、アニメーションがフレーム 2 以降から始まるように、アニメーションをシフトします。
3. タイムラインの他のレイヤーの上に新しいレイヤーを追加して、**buttons** という名前を付けます。このレイヤーが、ボタンを追加するレイヤーになります。
4. buttons レイヤーの上に新しいレイヤーを追加して、**actions** という名前を付けます。これが、アプリケーションに ActionScript コードを追加する場所になります。

ボタンの作成と追加

次に、インタラクティブアプリケーションの中核となるボタンを実際に作成して、配置する必要があります。

ボタンを作成して、FLA に追加するには

1. 描画ツールを使用して、buttons レイヤーの最初のボタン（「play」ボタン）の外観を作成します。例えば、横長の楕円形を描いて、楕円形の中にテキストを入力します。
2. 選択ツールを使用して、1 つのボタンのすべてのグラフィックパーツを選択します。
3. メインメニューから、変更／シンボルに変換を選択します。
4. ダイアログボックスでシンボルのタイプとして「ボタン」を選択し、シンボルに名前を付けて、「OK」をクリックします。
5. プロパティインスペクターでボタンを選択し、ボタンに **playButton** というインスタンス名を付けます。
6. 手順 1 ～ 5 を繰り返して、作者のホームページに移動するためのボタンを作成します。このボタンに **homeButton** という名前を付けます。

コードの作成

このアプリケーションの ActionScript コードは 3 つの機能セットに分けることができますが、すべて同じ場所に入力されます。このコードでは、次の 3 つのことを行う必要があります。

- SWF ファイルが読み込まれたら（再生ヘッドがフレーム 1 に入ったら）、すぐに再生ヘッドを停止します。
- イベントを監視し、ユーザーが再生ボタンをクリックしたときに SWF ファイルの再生を開始します。
- イベントを監視し、ユーザーが作者ホームページボタンをクリックしたときにブラウザーを適切な URL に移動します。

フレーム 1 に入ったときに再生ヘッドを停止するコードを作成するには

1. actions レイヤーのフレーム 1 のキーフレームを選択します。
2. メインメニューからウィンドウ／アクションを選択して、アクションパネルを開きます。

3 スクリプトペインに次のコードを入力します。

```
stop();
```

再生ボタンがクリックされたときにアニメーションを開始するコードを作成するには

1 前の手順で入力したコードの末尾に、空白行を 2 行追加します。

2 スクリプトの末尾に、次のコードを入力します。

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

このコードでは、startMovie() という名前の関数を定義します。startMovie() が呼び出されると、メインタイムラインが再生を開始します。

3 前の手順で追加したコードの次の行に、以下のコード行を入力します。

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

このコード行は、startMovie() 関数を playButton の click イベントのリスナーとして登録します。言い換えると、playButton という名前のボタンがクリックされるたびに startMovie() 関数が呼び出されるようにします。

ホームページボタンがクリックされたときにブラウザーを URL に移動するコードを作成するには

1 前の手順で入力したコードの末尾に、空白行を 2 行追加します。

2 スクリプトの末尾に、次のコードを入力します。

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

このコードは、gotoAuthorPage() という名前の関数を定義します。この関数は、まず http://example.com/ という URL を表す URLRequest インスタンスを作成します。次に、その URL を navigateToURL() 関数に渡します。これによって、ユーザーのブラウザーでその URL が開きます。

3 前の手順で追加したコードの次の行に、以下のコード行を入力します。

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

このコード行は、gotoAuthorPage() 関数を homeButton の click イベントのリスナーとして登録します。言い換えると、homeButton という名前のボタンがクリックされるたびに gotoAuthorPage() 関数が呼び出されるようにします。

アプリケーションのテスト

この時点で、アプリケーションは完全に機能します。テストして、確認してみましょう。

アプリケーションをテストするには：

1 メインメニューから、制御／ムービープレビュー／を選択します。Flash Professional で SWF ファイルが作成され、Flash Player ウィンドウで開きます。

2 両方のボタンが期待どおりに機能するか確認してみます。

3 ボタンが機能しない場合は、以下の点を確認してください。

- 両方のボタンのインスタンス名が同じになっていませんか。
- addEventListener() メソッド呼び出しで使用される名前がボタンのインスタンス名と同じになっていますか。
- addEventListener() メソッド呼び出しで使用されるイベント名は正しいですか。

- それぞれの関数に対して正しいパラメーターが指定されていますか（どちらのメソッドでも、データ型 MouseEvent のパラメーターが 1 つ指定されている必要があります）。

上記のような問題があると、エラーメッセージが表示されます。エラーメッセージは、「ムービープレビュー」コマンドを選択するか、プロジェクトのテスト中にボタンをクリックしたときに表示されます。コンパイルエラーパネルで、コンパイルエラー（初めて「ムービープレビュー」を選択したときに発生するエラー）がないかどうかを確認してください。また、出力パネルで、ユーザーがボタンをクリックしたときなど、コンテンツの再生中に発生するランタイムエラーがないかどうかを確認してください。

ActionScript でのアプリケーションの構築

アプリケーションを構築するための ActionScript の作成プロセスでは、シンタックスや使用するクラス名を確認する以外の作業が必要です。ほとんどの Flash Platform のマニュアルには、この 2 つのトピック（シンタックスと ActionScript クラスの使用）についての説明があります。しかし、ActionScript アプリケーションを作成するには、次のような情報も把握する必要があります。

- ActionScript の作成に使用できるプログラム
- ActionScript の構成方法
- ActionScript コードをアプリケーションに含める方法
- ActionScript アプリケーションの開発時に従うべき手順

コードを構成するためのオプション

ActionScript 3.0 のコードを使用すると、簡単なグラフィックアニメーションから複雑なクライアント / サーバートランザクション処理システムまであらゆるものを強化できます。ActionScript をプロジェクトに含める場合、構築中のアプリケーションの種類に応じて、様々な方法を選択できます。

Flash Professional タイムラインのフレームへのコードの保存

Flash Professional では、タイムラインのフレームに ActionScript コードを追加できます。追加したコードは、ムービーの再生中に、再生ヘッドがそのフレームに移動すると実行されます。

フレームに ActionScript コードを配置すると、Flash Professional に組み込まれているアプリケーションにビヘイビアーを簡単に追加できます。メインタイムラインのフレームまたは MovieClip シンボルのタイムラインのフレームに、コードを追加できます。しかし、こうした柔軟性を持たせると問題がでてきます。大規模なアプリケーションを構築する場合、どのフレームにどのスクリプトが入っているかを見逃しがちになります。このような複雑な構造にすると、時間の経過に伴いアプリケーションを維持するのがより困難になります。

多くの開発者は、タイムラインの先頭フレーム内のみまたは Flash ドキュメントの特定のレイヤー上のみにコードを配置して、Flash Professional で ActionScript コードの構成を簡略化します。コードを分離することで、Flash FLA ファイルのコードが見つけやすくなり、メンテナンスも簡単になります。ただし、コードをコピーして新しいファイルにペーストしない限り、同じコードを別の Flash Professional プロジェクトで使用することはできません。

作成した ActionScript コードを今後別の Flash Professional プロジェクトで使用できるようにするには、外部 ActionScript ファイル（拡張子 .as のテキストファイル）にコードを保存します。

Flex MXML ファイルへのコードの埋め込み

Flash Builder などの Flex 開発環境では、ActionScript コードを Flex MXML ファイルの <fx:Script> タグの中に含めることができます。ただし、大きなプロジェクトでこのテクニックを使用すると、プロジェクトが複雑になり、同じコードを別の Flex プロジェクトで使用しづらくなります。作成した ActionScript コードを今後別の Flex プロジェクトで使用しやすくするには、外部 ActionScript ファイルにコードを保存します。

注意：<fx:Script> タグには、ソースパラメーターを指定することができます。ソースパラメーターを使用すると、ActionScript コードが <fx:Script> タグに直接入力されたかのように、外部ファイルから ActionScript コードを「読み込む」ことができます。ただし、使用するソースファイルで、その再利用性を制限するカスタムクラスを定義することはできません。

ActionScript ファイルへのコードの保存

プロジェクトに重要な ActionScript コードが含まれている場合、別の ActionScript ソースファイル（.as 拡張子のテキストファイル）でコードを編集するのが最適です。ActionScript ファイルは、アプリケーションで予定される用途に応じて次のいずれかの方法で構築できます。

- 構造化されていない ActionScript コード：タイムラインスクリプトまたは MXML ファイルに直接入力されたかのように記述された、ステートメントまたは関数定義などの ActionScript コードの行。

  この方法で記述された ActionScript は、ActionScript の include ステートメントまたは Flex MXML の <fx:Script> タグを使用してアクセスできます。ActionScript の include ステートメントは、外部 ActionScript ファイルのコンテンツを、スクリプト内の特定の場所の指定されたスコープ内に挿入します。この処理の結果は、コードを直接入力した場合と同じになります。MXML 言語では、ソース属性に <fx:Script> タグを使用することで、アプリケーションのそのポイントでコンパイラーがロードする外部 ActionScript を特定することができます。例えば、次のタグは Box.as という外部 ActionScript ファイルをロードします。

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

- ActionScript クラスの定義：メソッドとプロパティの定義を含む ActionScript クラスの定義。

  クラスを定義する場合、クラスのインスタンスを作成し、そのプロパティ、メソッド、イベントを使用することで、クラスの ActionScript コードにアクセスできます。カスタムクラスを使用することは、組み込まれている ActionScript クラスを使用することとまったく同じです。これには 2 段階の操作が必要です。

  - import ステートメントを使用して、ActionScript コンパイラーが検索箇所を見つけられるように、クラスの完全な名前を指定します。例えば、ActionScript で MovieClip クラスを使用するには、完全な名前を指定し、そのクラスをパッケージとクラスを含めて読み込む必要があります。

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

    または、MovieClip クラスを格納したパッケージを読み込むこともできます。この操作は、パッケージ内の各クラスに import ステートメントを個別に記述することと同じです。

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

    コードでクラスを使用するにはそのクラスを読み込む必要があるというルールには 1 つだけ例外があります。それは、トップレベルのクラスです。これらのクラスは、パッケージで定義されていません。

  - 明示的にクラス名を使用するコードを記述します。例えば、データ型としてそのクラスを使用する変数を宣言するか、その変数に格納するクラスのインスタンスを作成します。ActionScript コード内のクラス名を使用して、コンパイラーにそのクラスの定義をロードするように指示します。例えば、Box という外部クラスがあるとすると、このステートメントによって Box クラスのインスタンスが作成されます。

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

    コンパイラーは、Box クラスへの参照を初めて検出したときに、利用可能なソースコード内でその Box クラスの定義を探します。

適切なツールの選択

ActionScript コードの作成や編集には、多くのツールの中から 1 つのツールを選んで使用することも、いくつかのツールを一緒に使用することもできます。

Flash Builder

Adobe Flash Builder は、Flex フレームワークでのプロジェクトの作成や、主に ActionScript コードから構成されるプロジェクトの作成に最適のツールです。視覚的なレイアウトと MXML 編集機能のほか、Flash Builder には、機能が完備された ActionScript エディターがあります。Flash Builder は、Flex プロジェクトまたは ActionScript 専用プロジェクトの作成にのみ使用できます。Flex が提供する様々な利点の 1 つに、豊富な機能セットがあります。例えば、構築済みのユーザーインターフェイス制御機能や柔軟な動的レイアウト制御機能、リモートデータを操作し、外部データをユーザーインターフェイス要素に結合するためのビルトインメカニズムなどがあります。ただし、このような機能を提供するにはコードの追加が必要になるので、Flex を使用するプロジェクトは、Flex を使用しない同等のプロジェクトに比べて、SWF ファイルのサイズが大きくなることがあります。

Flash Builder は、機能完備のデータ駆動型の高度なインターネットアプリケーションを Flex で作成する場合に使用します。また、ActionScript コードの編集、MXML コードの編集、アプリケーションの視覚的なレイアウトの作業すべてを 1 つのツール内で実行する場合に使用します。

Flash Professional を使用して大量の ActionScript コードを使用するプロジェクトを作成する場合、通常は、ビジュアルアセットの作成に Flash Professional を使用し、ActionScript コードのエディターとして Flash Builder を使用します。

Flash Professional

Flash Professional には、グラフィックとアニメーションを作成する機能に加えて、ActionScript コードを操作するためのツールが含まれています。コードは、FLA ファイル内のエレメントか、外部 ActionScript 専用ファイル内のエレメントに割り当てることができます。Flash Professional は、アニメーションまたはビデオを大量に使用するプロジェクトに最適です。また、グラフィックアセットの大半を自分で作成する場合にも役に立ちます。他にも、同じアプリケーション内にビジュアルアセットとコードの両方を作成する場合には、ActionScript プロジェクトの開発に Flash Professional を使用すると便利です。Flash Professional には、さらに、構築済みのユーザーインターフェイスコンポーネントも用意されています。これらのコンポーネントを使用して、プロジェクトの SWF ファイルサイズを小さく抑え、視覚的なツールを使用してファイルにスキンを適用することができます。

Flash Professional には、ActionScript コードを記述するための 2 つのツールが含まれています。

- アクションパネル：FLA ファイルを操作する場合に選択可能なこのパネルは、タイムラインのフレームにアタッチされた ActionScript コードを作成する場合に使用します。
- スクリプトウィンドウ：スクリプトウィンドウは、ActionScript コードファイル（.as）の操作専用のテキストエディターです。

サードパーティー ActionScript エディター

ActionScript（.as）ファイルは単純なテキストファイルとして保存されているため、プレーンテキストファイルの編集機能を備えたプログラムであればどのプログラムでも、ActionScript ファイルの作成に使用できます。アドビ システムズ社の ActionScript 製品以外に、ActionScript 専用の機能を備えた複数のサードパーティーテキスト編集プログラムが作成されています。MXML ファイルまたは ActionScript クラスは、どのテキストエディタープログラムを使用しても作成できます。その後、Flex SDK を使用して、作成したファイルからアプリケーションを作成できます。このようなプロジェクトでは、Flex または、ActionScript 専用のアプリケーションを使用できます。また、開発者によっては、グラフィカルコンテンツを作成する Flash Professional と、ActionScript クラスを作成する Flash Builder またはサードパーティ製の ActionScript エディターを組み合わせて使用することもあります。

サードパーティ製の ActionScript エディターを使用する理由は、次のとおりです。

- ActionScript は別のプログラムで作成し、ビジュアルエレメントは Flash Professional でデザインする場合。
- ActionScript 以外のプログラミング用のアプリケーションを使用していて、ActionScript のコーディングにもそのアプリケーションを使用する必要がある場合（他のプログラミング言語で HTML ページの作成やアプリケーションの構築を行う場合など）。
- ActionScript 専用プロジェクトまたは Flex プロジェクトを、Flash Professional も Flash Builder も使用せずに、Flex SDK を使用して作成する場合。

ActionScript 専用のサポートを行う主なコードエディターには、次のようなものがあります。

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate（ActionScript と Flex がバンドルされている場合）

ActionScript の開発プロセス

ActionScript プロジェクトの大小に関わらず、アプリケーションの設計および開発のためのプロセスを活用すると、より効率的かつ効果的に作業を進めることができます。次の手順では、ActionScript 3.0 を使用するアプリケーションを構築するための基本的な開発プロセスを説明します。

1 アプリケーションを設計します。

アプリケーションの作成を開始する前に、何らかの方法でアプリケーションを記述する必要があります。

2 ActionScript 3.0 コードを構成します。

ActionScript コードは、Flash Professional、Flash Builder、Dreamweaver、テキストエディターのいずれかを使用して作成できます。

3 Flash プロジェクトまたは Flex プロジェクトを作成して、コードを実行します。

Flash Professional では、FLA ファイルを作成し、パブリッシュ設定を行ってから、ユーザーインターフェイスコンポーネントをアプリケーションに追加して、ActionScript コードを参照します。Flex では、MXML を使用してユーザーインターフェイスコンポーネントを追加し、ActionScript コードを参照します。

4 ActionScript アプリケーションをパブリッシュし、テストします。

アプリケーションのテストでは、開発環境からアプリケーションを実行し、意図したことがすべて実行されていることを確認します。

これらの手順は、順番どおりに実行しなくてもかまいません。また現在の手順を完全に終了せずに次の手順を開始してもかまいません。例えば、まず、アプリケーションの画面の設計を行ってから（手順 1）、グラフィック、ボタンなどを作成（手順 3）します。その後で、ActionScript コードを作成し（手順 2）、テストする（手順 4）ことができます。また、画面の一部を設計し、次に、ボタンまたはインターフェイスの ActionScript を作成し、構築しながらテストすることで、これらのエレメントを一度に 1 つずつ追加することもできます。開発プロセスのこの 4 つの手順は、覚えておくと役に立ちます。実際の開発では、必要に応じて、手順を行う順序を適宜入れ替えると効果的です。

カスタムクラスの作成

プロジェクトで使用するクラスを作成するプロセスは、困難に見えるかもしれません。しかし、クラスの作成においてより困難な段階は、クラスのメソッド、プロパティ、イベントを設計する作業です。

クラスの設計戦略

オブジェクト指向型の設計の原則は複雑なものです。この分野の学術的な研究と専門的な実務に、これまでの知識がすべて注がれてきました。ここではさらに、設計に取り組みやすくするためのアプローチをいくつか提言します。

1 このクラスのインスタンスがアプリケーションで果たす役割を考えてみましょう。一般に、オブジェクトは次の 3 つの役割のいずれかを果たします。

- 値オブジェクト：このオブジェクトは、主にデータのコンテナとして機能します。多数のプロパティと、それよりも数の少ないメソッドを含む可能性があります（あるいはメソッドをまったく含まない場合があります）。このオブジェクトは、通常は、明確に定義されたアイテムをコードで表現したものです。例えば、ミュージックプレーヤーアプリケーションの Song クラス（実世界の 1 つの曲を表す）や、Playlist クラス（曲のグループを表す）などです。
- 表示オブジェクト：実際に画面上に表示されるオブジェクト。例として、ドロップダウンリストやステータス読み出しなどのユーザーインターフェイス要素や、ビデオゲームの登場人物などのグラフィカル要素などがあります。
- アプリケーション構造：このオブジェクトは、ロジック内のサポート、またはアプリケーションで実行される処理といった幅広い役割を担います。例えば、生物学におけるシミュレーションで、特定の計算を実行するオブジェクトを作成できます。ミュージックプレーヤーアプリケーションで、ダイヤルコントロールとボリューム読み出しとの値を同期させる役割を持つオブジェクトを作成できます。また、ビデオゲームで、ルールを管理するオブジェクトを作成することもできます。さらに、保存された写真を描画アプリケーションに読み込むオブジェクトを作成することもできます。

2 クラスで必要とされる個々の機能を決定します。異なる種類の機能がクラスのメソッドになることが頻繁にあります。

3 クラスが値オブジェクトとして機能するように設計されている場合、インスタンスに格納するデータを決定します。このようなアイテムは、適切なプロパティ候補になります。

4 クラスはプロジェクト用に設計するので、アプリケーションに必要な機能を提供することが最も重要な目的になります。次の質問の答えを考えてみましょう。

- どのような情報分野がアプリケーションで保存、追跡、操作されるか。この答えを考えることで、必要な値オブジェクトとプロパティを特定しやすくなります。
- アプリケーションで、どのアクションセットを実行するのか。例えば、アプリケーションが初めてロードされたとき、特定のボタンがクリックされたとき、ムービーの再生が停止したときには、アプリケーションはどのように動作するのでしょうか。このような情報から、どのメソッドを使用するのかを判断します。その「アクション」が個々の値の変更を必要とする場合は、使用するプロパティを判断します。
- 特定のアクションについて、クラスがそのアクションを実行するため、どのような種類の情報を知っておく必要があるか。このような種類の情報は、メソッドのパラメーターになります。
- アプリケーションが作業を進める過程で、アプリケーションの別の部分が知る必要のある、どのようなことがクラスで変化するか。これは適切なイベント候補になります。

5 必要なオブジェクトとよく似た既存のオブジェクトが存在し、必要な機能の一部だけが欠けている場合は、どうすればよいでしょう。そのような場合には、サブクラスの作成を検討してください。（サブクラスとは、カスタムの機能をすべて定義する代わりに、既存のクラスの機能に基づいて構築するクラスです。）例えば、画面上にビジュアルオブジェクトとして表示されるクラスを作成する場合、そのクラスの基本として、既存の表示オブジェクトのビヘイビアーを使用できます。この場合は、その表示オブジェクト（Sprite や MovieClip など）が基本クラスとなり、作成するクラスは「基本クラス」の拡張となります。

クラスのコードの作成

クラスの設計を決定した後、または少なくとも、格納する情報の種類と実行するアクションの種類を決定した後は、実際のクラス作成のシンタックスはまったく簡単なものになります。

次にカスタム ActionScript クラスを作成する最小限の手順を示します。

1 ActionScript テキストエディタープログラムで、新しいテキストドキュメントを開きます。

2 クラスの名前を定義する class ステートメントを入力します。class ステートメントを追加するには、public class、クラス名の順に入力します。次に、クラスのコンテンツ（メソッドとプロパティの定義）を左中括弧と右中括弧の間に入力します。次に例を示します。

```
public class MyClass
{
}
```

単語 public は、そのクラスが他のコードからアクセスできることを示します。他の例については、アクセス制御名前空間の属性を参照してください。

3 クラスが含まれているパッケージの名前を指定する、package ステートメントを入力します。シンタックスは、package、完全なパッケージ名、左中括弧、右中括弧の順です。この中括弧は class ステートメントブロックを囲んでいます。例えば、前の手順のコードは次のように変更されます。

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

4 クラス本文内で var ステートメントを使用して、クラスの各プロパティを定義します。シンタックスは変数の宣言に使用するシンタックスと同じです（public モディファイアが追加されます）。例えば、クラス定義の左中括弧と右中括弧の間に次の行を追加すると、textProperty、numericProperty および dateProperty という名前のプロパティが生成されます。

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

5 関数の定義に使用されるのと同じシンタックスを使用して、クラスに各メソッドを定義します。次に例を示します。

- myMethod() メソッドを作成するには、次のように入力します。

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

- コンストラクター（クラスのインスタンスの作成プロセスの一環として呼び出される特別なメソッド）を作成する場合、そのクラス名と名前が正確に一致するメソッドを作成します。

```
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);
}
```

クラスにコンストラクターメソッドを含めなかった場合、コンパイラーが自動的に空のコンストラクターを作成します（つまり、パラメーターもステートメントも指定されていないコンストラクターが作成されます）。

さらにいくつかのクラスエレメントを定義できます。これらのエレメントは、より複雑な構造を持ちます。

- アクセッサーは、メソッドとプロパティを特別に組み合わせたものです。クラスを定義するコードを作成するときは、メソッドを記述するのと同じようにアクセッサーを記述します。プロパティとして定義すると単なる値の読み取りや代入の操作しかできませんが、アクセッサーでは複数のアクションを実行できます。ところが、クラスのインスタンスを作成するときは、アクセッサーをプロパティと同様に扱い、アクセッサー名を使用して値の読み取りや代入を行うことができます。

- ActionScript のイベントは、特定のシンタックスを使用して定義されません。EventDispatcher クラスの機能を使用して、クラス内に定義します。

関連項目

イベント処理

例：基本的なアプリケーションの作成

ActionScript 3.0 は、Flash Professional や Flash Builder ツール、任意のテキストエディターなどの様々なアプリケーション開発環境で使用できます。

この例では、Flash Professional または Flash Builder を使用した簡単な ActionScript 3.0 アプリケーションの作成と拡張を、手順を追って学習します。作成するアプリケーションは、ActionScript 3.0 の外部クラスファイルを Flash Professional および Flex で使用するための簡単な見本です。

ActionScript アプリケーションの設計

ActionScript アプリケーションのこの例は、標準的な「Hello World」アプリケーションで、その設計は非常に単純です。

- このアプリケーションの名前は HelloWorld です。
- 「Hello, World!」という語句を含むテキストフィールドが 1 つ表示されます。
- このアプリケーションでは、Greeter という名前のオブジェクト指向クラスを 1 つ使用します。この設計では、このクラスは Flash Professional または Flex プロジェクトで使用できます。
- この例では、まず、基本のアプリケーションを作成します。次に、ユーザーにユーザー名を入力させ、アプリケーションにその名前をユーザーリストと照合させる新しい機能を追加します。

この簡潔な定義を使用して、アプリケーションの作成を開始します。

HelloWorld プロジェクトと Greeter クラスの作成

Hello World アプリケーションの設計説明では、再利用しやすいコードを使用するよう指定されています。この目的を達成するために、アプリケーションは Greeter という名前のオブジェクト指向クラスを 1 つ使用します。このクラスは、Flash Builder または Flash Professional で作成したアプリケーション内から使用されます。

HelloWorld プロジェクトと Greeter クラスを Flex で作成するには

1. Flash Builder で、ファイル／新規／ Flex プロジェクトを選択します。
2. 「プロジェクト名」に HelloWorld と入力します。アプリケーションの種類が「Web」に設定されている（Adobe Flash Player で実行される）ことを確認し、「終了」をクリックします。

   Flash Builder によってプロジェクトが作成され、パッケージエクスプローラーに表示されます。デフォルトでは、プロジェクトには HelloWorld.mxml という名前のファイルが既に含まれており、そのファイルがエディターパネルで開きます。
3. 次に、ActionScript のカスタムクラスファイルを Flash Builder で作成するために、ファイル／新規／ ActionScript クラスを選択します。
4. 新規 ActionScript クラスダイアログボックスで、「名前」フィールドにクラス名として「**Greeter**」を入力し、「終了」をクリックします。

   新しい ActionScript 編集ウィンドウが表示されます。

Greeter クラスへのコードの追加に進みます。

Flash Professional で Greeter クラスを作成するには

1 Flash Professional で、ファイル／新規を選択します。

2 新規ドキュメントダイアログボックスで ActionScript ファイルを選択し、「OK」をクリックします。

新しい ActionScript 編集ウィンドウが表示されます。

3 ファイル／保存を選択します。アプリケーションを保存するフォルダーを選択し、ActionScript ファイルに **Greeter.as** という名前を付けて「OK」をクリックします。

Greeter クラスへのコードの追加に進みます。

Greeter クラスへのコードの追加

Greeter クラスはオブジェクト Greeter を定義します。このオブジェクトを HelloWorld アプリケーションで使用します。

Greeter クラスにコードを追加するには

1 新しいファイルに次のコードを入力します（コードの一部があらかじめ自動的に入力されていることもあります）。

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

Greeter クラスには、「Hello World!」というストリングを返す sayHello() メソッドが 1 つ含まれています。

2 ファイル／保存を選択して、この ActionScript ファイルを保存します。

これで、アプリケーションで Greeter クラスを使用できるようになりました。

ActionScript コードを使用するアプリケーションの作成

作成した Greeter クラスは、必要なものをすべて備えたアプリケーション機能を定義しますが、アプリケーション全部を表すわけではありません。Greeter クラスを使用するには、Flash Professional ドキュメントまたは Flex プロジェクトを作成する必要があります。

コードには、Greeter クラスのインスタンスが必要です。次に、アプリケーションで Greeter クラスを使用する方法を示します。

Flash Professional を使用して ActionScript アプリケーションを作成するには

1 ファイル／新規を選択します。

2 新規ドキュメントダイアログボックスで「Flash ファイル（ActionScript 3.0）」を選択し、「OK」をクリックします。

新しいドキュメントウィンドウが表示されます。

3 ファイル／保存を選択します。Greeter.as クラスファイルと同じフォルダーを選択し、Flash ドキュメントに **HelloWorld.fla** という名前を付けて「OK」をクリックします。

4 Flash Professional ツールパレットでテキストツールを選択します。ステージ上をドラッグして、幅 300 ピクセル、高さ 100 ピクセル程度の新しいテキストフィールドを定義します。

5 プロパティパネルのステージでテキストフィールドを選択したまま、テキストの種類を「ダイナミックテキスト」に設定し、テキストフィールドのインスタント名として **mainText** と入力します。

6 メインタイムラインの最初のフレームをクリックします。ウィンドウ／アクションを選択して、アクションパネルを開きます。

7 アクションパネルに次のスクリプトを入力します。

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

8 ファイルを保存します。

ActionScript アプリケーションのパブリッシュとテストに進みます。

Flash Builder を使用して ActionScript アプリケーションを作成するには

1 HelloWorld.mxml ファイルを開き、次のリストと同じになるようにコードを追加します。

```
<?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>
```

この Flex プロジェクトには次の 4 つの MXML タグが含まれています。

- アプリケーションコンテナを定義する <s:Application> タグ
- Application タグのレイアウトスタイル（縦書きレイアウト）を定義する <s:layout> タグ
- ActionScript コードの一部を含む <fx:Script> タグ
- ユーザーにテキストメッセージを表示するためのフィールドを定義する <s:TextArea> タグ

<fx:Script> タグのコードは、アプリケーションがロードされたときに呼び出される initApp() メソッドの定義です。initApp() メソッドは、mainTxt TextArea のテキスト値を、作成したカスタムクラス Greeter の sayHello() メソッドによって返される「Hello World!」ストリングに設定します。

2 ファイル／保存を選択してアプリケーションを保存します。

ActionScript アプリケーションのパブリッシュとテストに進みます。

ActionScript アプリケーションのパブリッシュとテスト

アプリケーション開発は反復プロセスです。コードを記述し、コンパイルして、正しくコンパイルされるまでコードを編集します。コンパイルされたアプリケーションを実行し、テストして、意図した設計が実現されているかどうかを確認します。実現されていない場合は、実現するまでコードを編集します。Flash Professional および Flash Builder の開発環境には、アプリケーションをパブリッシュ、テスト、およびデバッグするための方法が多数用意されています。

次に各環境で HelloWorld アプリケーションをテストする基本的な手順を示します。

Flash Professional を使用して ActionScript アプリケーションのパブリッシュとテストを行うには

1. アプリケーションをパブリッシュして、コンパイルエラーがないか確認します。Flash Professional で、制御／ムービープレビューを選択し、ActionScript コードをコンパイルして HelloWorld アプリケーションを実行します。
2. アプリケーションをテストしたときに、出力ウィンドウにエラーや警告が表示された場合は、HelloWorld.fla または HelloWorld.as ファイルでエラーの原因を修正します。その後、アプリケーションを再びテストします。
3. コンパイルエラーがなければ、Flash Player ウィンドウに Hello World アプリケーションが表示されます。

これで、ActionScript 3.0 を使用する、単純であっても完全なオブジェクト指向アプリケーションが作成されました。HelloWorld アプリケーションの拡張に進んでください。

Flash Builder を使用して ActionScript アプリケーションのパブリッシュとテストを行うには

1. 実行／ HelloWorld を実行を選択します。
2. HelloWorld アプリケーションが起動します。
   - アプリケーションをテストしたときに、出力ウィンドウにエラーや警告が表示された場合は、HelloWorld.mxml または Greeter.as ファイルでエラーの原因を修正します。その後、アプリケーションを再びテストします。
   - コンパイルエラーがなければ、ブラウザーウィンドウに Hello World アプリケーションが表示され、「Hello World!」のテキストが表示されます。

   これで、ActionScript 3.0 を使用する、単純であっても完全なオブジェクト指向アプリケーションが作成されました。HelloWorld アプリケーションの拡張に進んでください。

HelloWorld アプリケーションの拡張

アプリケーションをもう少し興味深いものにするために、アプリケーションがユーザー名を要求し、それを定義済みの名前リストと検証するようにします。

まず、Greeter クラスを更新して新しい機能を追加します。次に、アプリケーションを更新して、新しい機能を使用できるようにします。

Greeter.as ファイルを更新するには

1. "Greeter.as" ファイルを開きます。
2. ファイルの内容を次のように変更します。新しい行および変更された行は太字で表示されています。

```
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;
            }
        }
    }
}
```

Greeter クラスは、いくつかの新しい機能が追加されました。

- validNames 配列は、有効なユーザー名のリストを表示します。この配列は、Greeter クラスのロード時に 3 つの名前のリストに初期化されます。
- sayHello() メソッドは、ユーザー名を受け付け、何らかの条件に基づいて挨拶を変更します。userName が空のストリング（""）である場合、greeting プロパティはユーザーに名前の入力を要求する設定になります。ユーザー名が有効であれば、挨拶は「Hello, < ユーザー名 >」になります。最後に、2 つの条件のいずれかに適合しない場合、greeting 変数は、「Sorry < ユーザー名 >, you are not on the list.」に設定されます。
- validName() メソッドは、inputName が validNames 配列内で見つかった場合は true を、見つからない場合は false を返します。ステートメント validNames.indexOf(inputName) は、validNames 配列内で inputName ストリングに対して各ストリングを確認します。Array.indexOf() メソッドは、配列内のオブジェクトの最初のインスタンスのインデックス位置を返します。配列内にオブジェクト見つからない場合は、値 -1 を返します。

次に、この ActionScript クラスを参照するアプリケーション ファイルを編集します。

Flash Professional を使用してアプリケーションを変更するには

1 HelloWorld.fla ファイルを開きます。

2 フレーム 1 のスクリプトを変更して、空のストリング（""）が Greeter クラスの sayHello() メソッドに渡されるようにします。

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

3 ツールパレットでテキストツールを選択します。ステージ上に 2 つの新しいテキストフィールドを作成します。作成したテキストフィールドを、既存の mainText テキストフィールドのすぐ下に並べて配置します。

4 1 つ目の新しいテキストフィールドに、ラベルのテキストとして **User Name:** と入力します。

5 もう 1 つの新しいテキストフィールドを選択して、プロパティインスペクターでテキストフィールドの種類として InputText を選択します。「行タイプ」に「単一行」を選択します。インスタンス名として **textIn** と入力します。

6 メインタイムラインの最初のフレームをクリックします。

7 アクションパネルで、既存のスクリプトの末尾に以下の行を追加します。

```
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);
    }
}
```

この新しいコードでは、以下の機能が追加されます。

- 最初の 2 行は、単純に 2 つのテキストフィールドの境界線を定義します。
- textIn フィールドなどの入力テキストフィールドには、送出できるイベントのセットがあります。addEventListener() メソッドによって、特定のタイプのイベントが発生したときに実行される関数を定義することができます。この例で該当するイベントは、キーボードのキーが押されることです。
- keyPressed() カスタム関数は、押されたキーが Enter キーであるかどうかを確認します。Enter キーである場合は、myGreeter オブジェクトの sayHello() メソッドを呼び出します。このとき、パラメーターとして textIn テキストフィールドのテキストを渡します。このメソッドは、渡された値に基づいて挨拶のストリングを返します。返されたストリングは、mainText テキストフィールドの text プロパティに割り当てられます。

フレーム 1 の完全なスクリプトは次のようになります。

```
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 ファイルを保存します。

9 制御／ムービープレビューを選択して、アプリケーションを実行します。

アプリケーションを実行すると、ユーザー名の入力を求められます。有効なユーザー名（Sammy、Frank または Dean）を入力すると、「hello」という確認メッセージが表示されます。

Flash Builder を使用してアプリケーションを変更するには

1 HelloWorld.mxml ファイルを開きます。

2 次に、<mx:TextArea> タグを変更し、表示専用であることをユーザーに示します。背景色を明るいグレーに変更し、editable 属性を false に設定します。

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

3 ここで、次の行を <s:TextArea> 終了タグの直後に追加します。これらの行により、ユーザーがユーザー名の値を入力するための TextInput コンポーネントが作成されます。

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

enter 属性は、ユーザーが userNameTxt フィールドで Enter キーを押したときに、行われる動作を定義します。この例では、そのフィールドのテキストが Greeter.sayHello() メソッドに渡されます。mainTxt フィールドに表示される挨拶がそれに応じて変更されます。

HelloWorld.mxml ファイルは次のようになります。

```
<?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 編集した HelloWorld.mxml ファイルを保存します。実行／ HelloWorld を実行を選択してアプリケーションを実行します。

アプリケーションを実行すると、ユーザー名の入力を求められます。有効なユーザー名（Sammy、Frank または Dean）を入力すると、「Hello, **userName**」という確認メッセージが表示されます。

第 3 章：ActionScript 言語とシンタックス

ActionScript 3.0 には、コア ActionScript 言語と Adobe Flash Platform アプリケーションプログラミングインターフェイス（API）の両方が含まれています。コア言語は、言語のシンタックスおよび最上位のデータ型を定義する ActionScript の一部です。ActionScript 3.0 は、Flash Platform ランタイム（Adobe Flash Player および Adobe AIR）へのプログラムによるアクセスを提供します。

言語の概要

オブジェクトは、ActionScript 3.0 言語の中心部に位置する基本的要素です。宣言する変数、記述する関数、作成するクラスインスタンスはすべてオブジェクトです。ActionScript 3.0 プログラムは、タスクを実行し、イベントに応答し、相互に通信するオブジェクトのグループと考えることができます。

Java または C++ のオブジェクト指向プログラミング（OOP）に詳しいプログラマーは、オブジェクトをメンバー変数またはプロパティに格納されたデータ、またはメソッドを介してアクセスできる動作の 2 種類のメンバーを含むモジュールと見なす場合があります。ActionScript 3.0 では、似ているけれども少し異なる方法で、オブジェクトを定義します。ActionScript 3.0 では、オブジェクトは単なるプロパティのコレクションです。プロパティは、データだけではなく関数や他のオブジェクトも保持できるコンテナです。このようにオブジェクトに関連付けられている関数をメソッドと呼びます。

ActionScript 3.0 の定義は Java または C++ のプログラマーには少し奇妙に思えるかもしれませんが、実際には ActionScript 3.0 クラスを使用するオブジェクト型の定義は、Java や C++ でクラスを定義する方法に非常によく似ています。オブジェクトの 2 つの定義の違いは ActionScript オブジェクトモデルおよびその他の高度なテクニックについて説明する場合には重要ですが、ほとんどの場合、プロパティという用語は、メソッドとは対照的に、クラスのメンバー変数を意味します。例えば、Adobe Flash Platform 用 ActionScript 3.0 リファレンスガイドでは、プロパティという用語は、変数または getter/setter プロパティを示すために使用されています。草案では、クラスの一部である関数を意味するメソッドという用語が使用されています。

ActionScript のクラスと Java または C++ のクラスには微妙な違いがあります。それは、ActionScript ではクラスは単なる抽象エンティティではないということです。ActionScript のクラスは、クラスのプロパティおよびメソッドを格納するクラスオブジェクトによって表されます。これにより、クラスまたはパッケージの最上位のインクルードステートメントまたは実行可能なコードなど、Java および C++ プログラマーには異質なものに見える手法が可能になります。

ActionScript のクラスと Java または C++ のクラスのもう 1 つの違いは、ActionScript の各クラスにはプロトタイプオブジェクトと呼ばれるものがあることです。旧バージョンの ActionScript では、プロトタイプオブジェクトはプロトタイプチェーンに関連付けられてクラスの継承階層全体の基盤として機能しましたが、ActionScript 3.0 では、プロトタイプオブジェクトは継承システムにおいて小さな役割を果たすだけです。ただし、クラスのすべてのインスタンスでプロパティとその値を共有する場合、プロトタイプオブジェクトは静的プロパティおよびメソッドの代わりとして引き続き役立ちます。

これまで、熟練した ActionScript プログラマーなら、特別なビルトイン言語エレメントを使用してプロトタイプチェーンを直接操作できました。現在では、ActionScript はクラスベースのプログラミングインターフェイスのさらに成熟した実装を提供しています。__proto__ や __resolve などの特別な言語エレメントの多くは ActionScript に含まれていません。また、パフォーマンスを大幅に向上させる内部継承メカニズムが最適化され、継承メカニズムに直接アクセスできなくなりました。

オブジェクトとクラス

ActionScript 3.0 では、各オブジェクトはクラスにより定義されます。クラスは、オブジェクト型のテンプレートまたは設計図と考えることができます。クラス定義には、データ値を保持する変数と定数、およびクラスにバインドされているビヘイビアーをカプセル化する関数であるメソッドを含めることができます。プロパティに格納される値は、プリミティブ値または他のオブジェクトです。プリミティブ値とは、数値、ストリング、またはブール値です。

ActionScript には、コア言語の一部であるビルトインクラスが多数あります。Number、Boolean、String などのビルトインクラスは、ActionScript で使用可能なプリミティブ値を表します。Array、Math、XML などその他のクラスでは、より複雑なオブジェクトを定義します。

ビルトインおよびユーザー定義のすべてのクラスは、Object クラスから派生します。旧バージョンの ActionScript に慣れているプログラマーは、他のクラスはすべて Object クラスから派生しますが、Object データ型はデフォルトのデータ型ではなくなったことに注意する必要があります。ActionScript 2.0 では、型注釈が存在しないため、変数が Object 型であることを意味しているので、次の 2 行のコードが同等でした。

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

ただし、ActionScript 3.0 は、型指定されていない変数の概念を導入しています。この変数は、次の 2 つの方法で指定することができます。

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

型指定されていない変数は、Object 型の変数と同じではありません。主な違いは、型指定されていない変数は特別な値 undefined を保持できますが、Object 型の変数はその値を保持できないことです。

class キーワードを使用して独自のクラスを定義することができます。クラスプロパティは 3 とおりの方法で宣言できます。定数を定義するには const キーワード、変数を定義するには var キーワード、getter および setter プロパティを定義するにはメソッド宣言で get および set 属性を使用します。メソッドを宣言するには、function キーワードを使用します。

クラスのインスタンスを作成するには、new 演算子を使用します。次の例では、myBirthday と呼ばれる Date クラスのインスタンスを作成します。

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

パッケージと名前空間

パッケージと名前空間は関連する概念です。パッケージを使用すると、コードを共有でき、名前のコンフリクトを最小限に抑えられるようにクラス定義をバンドルできます。名前空間を使用すると、プロパティ名やメソッド名などの識別子の可視性を制御でき、コードがパッケージ内部にあるのか外部にあるのかに関係なく適用できます。パッケージを使用してクラスファイルを構成でき、名前空間を使用して個々のプロパティおよびメソッドの可視性を制御できます。

パッケージ

ActionScript 3.0 では、パッケージは名前空間で実装されますが、名前空間と同義ではありません。パッケージを宣言すると、コンパイル時に必ず既知である名前空間が暗黙的に作成されます。名前空間は、明示的に作成された場合はコンパイル時に既知である必要はありません。

次の例では、package ディレクティブを使用して、クラスを 1 つ含む単純なパッケージを作成します。

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

この例のクラスの名前は SampleCode です。クラスは samples パッケージ内にあるので、コンパイラーはコンパイル時にクラス名を自動的に修飾し、完全修飾名の samples.SampleCode にします。また、コンパイラーはプロパティやメソッドの名前も修飾するので、sampleGreeting および sampleFunction() は、それぞれ samples.SampleCode.sampleGreeting および samples.SampleCode.sampleFunction() になります。

多くの開発者、特に Java 開発者は、パッケージの最上位にクラスだけを配置することを選択します。しかし、ActionScript 3.0 では、パッケージの最上位でクラスだけではなく、変数、関数、ステートメントもサポートされています。この機能の高度な使用方法として、パッケージ内のすべてのクラスで使用できるように、パッケージのトップレベルで名前空間を定義できます。ただし、パッケージの最上位で指定できるアクセス指定子は public と internal の 2 つだけです。ネストされたクラスをプライベート宣言できる Java と異なり、ActionScript 3.0 はネストされたクラスも、プライベートクラスもサポートしていません。

しかし、その他の多くの点で、ActionScript 3.0 のパッケージは Java のパッケージと似ています。前の例を見ればわかるとおり、完全修飾パッケージ参照は、Java と同じようにドット演算子（.）を使用して表します。パッケージを使用すると、他のプログラマーも使用できるように、直感的な階層構造にコードを構成できます。これにより、独自のパッケージを作成して他のプログラマーと共有したり、他のプログラマーが作成したパッケージを利用したりでき、コードの共有が容易になります。

パッケージを使用することで、使用する識別子名が一意になり、他の識別子名と競合しなくなります。実際、これがパッケージの最も重要な利点であるという人もいます。例えば、コードを共有しようとしている 2 人のプログラマーがそれぞれ SampleCode というクラスを作成したとします。パッケージを使用しなければ、名前の競合が発生し、いずれかのクラスの名前を変更するしかなくなります。ただし、パッケージを使用すると、一意の名前を持つパッケージ内のクラスのいずれか、またはできれば両方を配置することによって、名前の競合は簡単に回避されます。

パッケージ名に埋め込みドットを入れて、ネストされたパッケージを作成することもできます。これにより、パッケージの階層構造を作成できます。これを示す適当な例は、ActionScript 3.0 で提供される flash.display パッケージです。flash.display パッケージは flash パッケージ内にネストされています。

ActionScript 3.0 のほとんどは、flash パッケージで編成されています。例えば、flash.display パッケージには表示リスト API が含まれ、flash.events パッケージには新しいイベントモデルが含まれています。

パッケージの作成

ActionScript 3.0 では、パッケージ、クラス、およびソースファイルを整理する方法に大きな柔軟性を持たせています。旧バージョンの ActionScript では、ソースファイルごとにクラスが 1 つだけ許可され、ソースファイルの名前がクラスの名前と一致する必要がありました。ActionScript 3.0 では、1 つのソースファイルに複数のクラスを含めることができますが、ファイルの外部にあるコードで使用できるのは各ファイルのクラス 1 つだけです。つまり、各ファイルのクラスを 1 つだけパッケージ宣言内で宣言できます。その他のクラスはパッケージ定義の外部で定義する必要があります。これにより、追加されるクラスはソースファイル外部にあるコードに対して表示されなくなります。パッケージ定義内で定義されたクラスの名前は、ソースファイルの名前に一致している必要があります。

ActionScript 3.0 では、パッケージをより柔軟に宣言することができます。旧バージョンの ActionScript では、パッケージはソースファイルを配置するディレクトリを表すだけでした。また、package ステートメントでパッケージを宣言するのではなく、クラス宣言内の完全修飾クラス名の一部としてパッケージ名を含めていました。ActionScript 3.0 でもパッケージ

はディレクトリを表しますが、パッケージに含めることができるのはクラスだけではありません。ActionScript 3.0 では、package ステートメントを使用してパッケージを宣言します。つまり、パッケージの最上位で変数、関数、および名前空間を宣言することもできます。パッケージの最上位に実行可能ステートメントを含めることもできます。パッケージの最上位で変数、関数、または名前空間を宣言した場合は、そのレベルで使用できる属性は public および internal だけです。また、この宣言がクラス、変数、関数、または名前空間のいずれであるかに関係なく、public 属性を使用できるのはファイルごとにパッケージレベルの宣言 1 つだけです。

パッケージは、コードを構成したり、名前のコンフリクトを防いだりするのに便利です。パッケージの概念および関連のないクラス継承の概念を混同しないようにしてください。同じパッケージ内にある 2 つのクラスには共通する名前空間がありますが、この 2 つのクラスはその他で関連しているとは限りません。同様に、ネストされたパッケージは、親パッケージと意味的な関係がない場合があります。

パッケージの読み込み

パッケージ内部にあるクラスを使用する場合、パッケージまたは使用するクラスを読み込む必要があります。これは、クラスの読み込みがオプションであった ActionScript 2.0 とは異なります。

例えば、前述の SampleCode クラスの例を考えてみます。クラスが samples という名前のパッケージにある場合、SampleCode クラスを使用する前に次の import ステートメントのいずれかを使用する必要があります。

```
import samples.*;
```

または

```
import samples.SampleCode;
```

一般的に、import ステートメントは可能な限り明確にします。samples パッケージから SampleCode クラスだけを使用する場合、属するパッケージ全体ではなく SampleCode クラスだけを読み込む必要があります。パッケージ全体の読み込みにより、予期しない名前の競合が発生する場合があります。

クラスパス内にパッケージまたはクラスを定義するソースコードを配置する必要もあります。クラスパスは、ローカルディレクトリパスのユーザー定義リストです。このパスは、コンパイラーが読み込むパッケージとクラスを検索する場所を指定します。クラスパスはビルドパスまたはソースパスと呼ばれることもあります。

クラスまたはパッケージを適切に読み込むと、クラスの完全修飾名（samples.SampleCode）またはクラス名のみ（SampleCode）のいずれかを使用できます。

完全修飾名は同じ名前のクラス、メソッド、またはプロパティが存在してコードがあいまいになるときに役立ちますが、すべての識別子に使用すると管理が困難になる可能性があります。例えば、SampleCode クラスのインスタンスをインスタンス化するときに完全修飾名を使用すると、コードが冗長になります。

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

ネストされたパッケージのレベルが高くなるにつれて、コードの可読性が低下します。あいまいな識別子が問題ではないことが確実な場合、単純な識別子を使用してコードを読みやすくすることができます。例えば、クラス識別子のみを使用する場合、SampleCode クラスの新しいインスタンスのインスタンス化はかなり簡単になります。

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

最初に適切なパッケージまたはクラスを読み込まずに識別子の名前を使用しようとしても、コンパイラーはクラス定義を検出できません。また、最初にパッケージまたはクラスを読み込んだ場合でも、読み込んだ名前と競合する名前を定義しようとすると、エラーが生成されます。

パッケージの作成時には、そのパッケージのすべてのメンバーのデフォルトのアクセス制御子は internal です。この場合、デフォルトではパッケージメンバーは、そのパッケージの他のメンバーに対してのみ表示されます。パッケージ外部のコードでクラスを使用できるようにする場合、public でクラスを宣言する必要があります。例えば、次のパッケージには、SampleCode と CodeFormatter の 2 つのクラスが含まれています。

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

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

SampleCode クラスは、public クラスとして宣言されるので、パッケージの外部に表示されます。しかし、CodeFormatter クラスは、samples パッケージ内にのみ表示されます。サンプルパッケージの外部にある CodeFormatter クラスにアクセスしようとすると、次の例に示すように、エラーが発生します。

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

パッケージ外部で両方のクラスを使用する場合、両方のクラスを public として宣言する必要があります。パッケージ宣言に public 属性を適用することはできません。

完全修飾名は、パッケージを使用するときに発生する可能性がある名前のコンフリクトを解決する際に役立ちます。例えば、同じ識別子でクラスを定義する 2 つのパッケージを読み込む場合などです。例えば、SampleCode という名前のクラスも持つ次のパッケージがあるとします。

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

次に示すように両方のクラスを読み込むと、SampleCode クラスの使用時に名前の競合が発生します。

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

コンパイラーには、どちらの SampleCode クラスを使用するのかを知る方法はありません。この競合を解決するには、次のように各クラスの完全修飾名を使用する必要があります。

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

注意：C++ を使用していたプログラマーは、import ステートメントと #include を混同しがちです。C++ コンパイラーは一度に 1 つのファイルを処理するため、C++ では #include ディレクティブが必要です。このディレクティブは、ヘッダーファイルが明示的に含まれていない限り、他のファイル内のクラス定義を探しません。ActionScript 3.0 には include ディレクティブがありますが、クラスやパッケージを読み込むように設計されていません。ActionScript 3.0 でクラスまたはパッケージを読み込むには、import ステートメントを使用して、パッケージを含むソースファイルをクラスパスに配置する必要があります。

名前空間

名前空間では、作成したプロパティおよびメソッドの可視性を制御することができます。public、private、protected、および internal アクセス制御指定子は、ビルトイン名前空間のようなものです。これらのあらかじめ定義されているアクセス制御指定子が要件を満たさない場合は、独自の名前空間を作成することができます。

ActionScript 実装のシンタックスおよび詳細は XML とは少し異なりますが、XML 名前空間を使い慣れていれば、ここでの説明は新しいものではないかもしれません。これまで名前空間を操作したことがない場合は、概念自体は単純ですが、その実装には特定の用語を習得する必要があります。

名前空間の操作を理解するには、プロパティまたはメソッドの名前に識別子と名前空間の 2 つの部分があることを認識することが役立ちます。識別子は名前のようなものと考えることができます。例えば、次のクラス定義の識別子は sampleGreeting と sampleFunction() です。

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

定義の前に名前空間属性がない場合、名前は常にデフォルトの internal 名前空間によって修飾されます。つまり、名前は同じパッケージの呼び出し元だけに表示されます。コンパイラーが strict モードに設定されている場合は、名前空間属性を変更せずに internal 名前空間がすべての識別子に適用されることを示す警告がコンパイラーから出されます。識別子をどこでも使用できるようにするには、識別子名の前に public 属性を付ける必要があります。前のコード例では、sampleGreeting と sampleFunction() の両方とも internal の名前空間値を持ちます。

名前空間を使用するときは、次の 3 つの手順に従います。最初に、namespace キーワードを使用して名前空間を定義する必要があります。例えば、次のコードは、version1 名前空間を定義します。

```
namespace version1;
```

次に、プロパティまたはメソッドの宣言でアクセス制御指定子の代わりに使用して、名前空間を適用します。次の例では、myFunction() という関数を version1 名前空間に配置します。

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

最後に、名前空間を適用すると、use ディレクティブを使用するか、識別子の名前を名前空間で修飾して参照できるようになります。次の例では、use ディレクティブから myFunction() 関数を参照します。

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

次の例に示すように、修飾名を使用して myFunction() 関数を参照することもできます。

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

名前空間の定義

名前空間には、名前空間名とも呼ばれる URI（Uniform Resource Identifier）という値が含まれます。URI によって、名前空間定義を一意にすることができます。

次のいずれかの方法で名前空間定義を宣言して、名前空間を作成します。XML 名前空間を定義するように、明示的な URI で名前空間を定義することも、URI を省略することもできます。次の例では、URI を使用して名前空間を定義する方法を示します。

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

URI は、その名前空間の一意の識別ストリングとして機能します。次の例のように URI を省略した場合、コンパイラーは URI の代わりに一意の内部 ID ストリングを作成します。この内部 ID ストリングにはアクセスできません。

```
namespace flash_proxy;
```

名前空間を定義すると、URI の有無に関わらず、その名前空間は同じスコープ内で再定義することはできません。同じスコープ内で以前に定義された名前空間を定義しようとすると、コンパイルエラーが発生します。

名前空間がパッケージまたはクラス内で定義された場合は、適切なアクセス制御指定子が使用されていなければ、名前空間はそのパッケージまたはクラスの外部にあるコードに対して表示されません。例えば、次のコードは flash.utils パッケージ内で定義された flash_proxy 名前空間を示します。次の例で、アクセス制御指定子が存在しないことは、flash_proxy 名前空間が flash.utils パッケージ内のコードだけに表示され、パッケージ外のコードからは参照できないことを意味します。

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

次のコードは、public 属性を使用して、flash_proxy 名前空間がパッケージ外のコードから参照できるようにします。

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

名前空間の適用

名前空間を適用するとは、名前空間に定義を配置することです。名前空間に配置できる定義には、関数、変数、および定数があります。カスタム名前空間にクラスを配置することはできません。

例えば、public アクセス制御名前空間で宣言された関数があるとします。関数定義に public 属性を使用すると、関数はパブリックの名前空間に配置され、すべてのコードで利用できるようになります。名前空間を定義した後は、public 属性を使用する場合と同じように使用でき、カスタム名前空間を参照可能なコードでその名前空間定義を使用できるようになります。例えば、次の例に示すように、名前空間 example1 を定義すると、example1 を属性として使用して myFunction() というメソッドを追加できます。

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

名前空間 example1 を属性として使用して myFunction() メソッドを宣言することは、メソッドが example1 名前空間に属することを意味します。

名前空間を適用する際には、次の点に注意してください。

- 適用できる名前空間は宣言ごとに 1 つだけです。
- 名前空間属性を一度に複数の定義に適用することはできません。つまり、10 個の異なる関数に名前空間を適用する場合、10 個の異なる関数それぞれに名前空間を属性として追加する必要があります。
- 名前空間とアクセス制御指定子は相互に排他的であるため、名前空間を適用すると、アクセス制御指定子を指定することはできません。つまり、名前空間を適用すると、public、private、protected、または internal として関数またはプロパティを宣言することはできません。

名前空間の参照

public、private、protected、internal などのアクセス制御名前空間で宣言されたメソッドまたはプロパティを使用するとき、名前空間を明示的に参照する必要はありません。このような特別な名前空間へのアクセスはコンテキストによって制御されるからです。例えば、private 名前空間に配置された定義は、自動的に同じクラス内のコードで利用できるようになります。しかし、定義した名前空間では、こうした状況依存性は存在しません。カスタム名前空間に配置したメソッドまたはプロパティを使用するには、その名前空間を参照する必要があります。

use namespace ディレクティブで名前空間を参照できます。また、名前修飾子（::）を使用して名前空間で名前を修飾できます。use namespace ディレクティブで名前空間を参照すると、修飾されていない識別子に名前空間を適用できるように、名前空間が開きます。例えば、example1 名前空間を定義している場合、use namespace example1 を使用して、その名前空間内の名前にアクセスすることができます。

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

一度に複数の名前空間を開くことができます。use namespace で名前空間を開くと、名前空間はそれ自体が開かれたコードブロック全体で開かれたままになります。名前空間を明示的に閉じることはできません。

しかし、複数の名前空間を開くと、名前のコンフリクトが起こりやすくなります。名前空間を開かない場合は、メソッドまたはプロパティの名前を名前空間と名前修飾子で修飾すると、use namespace ディレクティブを使用する必要はありません。例えば、次のコードは、example1 名前空間で名前 myFunction() を修飾する方法を示します。

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

名前空間の使用

ActionScript 3.0 の一部である flash.utils.Proxy クラス内に名前の競合を避けるために使用されている名前空間の実際の例を見つけることができます。Proxy クラスは、ActionScript 2.0 の Object.__resolve プロパティの代替クラスであり、エラーが発生する前に未定義のプロパティまたはメソッドへの参照を取得します。Proxy クラスのすべてのメソッドは、名前の競合を避けるため flash_proxy 名前空間に置かれます。

flash_proxy 名前空間の使用方法についての理解を深めるために、Proxy クラスの使用方法を理解する必要があります。Proxy クラスの機能は、Proxy クラスを継承するクラスでのみ使用できます。つまり、オブジェクト上で Proxy クラスのメソッドを使用する場合、そのオブジェクトのクラス定義は Proxy クラスを拡張する必要があります。例えば、未定義のメソッドへの呼び出しの試みを取得する場合、Proxy クラスを拡張し、Proxy クラスの callProperty() メソッドをオーバーライドします。

名前空間の実装には通常、名前空間の定義、適用、および参照の 3 つの手順を実行します。しかし、Proxy クラスのメソッドを明示的に呼び出していないため、flash_proxy 名前空間は定義および適用されるだけで、参照されません。ActionScript 3.0 では、flash_proxy 名前空間を定義し、これを Proxy クラス内に適用します。コードは、flash_proxy 名前空間を Proxy クラスを拡張するクラスに適用するだけです。

flash_proxy 名前空間は、次のような方法で flash.utils パッケージで定義されます。

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

次の Proxy クラスからの抜粋に示すように、名前空間が Proxy クラスのメソッドに適用されます。

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

次のコードに示すように、まず Proxy クラスと flash_proxy 名前空間の両方を読み込む必要があります。次に、Proxy クラスを拡張するようにクラスを宣言する必要があります。strict モードでコンパイルしている場合は、dynamic 属性を追加する必要もあります。callProperty() メソッドをオーバーライドするときは、flash_proxy 名前空間を使用する必要があります。

```
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);
        }
    }
}
```

MyProxy クラスのインスタンスを作成し、次の例で呼び出される testing() メソッドなどの未定義のメソッドを呼び出すと、Proxy オブジェクトがメソッドの呼び出しを取得し、オーバーライドされた callProperty() メソッド内のステートメントを実行します（この場合は、単純な trace() ステートメント）。

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

flash_proxy 名前空間内に Proxy クラスのメソッドを配置すると、2 つの利点があります。1 つ目は、別の名前空間があると、Proxy クラスを拡張するクラスのパブリックインターフェイスを整理できます。Proxy クラスには、オーバーライドできるメソッドが 10 個以上ありますが、これらのメソッドは直接呼び出せるように設計されていません。これらすべてをパブリック名前空間に配置すると、混乱の元になります。2 つ目は、Proxy サブクラスに Proxy クラスのメソッドと一致する名前のインスタンスメソッドが含まれている場合、flash_proxy 名前空間を使用すると、名前のコンフリクトを回避することができます。例えば、独自のメソッド callProperty() に名前を付けるとします。この callProperty() メソッドは別の名前空間にあるため、次のコードは有効です。

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

名前空間は、4 つのアクセス制御指定子（public、private、internal、および protected）では実行できない方法でメソッドまたはプロパティにアクセスする場合にも便利です。例えば、複数のパッケージに分散するいくつかのユーティリティメソッドがあるとします。これらのメソッドをパブリックにしないで、すべてのパッケージで利用できるようにするには、名前空間を作成して、独自のアクセス制御指定子として使用します。

次の例では、ユーザー定義の名前空間を使用して、異なるパッケージにある 2 つの関数をグループ化します。これらを同じ名前空間にグループ化することにより、両方の関数を 1 つの use namespace ステートメントを介してクラスまたはパッケージに表示することができます。

この例では、4 つのファイルを使用してその手法を示します。ファイルはすべてクラスパス内にある必要があります。1 つ目のファイル myInternal.as を使用して myInternal 名前空間を定義します。このファイルは example パッケージにあるため、example という名前のフォルダーに配置する必要があります。名前空間は public とマークされているので、他のパッケージに読み込むことができます。

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

2 つ目と 3 つ目のファイル Utility.as と Helper.as は、他のパッケージで使用できるメソッドを含むクラスを定義します。Utility クラスは example.alpha パッケージ内にあります。つまり、example フォルダーのサブフォルダーである alpha という名前のフォルダーにファイルを配置する必要があります。Helper クラスは example.beta パッケージ内にあります。つまり、example フォルダーのサブフォルダーでもある beta という名前のフォルダーにファイルを配置する必要があります。これらのパッケージ example.alpha と example.beta の両方とも名前空間を使用する前に読み込む必要があります。

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

4 つ目のファイル "NamespaceUseCase.as" は、メインアプリケーションクラスで、example フォルダーと兄弟関係になければなりません。Flash Professional では、このクラスは FLA の document クラスとして使用されます。NamespaceUseCase クラスも myInternal 名前空間を読み込み、この名前空間を使用して、他のパッケージにある 2 つの静的メソッドを呼び出します。この例では、コードを簡略化するためにのみ静的メソッドを使用します。静的メソッドおよびインスタンスメソッドは両方とも myInternal 名前空間に配置することができます。

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

変数

変数を使用すると、プログラムで使用する値を格納できます。変数を宣言するには、変数名を使用した var ステートメントを使用する必要があります。ActionScript 3.0 では、常に var ステートメントを使用する必要があります。例えば、ActionScript の次の行は、i という名前の変数を宣言します。

```
var i;
```

変数を宣言するときに var ステートメントを省略すると、strict モードではコンパイルエラーが発生し、standard モードではランタイムエラーが発生します。例えば、次のコード行は、変数 i があらかじめ定義されていない場合はエラーになります。

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

変数をデータ型に関連付ける場合は、変数を宣言するときに行います。変数の型を指定しないで変数を宣言することができますが、strict モードではコンパイラー警告が生成されます。変数の型は、変数名にコロン（：）を付け、その後に変数の型を追加することで指定します。例えば、次のコードは int 型の変数 i を宣言します。

```
var i:int;
```

代入演算子（=）を使用して、変数に値を割り当てることができます。例えば、次のコードは、変数 i を宣言し、その変数に値 20 を割り当てます。

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

次の例に示すように、変数を宣言するときに同時に変数を割り当てる方が便利です。

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

変数を宣言するときに変数に値を割り当てる手法は、整数やストリングなどのプリミティブな値を割り当てる場合だけではなく、配列を作成する場合やクラスのインスタンスをインスタンス化する場合にもよく使用されています。次の例は、コードを 1 行使用して宣言され、値が割り当てられる配列を示します。

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

new 演算子を使用して、クラスのインスタンスを作成することができます。次の例では、CustomClass というクラスのインスタンスを作成し、新しく作成されたクラスインスタンスへの参照を customItem という変数に割り当てます。

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

複数の変数を宣言する場合は、カンマ演算子（,）を使用して変数を区切り、1 行のコードですべての変数を宣言することができます。例えば、次のコードは、コード 1 行で 3 つの変数を宣言します。

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

同じコード行で、各変数に値を割り当てることもできます。例えば、次のコードは 3 つの変数（a、b および c）を宣言し、各変数に値を割り当てます。

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

カンマ演算子を使用して変数宣言を 1 つのステートメントにグループ化できますが、コードの可読性が低下する可能性があります。

変数のスコープについて

変数のスコープとは、レキシカル参照によって変数にアクセスできるコードの範囲です。グローバル変数は、コードのすべての範囲で定義される変数で、ローカル変数は、コードの一部分でのみ定義される変数です。ActionScript 3.0 では、変数には、常にその変数が宣言された関数またはクラスのスコープが割り当てられます。グローバル変数は、関数またはクラスの定義の外部で定義する変数です。例えば、次のコードは、関数の外部で宣言して、グローバル変数 strGlobal を作成します。この例では、グローバル変数は関数定義の内部および外部の両方で利用できることを示します。

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

ローカル変数を宣言するには、関数定義内で変数を宣言します。ローカル変数を定義できるコードの最小範囲が関数定義です。関数内で宣言されたローカル変数は、その関数内でのみ存在します。例えば、変数 str2 を関数 localScope() の中で宣言すると、この変数は関数の外部では利用できません。

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

ローカル変数に使用した変数名がグローバル変数として宣言されている場合、ローカル変数がスコープ内にある間は、ローカル定義がグローバル定義を非表示に（シャドウ）します。ただし、グローバル変数は関数の外でも存在しています。例えば、次のコードは、グローバルストリング変数 str1 を作成した後、同じ名前のローカル変数を scopeTest() 関数内に作成します。関数内の trace ステートメントはこの変数のローカル値を出力しますが、関数の外にある trace ステートメントはこの変数のグローバル値を出力します。

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

C++ や Java の変数とは異なり、ActionScript の変数にはブロックレベルのスコープはありません。コードブロックは、左中括弧（{）と右中括弧（}）で囲まれたステートメントのグループです。C++ や Java などのプログラミング言語では、コードブロック内で宣言された変数は、コードブロック外では利用できません。このスコープ制限はブロックレベルのスコープと呼ばれますが、ActionScript にはありません。コードブロック内で変数を宣言すると、変数はそのコードブロック内だけではなく、コードブロックが属する関数の他の部分でも利用できます。例えば、次の関数には様々なブロックスコープで定義された変数が含まれています。すべての変数を関数全体で利用できます。

```
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"]);
```

ブロックレベルのスコープが存在しないので、関数が終了する前に宣言されている限り、変数を宣言する前に読み込んだり、変数に書き込んだりできます。これは、ホイストと呼ばれる手法によるもので、コンパイラーによりすべての変数宣言が関数の最上位に移動されます。例えば、num 変数が宣言される前に num 変数の初期 trace() 関数が実行される場合でも、次のコードはコンパイルされます。

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

しかし、コンパイラーは代入ステートメントをホイストしません。このため、num の初期の trace() は数値データ型の変数のデフォルト値である NaN（非数）になります。つまり、次の例に示すように、変数が宣言される前でも変数に値を割り当てることができます。

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

デフォルト値

デフォルト値とは、値を設定する前に変数に格納されている値です。変数に初めて値を設定する場合は、変数を初期化します。変数を宣言して、値を設定しない場合、その変数は初期化されません。初期化されていない変数の値は、そのデータ型によって異なります。次の表は、変数のデフォルト値をデータ型に整理して示します。

データ型	デフォルト値
Boolean	false
int	0
Number	NaN
Object	null
String	null

データ型	デフォルト値
uint	0
宣言されていない（型注釈 * と同じ）	未定義（undefined）
ユーザー定義クラスを含むその他すべてのクラス	null

Number 型の変数の場合、デフォルト値は NaN（非数）です。これは、IEEE-754 規格で定義されている特別な値で、数値を表さない値です。

変数を宣言して、そのデータ型を宣言しない場合、デフォルトのデータ型 * が適用されます。これは、変数の型指定がないことを示します。型指定されていない変数を値で初期化しない場合も、デフォルト値は undefined です。

Boolean、Number、int、および uint 以外のデータ型の場合、初期化されていない変数のデフォルト値は null です。これは、ActionScript 3.0 で定義されたすべてのクラスおよび作成したカスタムクラスに適用されます。

値 null は、Boolean、Number、int、および uint 型の変数では有効な値ではありません。値 null をこれらの変数に割り当てようとすると、この値はそのデータ型のデフォルト値に変換されます。Object 型の変数の場合、値 null を割り当てることができます。値 undefined を Object 型の変数に割り当てようとすると、この値は null に変換されます。

Number 型の変数では、変数が数値でない場合にブール値 true を返し、そうでない場合に false を返す isNaN() という名前の特殊なトップレベル関数があります。

データ型

データ型は値のセットを定義します。例えば、Boolean データ型は、厳密に 2 つの値 true と false のセットです。ActionScript 3.0 では、Boolean データ型に加えて、String、Number、Array などのよく使用されるデータ型が定義されています。独自のデータ型を定義するには、クラスまたはインターフェイスを使用して値のカスタムセットを定義します。ActionScript 3.0 のすべての値は、プリミティブ値または複合値にかかわらず、オブジェクトです。

プリミティブ値とは、Boolean、int、Number、String、および uint のいずれかのデータ型に属する値を言います。ActionScript では、プリミティブ値はメモリとスピードを最適化できるように特別に格納されるので、通常プリミティブ値は複合値より速く操作できます。

注意：技術的に説明すると、ActionScript ではプリミティブ値はイミュータブルオブジェクトとして内部的に格納されます。イミュータブルオブジェクトとして格納されるということは、参照渡しが値渡しと同じように効果的であることを示しています。参照は通常、値自体よりかなり小さいので、これによりメモリ使用量が削減し、実行速度が向上します。

複合値とはプリミティブ値ではない値です。複合値のセットを定義するデータ型には、Array、Date、Error、Function、RegExp、XML、XMLList などがあります。

多くのプログラム言語では、プリミティブ値とそのラッパーオブジェクトは区別されています。例えば、Java には int プリミティブとそれをラップする java.lang.Integer クラスがあります。Java のプリミティブはオブジェクトではありませんが、そのラッパーはオブジェクトです。このため、処理によってはプリミティブを使用する方が便利な場合とラッパーオブジェクトを使用する方が適切な場合とがあります。ActionScript 3.0 では、プリミティブ値とそのラッパーオブジェクトは実際には区別できません。プリミティブ値を含むすべての値はオブジェクトです。ランタイムでは、これらのプリミティブ型は、オブジェクトのように動作しながら、オブジェクトを作成するための通常のオーバーヘッドが不要な特殊なケースとして扱われます。これは、次の 2 つのコード行が同等であることを意味します。

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

上記に示したすべてのプリミティブデータ型と複合データ型は、ActionScript 3.0 コアクラスで定義されています。コアクラスを使用すると、new 演算子を使用する代わりにリテラル値を使用してオブジェクトを作成できます。例えば、次のようにリテラル値または Array クラスのコンストラクターを使用して、配列を作成することができます。

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

型チェック

型チェックは、コンパイル時または実行時のいずれかの時点で行われます。C++ や Java のような型指定を静的に行う言語では、コンパイル時に型チェックを行います。Smalltalk や Python のような型指定を動的に行う言語では、実行時に型チェックを行います。ActionScript 3.0 は型指定を動的に行う言語なので、実行時に型チェックが行われますが、strict モードと呼ばれる特別なコンパイラーモードでのコンパイル時型チェックもサポートされています。strict モードでは、型チェックはコンパイル時と実行時の両方で行われますが、standard モードでは、型チェックは実行時にのみ行われます。

型指定を動的に行う言語では、コードを非常に柔軟に構成できる反面、型指定エラーが実行時に明らかになるという難点があります。型指定を静的に行う言語では、コンパイル時に型指定エラーが報告されますが、コンパイル時に型情報が分かっている必要があるという難点があります。

コンパイル時の型チェック

コンパイル時の型チェックは、大規模なプロジェクトでよく利用されます。プロジェクトの規模が拡大するにつれて、通常はデータ型の柔軟性よりできるだけ早い時点で型指定エラーを把握する方が重要になるためです。これは、デフォルトで Flash Professional と Flash Builder の ActionScript コンパイラーが strict モードで実行するように設定されているからです。

Adobe Flash Builder

Flash Builder で strict モードを無効にするには、プロジェクトプロパティダイアログボックスの ActionScript コンパイラー設定を使用します。

コンパイル時に型チェックを行うためには、コンパイラーがコード内の変数または式のデータ型情報を分かっている必要があります。変数のデータ型を明示的に宣言するには、変数名の接尾辞としてコロン演算子（:）とその後ろにデータ型を追加します。データ型をパラメーターに関連付けるには、コロン演算子の後ろにデータ型を使用します。例えば、次のコードは xParam パラメーターにデータ型の情報を追加し、変数 myParam を明示的なデータ型で宣言します。

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

strict モードでは、ActionScript コンパイラーは、型の不一致をコンパイルエラーとしてレポートします。例えば、次のコードは、Object 型の関数パラメーター xParam を宣言した後で、このパラメーターに String 型と Number 型の値を割り当てようとします。これは、strict モードでコンパイルエラーを発生します。

```
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);
    }
}
```

ただし、strict モードの場合でも、型指定されていない代入ステートメントの右側をそのままにして、コンパイル時の型チェックを行わないように選択できます。変数または式に型指定なしとマークするには、型注釈を省略するか、特別なアスタリスク（*）型注釈を使用します。例えば、上記の例で、型注釈を使用しないように xParam パラメーターを変更して、strict モードでコードをコンパイルします。

```
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");
```

実行時の型チェック

ActionScript 3.0 では、実行時の型チェックは、strict モードと standard モードのいずれでコンパイルされるかに関係なく行われます。配列を必要とする関数に値 3 がパラメーターとして渡されるとします。strict モードでは、値 3 は Array データ型と互換性がないため、コンパイラーはエラーを生成します。strict モードを無効にし、standard モードで実行すると、コンパイラーは型の不一致について報告しませんが、実行時の型チェックではライタイムエラーになります。

次の例では、Array 引数を必要とするにもかかわらず、値 3 が渡される typeTest() という名前の関数を示します。この場合、値 3 はパラメーターで宣言されたデータ型（Array）のメンバーではないので、standard モードでランタイムエラーが発生します。

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

strict モードで操作しているときでも、実行時の型指定エラーが発生する場合もあります。これは strict モードを使用しているときに、型指定されていない変数を使用してコンパイル時の型チェックをしないようにしている場合に発生します。型指定されていない変数を使用すると、型チェックは省略されるのではなく、実行時まで保留されます。例えば、上記の例の myNum 変数に宣言したデータ型が存在しない場合、コンパイラーは型の不一致を検出できません。ただし、このコードは、代入ステートメントの結果として 3 に設定された実行時の値 myNum を Array データ型に設定された xParam の型と比較するので、ランタイムエラーを生成します。

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

実行時に型チェックを行うと、コンパイル時に型チェックを行うより継承を柔軟に使用することができます。standard モードでは、型チェックを実行時まで保留することで、アップキャストする場合でもサブクラスのプロパティを参照できます。アップキャストは、基本クラスを使用してクラスインスタンスの型を宣言し、クラスのインスタンス化にサブクラスを使用するときに行われます。例えば、拡張可能な ClassBase という名前のクラスを作成することができます（final 属性を持つクラスは拡張できません）。

```
class ClassBase
{
}
```

続いて、次のように someString という名前の 1 つのプロパティを持つ ClassExtender という名前の ClassBase のサブクラスを作成することができます。

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

両方のクラスを使用して、ClassBase データ型を使用して宣言されているが、ClassExtender コンストラクターを使用してインスタンス化されているクラスインスタンスを作成することができます。基本クラスにはサブクラスにないプロパティやメソッドは含まれないので、アップキャストは安全な操作と考えられています。

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

ただし、サブクラスには、その基本クラスに含まれないプロパティまたはメソッドが含まれます。例えば、ClassExtender クラスには someString プロパティが含まれますが、このプロパティは ClassBase クラスには含まれません。ActionScript 3.0 の standard モードでは、次の例に示すように、myClass インスタンスを使用してコンパイル時エラーを生成せずにこのプロパティを参照することができます。

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

is 演算子

is 演算子を使用すると、変数または式が特定のデータ型のメンバーであるかどうかをテストすることができます。旧バージョンの ActionScript では instanceof 演算子に同じ機能がありましたが、ActionScript 3.0 では instanceof 演算子はデータ型のメンバーシップのテストに使用しません。手動による型チェックでは、instanceof 演算子の代わりに is 演算子を使用する必要があります。これは、式 x instanceof y は、x のプロトタイプチェーンに y があるかどうかをチェックするだけであるため、また ActionScript 3.0 では、プロトタイプチェーンで継承階層全体を確認できないためです。

is 演算子は、適切な継承階層を調べます。また、オブジェクトが特定のクラスのインスタンスであるかどうかだけではなく、特定のインターフェイスを実装するクラスのインスタンスであるかどうかも調べるために使用することができます。次の例では、Sprite クラスのインスタンス mySprite を作成し、is 演算子を使用して mySprite が Sprite クラスおよび DisplayObject クラスのインスタンスであるかどうか、および IEventDispatcher インターフェイスを実装しているかどうかをテストします。

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

is 演算子は、継承階層を調べて、mySprite が Sprite クラスおよび DisplayObject クラスと互換性があることを適切に報告します。ただし、Sprite クラスは DisplayObject クラスのサブクラスです。is 演算子は、mySprite が IEventDispatcher インターフェイスを実装するクラスを継承するかどうかも調べます。Sprite クラスは、IEventDispatcher インターフェイスを実装する EventDispatcher クラスを継承するため、is 演算子は、mySprite が同じインターフェイスを実装することを正しく報告します。

次の例は、前の例と同じテストを示していますが、is 演算子の代わりに instanceof 演算子を使用しています。instanceof 演算子は mySprite が Sprite または DisplayObject のインスタンスであることを正しく識別しますが、mySprite が IEventDispatcher インターフェイスを実装しているかどうかをテストするのに使用すると false を返します。

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

as 演算子

as 演算子を使用しても、式が指定されたデータ型のメンバーであるかどうかをチェックすることができます。しかし、is 演算子とは異なり、as 演算子はブール値を返しません。as 演算子は、true の代わりに式の値、false の代わりに null を返します。次の例は、Sprite インスタンスが DisplayObject、IEventDispatcher、および Number データ型のメンバーであるかどうかをチェックする場合に、is 演算子の代わりに as 演算子を使用した結果を示します。

```
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
```

as 演算子を使用する場合、右側のオペランドはデータ型である必要があります。右側のオペランドにデータ型ではなく式を使用しようとすると、エラーになります。

ダイナミッククラス

ダイナミッククラスは、プロパティおよびメソッドを追加したり変更したりすることで実行時に変更可能なオブジェクトを定義します。String クラスなどの動的ではないクラスは、sealed クラスです。実行時に、sealed クラスにプロパティまたはメソッドを追加することはできません。

ダイナミッククラスを作成するには、クラスを宣言するときに dynamic 属性を使用します。例えば、次のコードは Protean という名前のダイナミッククラスを作成します。

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

後から Protean クラスのインスタンスをインスタンス化する場合、クラス定義外のインスタンスにプロパティまたはメソッドを追加することができます。例えば、次のコードは、Protean クラスのインスタンスを作成し、プロパティ aString とプロパティ aNumber をそのインスタンスに追加します。

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

ダイナミッククラスのインスタンスに追加したプロパティは、実行時エンティティです。したがって、型チェックは実行時に行われます。この方法で追加されたプロパティに型注釈を追加することはできません。

また、myProtean インスタンスにメソッドを追加するには、関数を定義し、その関数を myProtean インスタンスのプロパティに関連付けます。次のコードは、trace ステートメントを traceProtean() というメソッドに移動します。

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

しかし、この方法で作成されたメソッドは、Protean クラスのプライベートプロパティまたはメソッドにアクセスできません。また、Protean クラスのパブリックプロパティまたはメソッドへの参照も、this キーワードまたはクラス名のいずれかで修飾する必要があります。次の例は、Protean クラスのプライベート変数およびパブリック変数にアクセスしようとする traceProtean() メソッドを示します。

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

データ型の詳細

プリミティブデータ型には、Boolean、int、Null、Number、String、uint、および void があります。ActionScript コアクラスは、複合データ型 Object、Array、Date、Error、Function、RegExp、XML、および XMLList も定義します。

Boolean データ型

Boolean データ型は、true と false の 2 つの値で構成されます。Boolean 型の変数に有効な値は、この 2 つだけです。宣言されているが、初期化されていない Boolean 変数のデフォルト値は false です。

int データ型

int データ型は、32 ビット整数として内部に保存され、

-2,147,483,648 (-2^{31}）～ 2,147,483,647（$2^{31} - 1$）（両端を含む）の整数のセットで構成されます。旧バージョンの ActionScript には、整数と浮動小数点数の両方に使用する Number データ型しかありませんでした。ActionScript 3.0 では、32 ビット符号付および符号なし整数の低レベルマシン型にアクセスすることができるようになりました。変数が浮動小数点数を使用しない場合、Number データ型ではなく int データ型を使用する方が速く、効率的です。

最小 int 値と最大 int 値の範囲外の整数値の場合は、Number データ型を使用します。Number データ型では、-9,007,199,254,740,992 ～ +9,007,199,254,740,992 の値（53 ビット整数値）を処理できます。int データ型である変数のデフォルト値は 0 です。

Null データ型

Null データ型は、1 つの値 null だけで構成されます。これは、String データ型および Object クラスを含む複合データ型を定義するすべてのクラスのデフォルト値です。Boolean、Number、int、uint などの他のプリミティブデータ型には、値 null は含まれません。Boolean 型、Number 型、int 型、uint 型 のいずれかの変数に値 null を割り当てようとすると、実行時に null が適切なデフォルト値に変換されます。このデータ型を型注釈として使用することはできません。

Number データ型

ActionScript 3.0 では、Number データ型は、整数、符号なし整数、および浮動小数点数を表すことができます。ただし、パフォーマンスを最大化するために、32 ビット int および uint 型より大きい整数値に対してのみ、Number データ型を使用する必要があります。また、Number データ型には、浮動小数点数を格納することができます。浮動小数点数を格納するには、数値に小数点を含めます。小数点を省略すると、数値は整数として格納されます。

Number データ型では、2 進数浮動小数点計算のための IEEE 規格（IEEE-754）で指定されている 64 ビット倍精度フォーマットを使用します。この規格では、64 ビットを使用して浮動小数点数を格納する方法が指定されています。1 ビットを使用して数値が正か負かを指定します。指数には 11 ビットが使用され、2 を底として格納されます。残りの 52 ビットは、指数で指定される累乗した数値である仮数部（仮数とも呼ばれます）を格納するために使用されます。

Number データ型では、ビットの一部を使用して指数を格納することで、仮数のビットすべてを使用する場合より大きな浮動小数点数を格納できます。例えば、Number データ型で 64 ビットすべてを使用して仮数を格納した場合は、$2^{65} - 1$ の数値を格納できます。Number データ型では、11 ビットを使用して指数を格納することで、仮数を 2^{1023} 乗できます。

Number 型が表すことができる最大値と最小値は、Number.MAX_VALUE および Number.MIN_VALUE と呼ばれる Number クラスの静的プロパティに格納されます。

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

この数値の範囲が膨大である一方で、この範囲の精度が犠牲になっています。Number データ型では、仮数の格納に 52 ビットを使用します。その結果、分数 1/3 など、正確に表すためには 53 ビット以上必要な数値は近似値にしかなりません。アプリケーションで小数値を持つ絶対精度が必要な場合、2 進浮動小数点数計算ではなく、10 進浮動小数点計算を実装するソフトウェアを使用する必要があります。

Number データ型で整数値を格納する場合、仮数の 52 ビットだけが使用されます。Number データ型は、これらの 52 ビットと特殊な非表示のビットを使用して、-9,007,199,254,740,992（-2^{53}）～ 9,007,199,254,740,992（2^{53}）の整数を表します。

NaN 値は、Number 型の変数のデフォルト値としてだけではなく、数値を返す必要があるが返さない演算の結果としても使用します。例えば、負の数値の平方根を計算しようとすると、その結果は NaN になります。その他の特殊な Number の値には正の無限大と負の無限大があります。

注意：0 による除算の結果は、除数も 0 の場合、NaN だけです。0 による除算の結果は、被除数が正の場合に正の無限大、被除数が負の場合に負の無限大となります。

String データ型

String データ型は、16 ビット文字のシーケンスを表します。ストリングは、UTF-16 形式を使用して、Unicode 文字として内部的に格納されます。ストリングは、Java プログラミング言語の場合と同様に不変値です。String 値に対する操作は、そのストリングの新しいインスタンスを返します。String データ型で宣言される変数のデフォルト値は null です。値 null は、空のストリング（""）とは異なります。値 null は、変数に格納されている値がないということを意味するのに対して、空のストリングは、変数に格納されている値が文字を含まないストリングであることを意味します。

uint データ型

uint データ型は、32 ビット符号なし整数として内部に保存され、0 ～ 4,294,967,295（$2^{32} - 1$）（両端を含む）の整数のセットで構成されます。負以外の整数が必要な特別な場合に、uint データ型を使用します。例えば、ピクセルカラー値を表すためには uint データ型を使用する必要があります。これは、int データ型にはカラー値の処理には適していない内部符号ビットがあるためです。最大 uint 値より大きな整数値には、53 ビット整数値を処理できる Number データ型を使用します。uint データ型の変数のデフォルト値は 0 です。

void データ型

void データ型は、1 つの値 undefined だけで構成されます。旧バージョンの ActionScript では、undefined は Object クラスのインスタンスのデフォルト値でした。ActionScript 3.0 では、Object インスタンスのデフォルト値は null です。値 undefined を Object クラスのインスタンスに割り当てようとすると、この値は null に変換されます。型指定されていない変数には、undefined という値のみを割り当てることができます。型指定されていない変数は、型注釈がないか、型注釈にアスタリスク記号（*）が使用されている変数です。void は、戻り値の型注釈としてのみ使用することができます。

Object データ型

Object データ型は Object クラスによって定義されます。Object クラスは、ActionScript のすべてのクラス定義の基本クラスです。ActionScript 3.0 の Object データ型は、次の 3 つの点で旧バージョンとは異なります。1 つ目は、Object データ型は、型注釈のない変数に割り当てられるデフォルトのデータ型ではなくなりました。2 つ目は、Object データ型には、Object インスタンスのデフォルト値であった値 undefined が含まれなくなりました。3 つ目は、ActionScript 3.0 では、Object クラスのインスタンスのデフォルト値が null になった点です。

旧バージョンの ActionScript では、型注釈のない変数には自動的に Object データ型が割り当てられていました。ActionScript 3.0 ではこれは行われず、真に型指定されていない変数の概念が導入されました。型注釈のない変数は、型が指定されていないと見なされます。コード内で変数の型を指定しないままにしておくことを明確に示す場合、型注釈にアスタリスク記号（*）を使用することができます。これは、型注釈を省略するのと同じことを示します。次の例は、いずれも型指定されていない変数 x を宣言する 2 つの同等のステートメントを示します。

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

型指定されていない変数だけが値 undefined を保持することができます。値 undefined をデータ型が指定されている変数に割り当てようとすると、ランタイムは値 undefined をそのデータ型のデフォルト値に変換します。データ型のインスタンスでは、デフォルト値は null です。これは、Object インスタンスに undefined を割り当てようとすると、値が null に変換されることを意味します。

型変換

型変換は、値が別のデータ型の値に変換されることです。型変換は、暗黙的または明示的に行うことができます。暗黙的な変換は、強制型変換とも呼ばれ、実行時に行われることがあります。例えば、値 2 を Boolean データ型の変数に割り当てると、値 2 はブール値 true に変換されかてら、変数に割り当てらます。明示的な変換は、キャストとも呼ばれ、コードであるデータ型の変数を別のデータ型に属している場合と同様に処理するようにコンパイラーに指示されると実行されます。プリミティブ値が使用されている場合、キャストによって実際にあるデータ型の値が別のデータ型に変換されます。オブジェクトを異なる型にキャストするには、オブジェクト名を括弧で囲み、その前に新しい型の名前を置きます。例えば、次のコードはブール値を取り、整数にキャストします。

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

暗黙的な変換

暗黙的な変換は、次のような状況で実行時に行われます。

- 代入ステートメント内
- 値が関数パラメーターとして渡されるとき
- 関数から値が返されるとき
- 加算（+）演算子などの特定の演算子を使用した式内

  ユーザー定義型の場合、変換される値が変換先のクラスまたは変換先のクラスから派生するクラスのインスタンスである場合に、暗黙的な変換は成功します。暗黙的な変換が成功しなかった場合は、エラーが発生します。例えば、次のコードには、成功する暗黙的な変換と失敗する暗黙的な変換が含まれています。

```
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.
```

プリミティブ型の場合、暗黙的な変換は、明示的な変換関数によって呼び出される同じ内部変換アルゴリズムを呼び出して処理されます。

明示的な変換

strict モードでコンパイルする場合、明示的な変換、つまりキャストを使用すると便利です。型の不一致によるコンパイル時エラーを生成したくない場合があるためです。これは、実行時に強制的に値が適切に変換されることがわかっている場合などです。例えば、フォームから受信したデータを操作するとき、強制的にストリング値を数値に変換することができます。次のコードは、コードが standard モードで正しく実行されている場合であっても、コンパイル時エラーを発生します。

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

strict モードを引き続き使用して、ストリングを整数に変換する場合、次のように明示的な変換を使用することができます。

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

int、uint、および Number へのキャスト

任意のデータ型を 3 つの数値型 int、uint、および Number 型のいずれかにキャストできます。何らかの理由で数値を変換できない場合、int および uint データ型ではデフォルト値 0 が割り当てられ、Number データ型ではデフォルト値 NaN が割り当てられます。ブール値を数値に変換すると、true は値 1、false は値 0 になります。

```
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
```

数字のみを含むストリング値は、数値型のいずれかに変換することができます。数値型では、負の数のように見えるストリングまたは 16 進数値（例えば、0x1A）を表すストリングも変換することができます。変換プロセスでは、ストリング値の先頭および末尾の空白は無視されます。Number() を使用して浮動小数点数のように見えるストリングをキャストすることもできます。小数点が含まれていると、uint() および int() は、小数点とその後の数字が切り捨てられた整数を返します。例えば、次のストリング値は数値にキャストすることができます。

```
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
```

数値以外の文字を含むストリング値は、int() または uint() でキャストすると 0 を返し、Number() でキャストすると NaN を返します。変換プロセスでは、先頭および末尾の空白は無視されますが、ストリング値に 2 つの数値を区切る空白がある場合は、0 または NaN が返されます。

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

ActionScript 3.0 では、Number() 関数は 8 進数をサポートしていません。ActionScript 2.0 の Number() 関数に先頭が 0 のストリングを指定した場合、数値は 8 進数として解釈され、10 進数に変換されます。これは、ActionScript 3.0 の Number() 関数には当てはまりません。この関数では、先頭の 0 は無視されます。例えば、次のコードは、異なるバージョンの ActionScript を使用してコンパイルすると異なる出力を生成します。

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

数値型の値が別の数値型の変数に割り当てられる場合、キャストは必要ありません。strict モードでも、数値型は別の数値型に暗黙的に変換されます。これは、場合によっては、型の範囲を超えると予期しない値になることを示します。次の例では、予期しない値が生成される場合もありますが、すべて strict モードでコンパイルされます。

```
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
```

次の表に、別のデータ型から Number、int、uint データ型にキャストした結果の概要を示します。

データ型または値	**Number、int、uint 型への変換結果**
Boolean	値が true の場合は 1、それ以外の場合は 0
Date	Date オブジェクトの内部表現で、1970 年 1 月 1 日午前 0 時（世界時）からのミリ秒
null	0
Object	インスタンスが null で Number 型に変換される場合は NaN、それ以外の場合は 0
String	ストリングを数値に変換できる場合は数値、それ以外の場合は Number 型に変換されると NaN、int 型または uint 型に変換されると 0
未定義（undefined）	Number 型に変換される場合は NaN、int 型または uint 型に変換される場合は 0

Boolean へのキャスト

任意の数値データ型（uint、int、および Number）から Boolean へキャストすると、数値が 0 の場合は false が、それ以外の場合は true が返されます。Number データ型では、値 NaN のときも false になります。次の例では、数値 -1、0、および 1 をキャストした場合の結果を示します。

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

この例による出力では、3 つの数値の内 0 だけが値 false を返すことを示します。

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

ストリングが null または空のストリング（""）の場合、String 値から Boolean 値へキャストすると、false が返されます。それ以外の場合は、true を返します。

```
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
```

Object クラスのインスタンスから Boolean にキャストすると、インスタンスが null の場合は false が返され、それ以外の場合は true が返されます。

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

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

Boolean 型の変数は、strict モードで任意のデータ型の値をキャストしないで Boolean 型変数に割り当てることができます。すべてのデータ型から Boolean データ型への暗黙的な強制型変換は、strict モードでも行われます。つまり、ほとんどのデータ型とは異なり、strict モードのエラーを防ぐために Boolean 型へのキャストは必要ありません。次の例では、すべて strict モードでコンパイルされ、実行時に意図したとおりに動作します。

```
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
```

次の表に、他のデータ型から Boolean データ型へのキャストの結果を要約して示します。

データ型または値	**Boolean 型への変換結果**
String	値が null または空のストリング（""）の場合は false、それ以外の場合は true
null	false
Number、int、uint	値が NaN または 0 の場合は false、それ以外の場合は true
Object	インスタンスが null の場合は false、それ以外の場合は true

String へのキャスト

任意の数値データ型から String データ型へキャストすると、数値のストリング表現が返されます。Boolean 型から String 型にキャストすると、値が true の場合はストリング「true」、値が false の場合はストリング「false」が返されます。

Object クラスのインスタンスから String データ型にキャストすると、インスタンスが null の場合はストリング「null」が返されます。それ以外の場合は、Object クラスから String 型にキャストすると、ストリング「[object Object]」が返されます。

Array クラスのインスタンスから String 型にキャストすると、すべての配列エレメントのカンマ区切りリストから構成されるストリングが返されます。例えば、次の String データ型へのキャストは、配列の 3 つのエレメントすべてから成るストリングを 1 つ返します。

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

Date クラスのインスタンスから String 型にキャストすると、インスタンスに含まれる日付のストリング表現が返されます。例えば、次の例は Date クラスインスタンスのストリング表現を返します。この出力は、太平洋標準時の場合の結果を示します。

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

次の表に、別のデータ型から String データ型にキャストした結果の概要を示します。

データ型または値	String 型への変換結果
Array	すべての配列エレメントで構成されるストリング
Boolean	"true" または "false"
Date	Date オブジェクトのストリング表現
null	"null"
Number、int、uint	数値のストリング表現
Object	インスタンスが null の場合は null、それ以外の場合は [object Object]

シンタックス

言語のシンタックスは、実行可能なコードを記述する際に遵守する必要がある一連の規則を定義します。

大文字と小文字の区別

ActionScript 3.0 は、大文字と小文字を区別する言語です。スペルが同じで大文字か小文字かだけが異なる識別子は、異なる識別子と見なされます。例えば、次のコードは 2 つの異なる変数を生成します。

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

ドットシンタックス

ドット演算子（.）は、オブジェクトのプロパティまたはメソッドにアクセスする方法を提供します。ドットシンタックスを使用すると、インスタンス名の後にドット演算子とプロパティまたはメソッドの名前を付けたものを使ってクラスのプロパティまたはメソッドを参照できます。例えば、次のようなクラス定義があるとします。

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

ドットシンタックスを使用して、次のコードで作成されたインスタンス名を使用する prop1 プロパティと method1() メソッドにアクセスできます。

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

パッケージを定義するときに、ドットシンタックスを使用することができます。ドット演算子を使用してネストされたパッケージを参照します。例えば、EventDispatcher クラスは、flash というパッケージ内でネストされている events というパッケージにあります。次の式を使用して、events パッケージを参照することができます。

```
flash.events
```

この式を使用して、EventDispatcher クラスを参照することもできます。

```
flash.events.EventDispatcher
```

スラッシュシンタックス

ActionScript 3.0 では、スラッシュシンタックスはサポートされていません。スラッシュシンタックスは、ActionScript の以前のバージョンで、ムービークリップまたは変数のパスを指定するために使用されていました。

リテラル

リテラルは、コードに直接表示される値です。次の例は、すべてのリテラルを示します。

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

リテラルをグループ化して、複合リテラルを形成することもできます。配列リテラルは、角括弧（[]）で囲まれ、カンマを使用して配列エレメントが区切られます。

配列リテラルを使用して配列を初期化することができます。次の例では、配列リテラルを使用して初期化される 2 つの配列を示します。new ステートメントを使用して、複合リテラルを Array クラスのコンストラクターにパラメーターとして渡すことができますが、ActionScript コアクラスの Object、Array、String、Number、int、uint、XML、XMLList、および Boolean のインスタンスをインスタンス化するときに、リテラル値を直接割り当てることもできます。

```
// 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];
```

リテラルは汎用オブジェクトの初期化にも使用できます。汎用オブジェクトは Object クラスのインスタンスです。オブジェクトリテラルは、中括弧（{}）で囲まれ、カンマを使ってオブジェクトプロパティが区切られます。各プロパティはコロン（:）で宣言されます。コロンにより、プロパティ名とプロパティ値が区切られます。

new ステートメントを使用して汎用オブジェクトを作成し、オブジェクトリテラルをパラメーターとして Object クラスコンストラクターに渡すか、宣言するインスタンスにオブジェクトリテラルを直接割り当てることができます。次の例では、新しい汎用オブジェクトを作成し、3 つのプロパティ（propA、propB、および propC）を使ってオブジェクトを初期化する 2 種類の代替方法を示します。各プロパティの値は、1、2、および 3 に設定されます。

```
// 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};
```

関連項目

ストリングの操作

正規表現の使用

XML 変数の初期化

セミコロン

セミコロン（;）を使用してステートメントを終了することができます。セミコロンを省略した場合は、コンパイラーはコードの各行が 1 つのステートメントであると見なします。セミコロンを使用してステートメントの終了を示すことに慣れているプログラマーが多いので、セミコロンを使用してステートメントを終了するとコードが読みやすくなります。

セミコロンを使用してステートメントを終了すると、1 行に複数のステートメントを配置することができますが、こうするとコードが読みにくくなることがあります。

括弧

ActionScript 3.0 では、括弧（()）に 3 つの使用方法があります。1 つ目は、括弧を使用して、式内の演算の順序を変更します。括弧内にグループ化された演算は、常に最初に実行されます。例えば、括弧は次のコードの演算の順序を変更するために使用されます。

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

2 つ目は、次の例に示すように、括弧にカンマ演算子（,）を使用して、一連の式を評価し、最後に実行された式の結果を返します。

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

3 つ目は、次の例に示すように、括弧を使用して 1 つまたは複数のパラメーターを関数またはメソッドに渡します。この例では、trace() 関数にストリング値を渡します。

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

コメント

ActionScript 3.0 コードは、単一行コメントと複数行コメントの 2 種類のコメントをサポートします。このコメントメカニズムは、C++ や Java のコメントメカニズムに似ています。コンパイラーは、コメントとしてマークされたテキストを無視します。

単一行コメントは、2 つのスラッシュ（//）で始まり、その行の最後までです。例えば、次のコードには単一行コメントが含まれています。

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

複数行コメントは、スラッシュとアスタリスク（/*）で始まり、アスタリスクとスラッシュ（*/）で終わります。

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

キーワードと予約語

予約語とは、単語が ActionScript で使用するために予約されているので、ユーザーがコードで識別子として使用できない単語です。予約語には、コンパイラーによってプログラム名前空間から削除されるレキシカルキーワードが含まれます。レキシカルキーワードを識別子として使用すると、コンパイラーはエラーを報告します。次の表は、ActionScript 3.0 のレキシカルキーワードの一覧です。

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

シンタックスキーワードと呼ばれるいくつかのキーワードがあります。このキーワードは識別子として使用できますが、コンテキストによっては特別な意味になります。次の表に、ActionScript 3.0 シンタックスのキーワードを示します。

each	get	set	namespace
include	dynamic	final	native
override	static		

さらに、将来の予約語と呼ばれる識別子もいくつかあります。これらの識別子は ActionScript 3.0 では予約されていませんが、その一部は ActionScript 3.0 を組み込んだソフトウェアでキーワードとして扱われることがあります。これらの識別子の多くはコードで使用可能ですが、使用しないことをお勧めします。今後のバージョンの言語でキーワードとして表示される可能性があるからです。

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

定数

ActionScript 3.0 では、定数を作成するために使用する const ステートメントがサポートされています。定数は、変更できない固定値を持つプロパティです。定数には値を 1 回だけ割り当てることができますが、定数の宣言のきわめて近くで割り当てる必要があります。例えば、定数がクラスのメンバーとして宣言されると、宣言の一部としてのみ、またはクラスコンストラクター内でのみ、その定数に値を割り当てることができます。

次のコードは、2 つの定数を宣言します。1 つ目の定数 MINIMUM には、宣言ステートメントの一部として値が割り当てられます。2 つ目の定数 MAXIMUM には、コンストラクター内で値が割り当てられます。strict モードでは初期化時にのみ定数値を割り当てることができるので、この例は standard モードでのみコンパイルされることに注意してください。

```
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
```

他の方法で定数に初期値を割り当てようとすると、エラーが発生します。例えば、クラス外部にある MAXIMUM の初期値を設定しようとすると、ランタイムエラーが発生します。

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

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

ActionScript 3.0 では、使用する広範囲の定数が定義されます。表記規則により、ActionScript の定数には大文字だけを使用し、単語はアンダースコア文字（_）で区切ります。例えば、MouseEvent クラス定義では、マウス入力に関連するイベントを表す定数にこの命名規則を使用します。

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

演算子

演算子とは、1 つまたは複数のオペランドを取り、1 つの値を返す特殊な関数です。オペランドとは、演算子が入力として使用する値で、通常はリテラル、変数、または式を取ります。例えば、次のコードでは、加算（+）および乗算（*）演算子をリテラルオペランド（2、3、および 4）と共に使用して、値を返します。この値は、続いて代入（=）演算子によって使用され、戻り値 14 を変数 sumNumber に代入します。

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

演算子には、単項、二項、および三項があります。単項演算子は、オペランドを 1 つだけ取ります。例えば、インクリメント（++）演算子は、オペランドを 1 つだけ取るため、単項演算子です。二項演算子は、オペランドを 2 つ取ります。例えば、除算（/）演算子はオペランドを 2 つ取ります。三項演算子はオペランドを 3 つ取ります。例えば、条件（?:）演算子は 3 つのオペランドを取ります。

一部の演算子はオーバーロードされます。つまり、渡されたオペランドの種類または数に応じて動作が異なります。加算（+）演算子は、オペランドのデータ型によって動作が異なるオーバーロード演算子の例です。オペランドが両方とも数値の場合、加算演算子は値の合計値を返します。オペランドが両方ともストリングの場合、加算演算子は 2 つのオペランドの連結を返します。次のコード例は、演算子の動作がオペランドによってどのように異なるかを示します。

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

演算子は、指定されたオペランドの数によっても動作が異なります。除算（-）演算子は、単項および二項演算子です。オペランドが 1 つだけ指定されると、除算演算子はオペランドを否定し、その結果を返します。オペランドが 2 つ指定されると、除算演算子は 2 つのオペランドの差を返します。次の例では、最初に単項演算子、次に二項演算子として使用される除算演算子を示します。

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

演算子の優先順位と結合性

演算子の優先順位と結合性により、演算子を処理する順序が決まります。算術プログラミングに慣れている開発者にとって、コンパイラーが乗算（*）演算子を加算（+）演算子より先に処理するのは自然なことですが、コンパイラーは、どの演算子を最初に処理するかについて明示的な指示を必要とします。このような指示を、総称して演算子の優先順位と呼びます。ActionScript では、括弧（()）を使用して変更できる、演算子のデフォルトの優先順位が定義されています。例えば、次のコードは、前の例のデフォルトの優先順位を変更して、乗算演算子の前に加算演算子を処理するようにコンパイラーに強制します。

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

同じ優先順位の複数の演算子が同じ式にある状況があります。このような状況では、コンパイラーは結合性のルールを使用して、最初に処理する演算子を決定します。代入演算子を除くすべてのバイナリ演算子は左結合となります。つまり、左にある演算子が右にある演算子よりも先に処理されます。代入演算子と条件（?:）演算子は右結合となります。つまり、右にある演算子が左にある演算子よりも先に処理されます。

例えば、「より小さい」（<）および「より大きい」（>）演算子について考えてみます。これらの演算子の優先順位は同じです。両方の演算子を同じ式で使用すると、左の演算子が最初に処理されます。これは、両方の演算子が左結合であるためです。つまり、次の 2 つのステートメントでは同じ出力が得られます。

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

「より大きい」演算子が最初に処理され、その結果は値 true になります。これはオペランド 3 がオペランド 2 より大きいためです。次に、値 true がオペランド 1 と共に「より小さい」演算子に渡されます。次のコードは、この中間状態を表します。

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

「より小さい」演算子は、値 true を数値 1 に変換し、この数値を 2 番目のオペランド 1 と比較して、値 false を返します（つまり、値 1 は 1 より小さくない）。

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

デフォルトの左結合を括弧演算子を使用して変更できます。演算子とオペランドを括弧で囲んで、" より小さい " 演算子を最初に処理するようにコンパイラーに指示できます。次の例では、括弧演算子を使用して、前の例と同じ数値で異なる出力を生成します。

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

「より小さい」演算子が最初に処理され、その結果は値 false になります。これは、オペランド 2 がオペランド 1 より小さくないためです。次に、値 false がオペランド 3 と共に「より大きい」演算子に渡されます。次のコードは、この中間状態を表します。

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

「より大きい」演算子は、値 false を数値 0 に変換し、この数値を別のオペランド 3 と比較して、値 true を返します（つまり、値 3 は 0 より大きい）。

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

次の表は、ActionScript 3.0 の演算子を優先順位の高いものから順に示します。同じ行に示されている演算子は優先順位が同じです。各行に示されている演算子はその下に表示される行の演算子より優先順位が高くなります。

グループ	**演算子**
基本	[] {x:y} () f(x) new x.y x[y] <></> @ :: ..
後置	x++ x--
単項	++x --x + - ~ ! delete typeof void
乗法	* / %
加算	+ -
ビット単位シフト	<< >> >>>
関係	< > <= >= as in instanceof is
等価	== != === !==
ビット単位の論理積（AND）	&
ビット単位の排他的論理和（XOR）	^
ビット単位の論理和（OR）	|
論理積（AND）	&&
論理和（OR）	||
条件	?:
代入	= *= /= %= += -= <<= >>= >>>= &= ^= |=
カンマ	,

基本演算子

基本演算子は、Array リテラルおよび Object リテラルの作成、式のグループ化、関数の呼び出し、クラスインスタンスのインスタンス化、およびプロパティへのアクセスに使用する演算子です。

次の表の基本演算子の優先順位はすべて同じです。E4X 仕様に規定されている演算子には、（E4X）と表示されます。

演算子	実行される演算
[]	配列の初期化
{x:y}	オブジェクトの初期化
()	式のグループ化
f(x)	関数の呼び出し
new	コンストラクターの呼び出し
x.y x[y]	プロパティへのアクセス
<></>	XMLList オブジェクトの初期化（E4X）
@	属性へのアクセス（E4X）
::	名前の修飾（E4X）
..	子孫 XML エレメントへのアクセス（E4X）

後置演算子

後置演算子は、1 つの演算子を取り、演算子の値をインクリメントまたはデクリメントします。これらの演算子は単項演算子ですが、優先順位が高いことと、その特別なビヘイビアーにより、他の単項演算子からは区別して分類されます。後置演算子をより大きい式の一部として使用する場合、後置演算子が処理される前に、式の値が返されます。例えば、次のコードは、値がインクリメントされる前に、式の値 xNum++ が返される方法を示します。

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

次の表の後置演算子の優先順位はすべて同じです。

演算子	実行される演算
++	インクリメント（後置方式）
--	デクリメント（後置方式）

単項演算子

単項演算子は 1 つのオペランドを取ります。このグループのインクリメント（++）演算子とデクリメント（--）演算子は前置演算子であり、式ではオペランドの前に表示されます。前置演算子が後置演算子と異なるのは、インクリメント演算またはデクリメント演算が、式全体の値が返される前に完了することです。例えば、次のコードは、値がインクリメントされてから、式の値 ++xNum が返される方法を示します。

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

次の表の単項演算子の優先順位はすべて同じです。

演算子	実行される演算
++	インクリメント（前置方式）
--	デクリメント（後置方式）
+	単項プラス
-	単項マイナス
!	論理否定（NOT）
~	ビット単位の論理否定（NOT）
delete	プロパティの削除
typeof	タイプ情報を返す
void	未定義の値を返す

乗法演算子

乗法演算子は、2 つのオペランドを取り、乗算、除算、または剰余計算を実行します。

次の表の乗法演算子の優先順位はすべて同じです。

演算子	実行される演算
*	乗算
/	除算
%	剰余

加算演算子

加算演算子は 2 つのオペランドを取り、加算または減算を実行します。次の表の加算演算子の優先順位はすべて同じです。

演算子	実行される演算
+	加算
-	減算

ビット単位シフト演算子

ビット単位シフト演算子は 2 つのオペランドを取り、2 番目のオペランドで指定された範囲で最初のオペランドのビットをシフトします。次の表のビット単位シフト演算子の優先順位はすべて同じです。

演算子	実行される演算
<<	ビット単位の左シフト
>>	ビット単位の右シフト
>>>	ビット単位の符号なし右シフト

関係演算子

関係演算子は 2 つのオペランドを取り、値を比較してブール値を返します。次の表の関係演算子の優先順位はすべて同じです。

演算子	実行される演算
<	より小さい
>	より大きい
<=	より小さいか等しい
>=	より大きいか等しい
as	データ型のチェック
in	オブジェクトプロパティのチェック
instanceof	プロトタイプチェーンのチェック
is	データ型のチェック

等価演算子

等価演算子は 2 つのオペランドを取り、値を比較してブール値を返します。次の表の等価演算子の優先順位はすべて同じです。

演算子	実行される演算
==	等価
!=	不等価
===	厳密な等価
!==	厳密な不等価

ビット論理演算子

ビット論理演算子は 2 つのオペランドを取り、ビットレベルの論理演算を実行します。ビット論理演算子の優先順位は異なりますが、高いものから順に次の表に示します。

演算子	実行される演算
&	ビット単位の論理積（AND）
^	ビット単位の排他的論理和（XOR）
|	ビット単位の論理和（OR）

論理演算子

論理演算子は 2 つのオペランドを取り、ブール値の結果を返します。論理演算子の優先順位は異なりますが、高いものから順に次の表に示します。

演算子	実行される演算
&&	論理積（AND）
||	論理和（OR）

条件演算子

条件演算子は三項演算子であり、3 つのオペランドを取ります。条件演算子は、if..else 条件ステートメントを適用する簡易的な方法です。

演算子	実行される演算
?:	条件

代入演算子

代入演算子は 2 つのオペランドを取り、他のオペランドの値に基づいて 1 つのオペランドに値を割り当てます。次の表の代入演算子の優先順位はすべて同じです。

演算子	実行される演算
=	代入
*=	乗算後代入
/=	除算後代入
%=	剰余を代入
+=	加算後代入
-=	減算後代入
<<=	ビット単位での左シフト後代入
>>=	ビット単位での右シフト後代入
>>>=	ビット単位での符号なし右シフト後代入
&=	ビット単位の論理積（AND）を代入
^=	ビット単位の排他的論理和（XOR）を代入
|=	ビット単位の排他的論理和（OR）を代入

条件

ActionScript 3.0 は、プログラムフローの制御に使用できる 3 つの基本的な条件ステートメントを提供します。

if..else

if..else 条件ステートメントを使用して、条件をテストし、その条件が存在する場合はコードブロックを実行し、その条件が存在しない場合は代わりのコードブロックを実行します。例えば、次のコードは x の値が 20 を超えるかどうかをテストし、超える場合は trace() 関数を生成し、超えない場合は異なる trace() 関数を生成します。

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

別のコードブロックを実行しない場合は、else ステートメントのない if ステートメントを使用できます。

if..else if

if..else if 条件ステートメントを使用すると、複数の条件をテストできます。例えば、次のコードは、x の値が 20 を超えるかどうかをテストすると共に、x が負の値かどうかをテストします。

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

if または else ステートメントに続くステートメントが 1 つだけの場合は、ステートメントを中括弧で囲む必要はありません。例えば、次のコードは中括弧を使用していません。

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

ただし、後で中括弧のない条件ステートメントがステートメントに追加された場合に予期しない動作を引き起こす可能性があるため、常に中括弧を使用することをお勧めします。例えば、次のコードで positiveNums の値は、条件が true と評価されるかどうかに関係なく 1 増加します。

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

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

trace(positiveNums); // 1
```

switch

switch ステートメントは、同じ条件式に依存する複数の実行パスが存在する場合に便利です。一連の if..else if ステートメントに似た機能ですが、多少読みやすくなっています。switch ステートメントは、ブール値の条件をテストするのではなく、式を評価し、その結果を使用して実行するコードブロックを決定します。コードブロックは、case ステートメントで始まり、break ステートメントで終わります。例えば、次の switch ステートメントは、Date.getDay() メソッドによって返された日数に基づいて曜日を出力します。

```
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;
}
```

ループ

ループステートメントを使用すると、一連の値または変数を使用して特定のコードブロックを繰り返し実行できます。コードブロックは常に中括弧（{}）で囲むことをお勧めします。コードブロックに含まれるステートメントが 1 つのみ場合は中括弧を省略できますが、条件の場合と同じ理由から省略することはお勧めしません。後で追加したステートメントがコードブロックから誤って除外される可能性が高くなるからです。コードブロックに含めるステートメントを後で追加するときに、必要な中括弧の追加を忘れた場合、そのステートメントはループの一部としては実行されません。

for

for ループを使用して、特定の値範囲で変数を繰り返し処理することができます。for ステートメントには、3 つの式を指定する必要があります。3 つの式とは、初期値に設定する変数、ループがいつ終了するかを指定する条件ステートメント、および各ループで変数の値を変更する式です。例えば、次のコードは 5 回ループします。変数 i の値は 0 で始まり、4 で終了し、出力は数値 0 ～ 4 の数値を取り、それぞれが独自の行に出力されます。

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

for..in

for..in ループは、オブジェクトのプロパティまたは配列のエレメントの繰り返し処理を行います。例えば、for..in ループを使用して、汎用オブジェクトのプロパティを繰り返し処理することができます（オブジェクトプロパティは特定の順番で保持されないので、プロパティは一見ランダムな順番のように表示されます）。

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

配列のエレメントの繰り返し処理を実行することもできます。

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

オブジェクトが sealed クラス（ビルトインクラスおよびユーザー定義クラスを含む）のインスタンスである場合、そのプロパティに対して繰り返し処理を実行することはできません。繰り返し処理を実行できるのは、ダイナミッククラスのプロパティのみに対してです。ダイナミッククラスのインスタンスの場合であっても、繰り返し処理を実行できるのは、動的に追加されるプロパティのみに対してです。

for each..in

for each..in ループは、XML または XMLList オブジェクトでタグ付けできるコレクションのアイテム、オブジェクトプロパティで保持される値、または配列のエレメントを繰り返し処理します。例えば、次のコードの抜粋に示すように、for each..in ループを使用して汎用オブジェクトのプロパティを繰り返し処理できますが、for..in ループとは異なり、for each..in ループのイテレータ変数にはプロパティの名前の代わりにプロパティによって保持される値が含まれます。

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

次の例に示すように、XML または XMLList オブジェクトの繰り返し処理を実行することができます。

```
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
*/
```

以下の例が示すように、配列のエレメントの繰り返し処理を実行することもできます。

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

オブジェクトが sealed クラスのインスタンスである場合、そのプロパティの繰り返し処理を実行することはできません。ダイナミッククラスのインスタンスの場合であっても、クラス定義の一部として定義されるプロパティである固定プロパティの繰り返し処理を実行することはできません。

while

while ループは、条件が true である限り繰り返される if ステートメントに似ています。例えば、次のコードは for ループの例と同じ出力を生成します。

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

for ループの代わりに while ループを使用する 1 つの短所は、while ループを使用すると無限ループを記述する可能性が高いことです。カウンター変数をインクリメントする式を省略した場合、for ループのサンプルコードはコンパイルされませんが、while ループのサンプルコードはコンパイルされます。i をインクリメントする式がないと、ループは無限ループになります。

do..while

do..while ループは、コードブロックが実行された後に条件がチェックされるので、少なくとも 1 回はコードブロックが実行される while ループです。次のコードは、条件を満たさない場合でも出力を生成する do..while ループの単純な例を示します。

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

関数

関数 は、特定のタスクを実行し、プログラム内で再利用できるコードブロックです。ActionScript 3.0 には、メソッドと関数クロージャの 2 種類の関数があります。関数をメソッドと呼ぶか関数クロージャと呼ぶかは、関数が定義されたコンテキストによって決まります。関数をクラス定義の一部として定義した場合、またはオブジェクトのインスタンスに関連付けた場合は、メソッドと呼びます。関数がその他の方法で定義された場合は、関数クロージャと呼びます。

ActionScript では、関数は非常に重要です。ActionScript 1.0 では、例えば、class キーワードが存在しなかったので、「クラス」はコンストラクター関数で定義されました。その後、class キーワードが追加されましたが、ActionScript をフルに活用するには、関数について理解しておくことが重要です。しかし、これは、ActionScript の関数が C++ や Java などの言語の関数と同じように動作することを期待するプログラマーにとっては難しい場合があります。経験豊富なプログラマーにとっては基本的な関数の定義や呼び出しは問題がありませんが、ActionScript の関数の高度な機能の中には説明が必要なものもあります。

基本的な関数の概念

関数の呼び出し

関数を呼び出すには、識別子の後に括弧（()）を使用します。関数に渡す関数パラメーターを括弧で囲みます。例えば、trace() 関数は ActionScript 3.0 のトップレベルの関数です。

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

関数をパラメーターなしで呼び出す場合、空の括弧のペアを使用する必要があります。例えば、パラメーターを取らない Math.random() メソッドは、乱数を生成します。

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

独自の関数の定義

ActionScript 3.0 の関数を定義するには、function ステートメントを使用する方法と関数式を使用する方法の 2 つがあります。どちらの方法を選択するかは、プログラミングスタイルをより静的にするか動的にするかによって決まります。静的な、つまり strict モードのプログラミングの方を好む場合は、function ステートメントで関数を定義します。そうすることに特定の必要性がある場合は、関数式で関数を定義します。関数式は、動的、つまり standard モードのプログラミングでより頻繁に使用されます。

function ステートメント

function ステートメントは、strict モードで関数を定義するのに適しています。function ステートメントは、function キーワードで始まり、次のアイテムが続きます。

- 関数の名前
- 括弧で囲まれたカンマ区切りリストで指定されたパラメーター
- 関数の本体、つまり関数を呼び出したときに実行される中括弧で囲まれた ActionScript コード

例えば、次のコードは、パラメーターを定義する関数を作成し、ストリング「hello」をパラメーター値として使用して関数を呼び出します。

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

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

関数式

関数を宣言する 2 つ目の方法は、代入ステートメントに関数式を使用することです。関数式は、関数リテラルまたは匿名関数とも呼ばれます。これは、旧バージョンの ActionScript で広く使用されている、より冗長になる方法です。

関数式を持つ代入ステートメントは、var キーワードで始まり、次のアイテムが続きます。

- 関数の名前
- コロン演算子（:）
- データ型を示すための Function クラス
- 代入演算子（=）
- function キーワード
- 括弧で囲まれたカンマ区切りリストで指定されたパラメーター
- 関数の本体、つまり関数を呼び出したときに実行される中括弧で囲まれた ActionScript コード

  例えば、次のコードは関数式を使用して traceParameter 関数を宣言します。

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

  function ステートメントとは異なり、関数の名前を指定していないことに注意してください。関数式が function ステートメントと異なるもう 1 つの特徴は、関数式はステートメントではなく式であることです。つまり、function ステートメントのように、関数式はそれだけでは成立しません。関数式は、通常は代入ステートメントなど、ステートメントの一部としてのみ使用することができます。次の例は、配列エレメントに代入された関数式を示します。

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

ステートメントと式の選択

原則として、式を使用する必要がある場合を除いて、function ステートメントを使用します。function ステートメントは、関数式より簡潔で、strict モードと standard モードで一貫した使いやすさを提供します。

function ステートメントは、関数式を含む代入ステートメントより読みやすくなります。function ステートメントを使用すると、コードが簡潔になり、var と function キーワードを両方使用する必要がある関数式よりわかりやすくなります。

function ステートメントは、2 つのコンパイラーモードで一貫した使いやすさを提供します。つまり、strict モードおよび standard モードの両方でドットシンタックスを使用し、function ステートメントを使用して宣言されたメソッドを呼び出すことができます。これは、関数式を使用して宣言されたメソッドには必ずしも当てはまりません。例えば、次のコードは、

2 つのメソッドで Example というクラスを定義します。関数式で宣言される methodExpression() と function ステートメントで宣言される methodStatement() です。strict モードでは、ドットシンタックスを使用して、methodExpression() メソッドを呼び出すことはできません。

```
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
```

関数式は、実行時、つまり動的なビヘイビアーを中心にしたプログラミングに適しています。strict モードを使用し、関数式で宣言されるメソッドを呼び出す必要がある場合は、次の 2 つの方法のいずれかを使用することができます。1 つ目は、ドット（.）演算子ではなく、角括弧（[]）を使用してメソッドを呼び出す方法です。次のメソッドの呼び出しは strict モードと standard モードの両方で成功します。

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

2 つ目は、クラス全体をダイナミッククラスとして宣言することができます。この場合、ドット演算子を使用してメソッドを呼び出すことができますが、そのクラスのすべてのインスタンスで strict モードの一部の機能が犠牲になるという短所があります。例えば、ダイナミッククラスのインスタンスの未定義のプロパティにアクセスしようとした場合、コンパイラーはエラーを生成しません。

関数式が便利な場合があります。関数式の一般的な使用法は、1 回だけ使用された後で破棄される関数に使用することです。また、一般的な使用法ではありませんが、関数をプロトタイププロパティに関連付けるために使用します。詳しくは、プロトタイプオブジェクトを参照してください。

function ステートメントと関数式には、どちらを使用するかを選択する際に考慮する必要がある微妙な違いが 2 つあります。1 つ目の違いは、関数式は、メモリ管理およびガベージコレクションに関してオブジェクトとして単独で存在しません。つまり、配列エレメントやオブジェクトプロパティなどの別のオブジェクトに関数式を割り当てると、コード内にその関数式への唯一の参照が作成されます。関数式が関連付けられている配列またはオブジェクトがスコープ外に移動するか、使用できなくなった場合、関数式にアクセスできなくなります。配列またはオブジェクトが削除されると、関数式が使用するメモリがガベージコレクションの対象となります。つまり、メモリは他の目的で再要求され、再利用される対象となります。

次の例では、関数式の場合、関数式が割り当てられているプロパティが削除されると、関数が利用できなくなることを示します。クラス Test は動的です。つまり、関数式を保持する functionExp というプロパティを追加できます。functionExp() 関数は、ドット演算子を使用して呼び出すことができますが、functionExp プロパティが削除されると、関数にアクセスできなくなります。

```
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
```

その一方で、関数が最初に function ステートメントで定義された場合、関数はそのオブジェクトとして存在し、関連付けられているプロパティを削除しても存在します。delete 演算子はオブジェクトのプロパティでのみ動作するので、関数 stateFunc() 自体を削除する呼び出しであっても動作しません。

```
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
```

function ステートメントと関数式の 2 つ目の違いは、function ステートメントは、関数ステートメントの前に現れるステートメント内を含む、定義されたスコープ全体で存在することです。対照的に、関数式はそれ以降のステートメントに対してのみ定義されます。例えば、次のコードは、定義される前に scopeTest() 関数を正常に呼び出します。

```
statementTest(); // statementTest

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

関数式は定義されるまで使用できないので、次のコードはランタイムエラーを発生します。

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

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

関数からの戻り値

関数から値を返すには、return ステートメントの後に、返す式またはリテラル値を使用します。例えば、次のコードは、パラメーターを表す式を返します。

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

return ステートメントは関数を終了させるので、次のように return ステートメントの下にあるステートメントは実行されないことに注意してください。

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

strict モードでは、戻り値の型を指定するように選択した場合、適切な型の値を返す必要があります。例えば、次のコードは有効な値を返さないので、strict モードではエラーが発生します。

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

ネストされた関数

関数をネスト、つまり別の関数内で関数を宣言することができます。ネストされた関数は、関数への参照が外部コードに渡される場合を除いて、その親関数内でのみ使用できます。例えば、次のコードは getNameAndVersion() 関数内に 2 つのネストされた関数を宣言します。

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

ネストされた関数が外部コードに渡される場合、関数クロージャとして渡されます。つまり、関数が定義されるとき、スコープ内にある定義を保持します。詳しくは、関数のスコープを参照してください。

関数のパラメータ

ActionScript 3.0 は、ActionScript を初めて使用するプログラマーには斬新な関数パラメーターの機能を備えています。値または参照によってパラメーターを渡す方法は、ほとんどのプログラマーが使い慣れている方法ですが、arguments オブジェクトと ... (rest) パラメーターは、多くのプログラマーにとって初めての方法と思われます。

引数の値渡しと参照渡し

多くのプログラミング言語では、値渡しと参照渡しによるパラメーターの受け渡しの違いを理解しておくことが重要です。この違いは、コードの設計方法に影響します。

値渡しとは、関数内で使用するためにパラメーターの値がローカル変数にコピーされることです。参照渡しとは、実際の値ではなく、パラメーターへの参照のみが渡されることです。実際の値がコピーされるのではなく、パラメーターとして渡される、変数への参照が作成され、関数内で使用するためにローカル変数に割り当てられます。関数外の変数への参照では、ローカル変数は元の変数の値を変更する機能を提供します。

ActionScript 3.0 では、値はすべてオブジェクトとして格納されているため、すべてのパラメーターは参照渡しです。しかし、Boolean、Number、int、uint、String などのプリミティブデータ型に属するオブジェクトには、値渡しのように動作する特別な演算子があります。例えば、次のコードは、xParam と yParam という 2 つの int 型パラメーターを定義する passPrimitives() という関数を作成します。この 2 つのパラメーターは、passPrimitives() 関数の本体内で宣言されるローカル変数に似ています。関数がパラメーター xValue および yValue で呼び出されると、パラメーター xParam および yParam は、xValue と yValue で表される int オブジェクトへの参照で初期化されます。パラメーターはプリミティブなので、値渡しのように動作します。xParam と yParam は、初期状態では xValue と yValue オブジェクトへの参照のみが含まれますが、関数本体内の変数の変更によりメモリ内に新しい値のコピーが生成されます。

```
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
```

passPrimitives() 関数内では、xParam および yParam の値はインクリメントされますが、最後の trace ステートメントに示されているように、これは xValue および yValue の値に影響を与えません。パラメーターが変数 xValue および yValue と同じ名前である場合でも同じです。これは、関数内の xValue および yValue は関数の外部にある同じ名前の変数とは別に存在するメモリ内の新しい位置を参照するためです。

プリミティブデータ型に属さないその他すべてのオブジェクトは、常に参照によって渡され、元の変数の値を変更する機能を提供します。例えば、次のコードは、2 つのプロパティ x および y を持つ objVar というオブジェクトを作成します。このオブジェクトは、passByRef() 関数にパラメーターとして渡されます。オブジェクトはプリミティブ型ではないため、参照渡しで渡されるだけではなく、参照のままでもあります。つまり、関数内のパラメーターを変更すると、関数の外側にあるオブジェクトのプロパティも影響を受けます。

```
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
```

objParam パラメーターは、グローバル変数 objVar と同じオブジェクトを参照します。この例の trace ステートメントからもわかるように、objParam オブジェクトの x および y プロパティを変更すると、objVar オブジェクトにも反映されます。

デフォルトのパラメーター値

ActionScript 3.0 では、関数にデフォルトのパラメーター値を宣言できます。デフォルトのパラメーター値を使用した関数の呼び出しでデフォルト値のパラメーターが省略されると、そのパラメーターの関数定義で指定された値が使用されます。デフォルト値のパラメーターはすべてパラメーターリストの末尾に配置する必要があります。デフォルト値として割り当てられた値はコンパイル時定数である必要があります。パラメーターのデフォルト値が存在すると、そのパラメーターはオプションパラメーターになります。デフォルト値を持たないパラメーターは、必須パラメーターと見なされます。

例えば、次のコードは、3 つのパラメーターで関数を作成します。このうち、2 つのパラメーターにはデフォルト値があります。パラメーター 1 つだけで関数を呼び出す場合、そのパラメーターのデフォルト値が使用されます。

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

arguments オブジェクト

パラメーターが関数に渡されると、arguments オブジェクトを使用して関数に渡されたパラメーターについての情報にアクセスできます。arguments オブジェクトには、次のようないくつかの重要な側面があります。

- arguments オブジェクトは、関数に渡されるすべてのパラメーターを含む配列です。
- arguments.length プロパティは、関数に渡されるパラメーターの数を報告します。
- arguments.callee プロパティを使用すると、関数自体を参照することができます。これは関数式の再帰呼び出しに便利です。

  注意：パラメーターが arguments という名前の場合、または ... (rest) パラメーターを使用する場合、arguments オブジェクトは利用できません。

  関数の本体内で arguments オブジェクトが参照されている場合、ActionScript 3.0 では、関数呼び出しで関数定義内で定義されているパラメーターより多いパラメーターを指定できますが、パラメーター数が必須パラメーター数（およびオプションで任意パラメーター数）と一致しない場合は strict モードでコンパイルエラーが生成されます。arguments オブ

ジェクトの配列を使用すると、関数に渡されるパラメーターが関数定義内で定義されたかどうかにかかわらず、このパラメーターにアクセスできます。standard モードでのみコンパイルされる次の例では、arguments 配列と arguments.length プロパティを使用して、traceArgArray() 関数に渡されるすべてのパラメーターをトレースします。

```
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
```

arguments.callee プロパティは、再帰を作成する場合に匿名関数でよく使用されます。このプロパティを使用すると、コードに柔軟性を持たせることができます。開発過程で再帰関数の名前が変更された場合でも、関数名ではなく arguments.callee を使用していれば、関数本体内の再帰呼び出しの変更について考慮する必要はありません。arguments.callee プロパティは、次の関数式で使用して再帰を有効にします。

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

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

関数宣言に ... (rest) パラメーターを使用する場合は、arguments オブジェクトは利用できません。代わりに、宣言したパラメーター名を使用してパラメーターにアクセスする必要があります。

ストリング「arguments」をパラメーター名として使用しないようにする必要があります。このストリングは arguments オブジェクトをシャドウするためです。例えば、関数 traceArgArray() が arguments パラメーターが追加されるように記述し直されると、関数本体内の arguments への参照は、arguments オブジェクトではなく、このパラメーターを参照します。次のコードは出力を作成しません。

```
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
```

旧バージョンの ActionScript の arguments オブジェクトには、caller という名前のプロパティも含まれていました。これは、現在の関数を呼び出した関数への参照です。ActionScript 3.0 には caller プロパティはありませんが、呼び出し関数の参照が必要な場合は、それ自体を参照する特別なパラメーターを渡すように呼び出し関数を変更することができます。

...(rest) パラメーター

ActionScript 3.0 は、...(rest) パラメーターと呼ばれる新しいパラメーター宣言を導入しています。このパラメーターを使用すると、任意の数のカンマ区切りのパラメーターを受け入れる配列パラメーターを指定できます。パラメーターには、予約語ではない名前を指定することができます。このパラメーター宣言は、指定される最後のパラメーターである必要があります。このパラメーターを使用すると、arguments オブジェクトは使用できなくなります。... (rest) パラメーターには arguments 配列および arguments.length プロパティと同じ機能がありますが、arguments.callee のような機能はありません。... (rest) パラメーターを使用する前に、arguments.callee を使用する必要はありません。

次の例は、arguments オブジェクトの代わりに ... (rest) パラメーターを使用して、traceArgArray() 関数を書き直したものです。

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

... (rest) パラメーターは、最後にパラメーターとしてリストする限り、他のパラメーターと共に使用することもできます。次の例は、最初のパラメーター x を int 型にし、2 つ目のパラメーターとして ... (rest) パラメーターを使用するように、traceArgArray() 関数を変更します。最初のパラメーターは ...(rest) パラメーターで作成された配列の一部でなくなっているので、出力は最初の値をスキップします。

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

オブジェクトとしての関数

ActionScript 3.0 の関数はオブジェクトです。関数を作成する場合、パラメーターとして別の関数に渡すことができるオブジェクトを作成するだけではなく、プロパティとメソッドも関連付けることができます。

パラメーターとして別の関数に渡される関数は、値渡しではなく、参照渡しによって渡されます。関数をパラメーターとして渡す場合、識別子のみを使用し、メソッドを呼び出すために使用する括弧は使用しません。例えば、次のコードは、addEventListener() メソッドへのパラメーターとして clickListener() という関数を渡します。

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

ActionScript を初めて使用するプログラマーには奇妙に思えるかもしれませんが、他のオブジェクトと同様に関数にプロパティおよびメソッドを含めることができます。実際には、どの関数にも、その関数用に定義されたパラメーターの数を格納する length という読み取り専用プロパティがあります。これは、関数に渡されたパラメーターの数を報告する

arguments.length プロパティとは異なります。ActionScript では、関数に渡されたパラメーターの数がその関数用に定義されたパラメーターの数を上回ってもかまいません。次の例では、strict モードでは渡されたパラメーターの数と定義されたパラメーターの数が完全に一致する必要があるので、standard モードでのみコンパイルされています。この例は 2 つのプロパティの違いを示します。

```
// 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 */
```

standard モードで独自の関数プロパティを定義するには、関数本体の外側で定義します。関数プロパティは、関数に関連する変数の状態を保存できる準静的なプロパティになります。例えば、特定の関数が呼び出される回数を追跡するとします。ゲームを記述していて、ユーザーが特定のコマンドを使用する回数を追跡する場合に、こうした機能は便利ですが、この場合静的クラスプロパティを使用することもできます。strict モードではダイナミックプロパティを関数に追加できないので、次の例は standard モードでのみコンパイルされます。この例では、関数宣言の外部に関数プロパティを作成し、関数が呼び出されるたびにプロパティをインクリメントします。

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

someFunction.counter = 0;

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

関数のスコープ

関数のスコープは、プログラム内の関数を呼び出すことができる場所だけではなく、関数がアクセスできる定義も決定します。変数識別子に適用される同じスコープの規則が関数識別子にも適用されます。グローバルスコープで宣言された関数は、コード全体で使用することができます。例えば、ActionScript 3.0 には、isNaN()、parseInt() などのコード内のどこでも使用できるグローバル関数があります。別の関数内で宣言された、ネストされた関数は、関数が宣言されている関数内のどの場所でも使用できます。

スコープチェーン

関数が実行を開始するたびに、多数のオブジェクトおよびプロパティが作成されます。最初に、アクティベーションオブジェクトという特別なオブジェクトが作成され、そこには関数本体で宣言されるパラメーターとローカル変数または関数が格納されます。アクティベーションオブジェクトは内部メカニズムであるため、そこに直接アクセスすることはできません。次に、スコープチェーンが作成され、そこにはランタイムによって識別子宣言の有無がチェックされるオブジェクトが列挙されたリストが含まれます。実行されるどの関数にも、内部プロパティに格納されるスコープチェーンがあります。ネストされた関数の場合は、スコープチェーンはそのアクティベーションオブジェクトから始まり、その後に親関数のアクティベーションオブジェクトが続きます。このように、スコープチェーンはグローバルオブジェクトに達するまで続きます。グローバルオブジェクトは、ActionScript プログラムが起動すると作成され、すべてのグローバル変数および関数を含みます。

関数クロージャ

関数クロージャは、関数の静的なスナップショットとそのレキシカル環境を含むオブジェクトです。関数のレキシカル環境には、関数のスコープチェーン内のすべての変数、プロパティ、メソッド、およびオブジェクトがその値と共に含まれます。関数クロージャは、オブジェクトまたはクラスとは別に、関数が実行されるたびに作成されます。関数クロージャはその関数クロージャが定義されたスコープを保持することから、関数がパラメーターまたは戻り値として別のスコープに渡されると興味深い結果が生まれます。

例えば、次のコードは 2 つの関数を作成します。作成される関数の foo() は、矩形の面積を計算する rectArea() という名前のネストされた関数を返し、bar() は、foo() を呼び出し、myProduct という名前の変数に返された関数クロージャを格納します。bar() 関数が独自のローカル変数 x（値 2 を持つ）を定義する場合でも、関数クロージャ myProduct() が呼び出されると、関数 foo() で定義された変数 x（値 40 を持つ）を保持します。このため、bar() 関数は値 8 ではなく、値 160 を返します。

```
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
```

メソッドは、そのメソッドが作成されたレキシカル環境に関する情報も維持するように動作します。この特性が最も顕著なのは、メソッドがバインドメソッドを作成するそのインスタンスから抽出されるときです。関数クロージャとバインドメソッドの主な違いは、バインドメソッドの this キーワードの値は常に最初に関連付けられたインスタンスを参照しますが、関数クロージャでは this キーワードの値が変更可能という点です。

第 4 章：ActionScript のオブジェクト指向プログラミング

オブジェクト指向プログラミングの概要

オブジェクト指向プログラミング（OOP）は、オブジェクトとしてコードをグループ化することによって、プログラム内のコードを構成する方法です。OOP でのオブジェクトという用語は、情報（データ値）と機能を含む個々の要素を意味します。オブジェクト指向のアプローチを使用してプログラムを構成すると、特定の情報を、その情報に関連付けられた共通の機能と共にグループ化できます。例えば、アルバムタイトル、トラックタイトル、アーティスト名などの音楽情報を、その情報に関連付けられた共通の機能やアクション（「プレイリストにトラックを追加」や「このアーチストのすべての曲を再生」など）と共にグループ化できます。これらの項目が結合されて、1 つの項目、つまりオブジェクト（「Album」や「MusicTrack」など）になります。値と機能を一緒にバンドルすることには、さまざまな利点があります。最大の長所として、複数の変数を使用する必要がなく、1 個の変数を使用するだけで済むという点が挙げられます。さらに、関連する機能をまとめて管理することもできます。また、情報と機能を組み合わせることで、プログラムをより現実に即した形で構成することができます。

クラス

クラスは、オブジェクトの抽象表現です。クラスには、オブジェクトが保持できるデータの型およびオブジェクトが表すことができるビヘイビアーに関する情報が格納されます。相互にやり取りする 2、3 個のオブジェクトのみを含む小さなスクリプトを記述する場合には、このような抽象化の利点はわかりにくいかもしれません。しかし、プログラムのスコープが拡大するほど、管理しなければならないオブジェクトの数が増加します。そのような場合、クラスを使用することで、オブジェクトの作成方法と、オブジェクトの相互のやり取りの方法を制御しやすくなります。

ActionScript 1.0 では、Function オブジェクトを使用してクラスに似たコンストラクトを作成できました。ActionScript 2.0 では、class、extends などのキーワードによるクラスのサポートが正式に追加されました。ActionScript 3.0 では、ActionScript 2.0 で導入されたキーワードを引き続きサポートするだけでなく、新しい機能も追加されています。例えば、ActionScript 3.0 では、protected 属性と internal 属性により、アクセス制御が強化されています。また、final キーワードおよび override キーワードによる継承も、より的確に制御できるようになりました。

Java、C++、C# などのプログラミング言語でクラスを作成した経験があれば、ActionScript でも同じように作成できます。ActionScript には、class、extends、public など、これらのプログラミング言語と共通するキーワード名と属性名が多数存在します。

注意：Adobe ActionScript マニュアルでは、「プロパティ」という用語は、オブジェクトまたはクラスのあらゆるメンバー（変数、定数、メソッドなど）を示します。また、「クラス」と「静的」という用語はよく同じ意味で使用されますが、このマニュアルではこの 2 つの用語を区別します。例えば、「クラスプロパティ」という語句は、静的メンバーだけではなく、クラスのすべてのメンバーを指します。

クラス定義

ActionScript 3.0 のクラス定義には、ActionScript 2.0 のクラス定義に使用したシンタックスとほぼ同じシンタックスを使用します。クラス定義の正しいシンタックスは、class キーワードの後にクラス名が必要です。クラス名の後には、中括弧（{}）で囲まれたクラス本体が続きます。例えば、次のコードは、visible という名前の変数を 1 つ含む Shape という名前のクラスを作成します。

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

シンタックスが大きく変更された点の 1 つに、パッケージ内のクラス定義があります。ActionScript 2.0 では、クラスがパッケージ内部にある場合は、クラス宣言にパッケージ名を含める必要があります。ActionScript 3.0 では package ステートメントが導入されていますが、パッケージ名は、クラス宣言ではなくパッケージ宣言に含める必要があります。例えば、次のクラス宣言は、flash.display パッケージに含まれる BitmapData クラスを、ActionScript 2.0 および ActionScript 3.0 で定義する方法を示します。

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

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

クラス属性

ActionScript 3.0 では、次の 4 つの属性のいずれかを使用して、クラス定義を変更できます。

属性	定義
dynamic	実行時にインスタンスにプロパティを追加できます。
final	別のクラスによって拡張することはできません。
internal（デフォルト）	現在のパッケージ内の参照に対して表示されます。
public	すべての場所にある参照に対して表示されます。

internal を除くこれらの属性では、属性を明示的に含めて関連付けられたビヘイビアーを取得します。例えば、クラスを定義するときに dynamic 属性を含めなかった場合、実行時にクラスインスタンスにプロパティを追加することはできません。次のコードで示すように、クラス定義の開始部分に属性を配置することで、明示的に属性を割り当てます。

```
dynamic class Shape {}
```

このリストには、abstract という名前の属性は含まれていません。これは抽象クラスが ActionScript 3.0 でサポートされていないためです。また、このリストには、private および protected という名前の属性も含まれていません。これらの属性は、クラス定義内でのみ意味を持ち、クラス自体には適用できません。パッケージの外部でクラスをパブリックに表示しない場合は、パッケージ内にクラスを配置し、internal 属性でクラスを宣言します。または、internal 属性と public 属性の両方を省略することもできます。この場合は、コンパイラーが自動的に internal 属性を追加します。定義されたソースファイル内でのみ参照されるように、クラスを定義することもできます。それには、パッケージ定義の右中括弧の下、ソースファイルの最後にクラスを挿入します。

クラス本体

クラス本体は、中括弧で囲みます。クラス本体は、クラスの変数、定数、メソッドを定義します。次の例は、ActionScript 3.0 の Accessibility クラスの宣言を示します。

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

クラス本体内で名前空間を定義することもできます。次の例は、クラス本体内で名前空間を定義し、そのクラスのメソッドの属性として使用する方法を示します。

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

ActionScript 3.0 では、クラス本体内に、定義だけでなくステートメントを含めることもできます。クラス本体内にあるが、メソッド定義の外にあるステートメントは、1 度だけ実行されます。この実行は、クラス定義が最初に検出され、関連付けられたクラスオブジェクトが作成されるときに発生します。次の例には、外部関数である hello() と、クラスが定義されたときに確認メッセージを出力する trace ステートメントの呼び出しが含まれています。

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

ActionScript 3.0 では、同じクラス本体内で同じ名前を持つ静的プロパティとインスタンスプロパティを定義することができます。例えば、次のコードは、message という名前の静的変数と同じ名前のインスタンス変数を宣言します。

```
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
```

クラスプロパティの属性

ActionScript オブジェクトモデルの説明では、プロパティという用語は、変数、定数、メソッドなど、クラスのメンバーになり得るものすべてを意味します。しかし、Adobe Flash Platform 用 ActionScript 3.0 リファレンスガイドでは、この用語はより狭い意味で使用されます。このリファレンスでは、プロパティは、変数であるクラスメンバー、または getter メソッドか setter メソッドで定義されたクラスメンバーを指します。ActionScript 3.0 には、クラスのプロパティと共に使用できる一連の属性があります。次の表は、この属性の一覧です。

属性	定義
internal（デフォルト）	同じパッケージ内の参照に対して表示されます。
private	同じクラス内の参照に対して表示されます。
protected	同じクラスおよび派生クラス内の参照に対して表示されます。
public	すべての場所にある参照に対して表示されます。
static	プロパティが、クラスのインスタンスではなくクラスに属することを指定します。
UserDefinedNamespace	ユーザーが定義するカスタム名前空間の名前です。

アクセス制御名前空間の属性

ActionScript 3.0 には、クラス内で定義されたプロパティへのアクセスを制御する 4 つの特別な属性、public、private、protected、および internal があります。

public 属性は、スクリプト内の任意の場所でプロパティを表示します。例えば、パッケージ外部のコードでメソッドを使用するには、public 属性でメソッドを宣言する必要があります。これは、宣言されたキーワードが var、const、function のいずれであるかにかかわらず、あらゆるプロパティに当てはまります。

private 属性は、プロパティが定義されたクラス内の呼び出し元に対してのみプロパティを表示します。このビヘイビアーは、サブクラスがスーパークラスのプライベートプロパティにアクセスできる ActionScript 2.0 の private 属性とは異なります。ランタイムアクセスに関しても、ビヘイビアーが大きく変更されています。ActionScript 2.0 では、private キーワードによりアクセスが禁止されるのはコンパイル時のみで、実行時にはアクセスが禁止されません。ActionScript 3.0 では、これは当てはまりません。private とマークされたプロパティは、コンパイル時も実行時も使用することができません。

例えば、次のコードは、プライベート変数を 1 つ持つ PrivateExample という単純なクラスを作成し、クラス外部からこのプライベート変数にアクセスしようとします。

```
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.
```

ActionScript 3.0 では、ドット演算子（myExample.privVar）を使用してプライベートプロパティにアクセスしようとすると、strict モードを使用している場合にコンパイル時エラーになります。それ以外の場合は、プロパティアクセス演算子（myExample["privVar"]）を使用するときと同様に、エラーは実行時に報告されます。

次の表に、sealed（dynamic ではない）クラスに属するプライベートプロパティにアクセスしようとした結果の概要を示します。

	strict モード	**standard モード**
ドット演算子（.）	コンパイル時エラー	ランタイムエラー
角括弧演算子（[]）	ランタイムエラー	ランタイムエラー

dynamic 属性で宣言されたクラスでは、プライベート変数にアクセスしようとしても、ランタイムエラーにはなりません。ただし、変数が参照不可になるため、undefined という値が返されます。しかし、strict モードでドット演算子を使用すると、コンパイル時エラーが発生します。次の例は、PrivateExample クラスがダイナミッククラスとして宣言されている点を除き、前の例と同じです。

```
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
```

通常、ダイナミッククラスは、クラス外部のコードがプライベートプロパティにアクセスしようとすると、エラーを生成するのではなく、戻り値として undefined を返します。次の表は、ドット演算子を使用して strict モードでプライベートプロパティにアクセスしようとしたときにのみエラーが生成されることを示します。

	strict モード	**standard モード**
ドット演算子（.）	コンパイル時エラー	未定義（undefined）
角括弧演算子（[]）	未定義（undefined）	未定義（undefined）

ActionScript 3.0 で新しく導入された protected 属性は、クラス内またはサブクラス内の呼び出し元に対してプロパティを表示します。つまり、protected プロパティは、そのクラス内または継承階層内でそのクラスの下の任意の場所にあるクラスで使用することができます。これは、サブクラスが同じパッケージ内にあるか別のパッケージ内にあるかに関係なく、同じです。

ActionScript 2.0 に詳しいユーザーにとっては、この機能は ActionScript 2.0 の private 属性とほぼ同じです。また、ActionScript 3.0 の protected 属性は Java の protected 属性とほぼ同じですが、Java のこの属性は同じパッケージ内の呼び出し元へのアクセスも許可するという点で異なります。ただし、Java では、この属性は同じパッケージ内の呼び出し元へのアクセスも許可するという点で異なります。protected 属性は、サブクラスで必要なメソッドまたは変数を継承チェーン外部のコードに対して非表示にするときに便利です。

ActionScript 3.0 で新しく導入された internal 属性は、パッケージ内にある呼び出し元に対してプロパティを表示します。これはパッケージ内のコードのデフォルト属性であり、次のいずれの属性も持たないあらゆるプロパティに適用されます。

- public
- private
- protected
- ユーザー定義の名前空間

internal 属性は、Java のデフォルトのアクセス制御に似ています。ただし、Java ではこのレベルのアクセスに明示的な名前はなく、他のアクセス修飾子が省略された場合にのみ、このデフォルトのアクセス制御が使用されます。internal 属性は ActionScript 3.0 で使用可能で、パッケージ内の呼び出し元に対してのみプロパティを表示する意図を明示的に表すオプションがあります。

static 属性

static 属性は、var、const、または function キーワードで宣言されたプロパティと共に使用でき、クラスのインスタンスではなくクラスにプロパティを関連付けることができます。クラス外にあるコードは、インスタンス名ではなくクラス名を使用して静的プロパティを呼び出す必要があります。

静的プロパティはサブクラスに継承されませんが、サブクラスのスコープチェーンの一部です。つまり、サブクラスの本体内では、静的変数またはメソッドは、定義されたクラスを参照しなくても使用できます。

ユーザー定義の名前空間属性

事前に定義されているアクセス制御属性の代わりに、属性として使用するカスタム名前空間を作成できます。1 つの定義に使用できる名前空間属性は 1 つだけです。また、名前空間属性をアクセス制御属性（public、private、protected、internal）と組み合わせて使用することはできません。

変数

変数は、var キーワードか const キーワードで宣言できます。var キーワードで宣言した変数は、スクリプトの実行中に複数回その値を変更できます。const キーワードで宣言した変数は定数と呼ばれ、1 回だけ値を割り当てることができます。初期化された定数に新しい値を割り当てようとすると、エラーが発生します。

静的変数

静的変数は、static キーワードと、var または const ステートメントのいずれかの組み合わせを使用して宣言します。クラスのインスタンスではなく、クラスに関連付けられた静的変数は、オブジェクトのクラス全体に適用される情報の格納および共有に役立ちます。例えば、静的変数は、クラスがインスタンス化された回数を記録する場合や可能なクラスインスタンスの最大数を格納しておく場合に適しています。

次の例では、クラスのインスタンス化の回数を追跡する totalCount 変数およびインスタンス化の最大回数を格納する MAX_NUM 定数を作成します。totalCount と MAX_NUM 変数には、特定のインスタンスではなくクラス全体に適用される値が含まれているので、この 2 つの変数は静的です。

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

StaticVars クラスとそのサブクラスの外部にあるコードは、StaticVars クラス自体を通じた場合にのみ、totalCount プロパティと MAX_NUM プロパティを参照できます。例えば、次のコードは正常に動作します。

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

クラスのインスタンスから静的変数にアクセスできないため、次のコードはエラーを返します。

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

static および const の 2 つのキーワードで宣言された変数は、StaticVars クラスが MAX_NUM に対する場合と同様に、定数の宣言時に初期化する必要があります。コンストラクターまたはインスタンスメソッド内の MAX_NUM に値を割り当てることはできません。次のコードは静的定数を初期化する有効な方法ではないため、エラーが生成されます。

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

インスタンス変数

インスタンス変数には、var および const キーワードを使用し、static キーワードを使用せずに宣言されたプロパティが含まれます。クラス全体ではなくクラスインスタンスに関連付けられたインスタンス変数は、インスタンス固有の値の格納に便利です。例えば、Array クラスには、この Array クラスの特定のインスタンスが保持する配列エレメントの数を格納する length という名前のインスタンスプロパティがあります。

インスタンス変数は、var または const として宣言されたかどうかにかかわらず、サブクラスではオーバーライドできません。ただし、getter および setter メソッドをオーバーライドすることにより、変数をオーバーライドするのに似た機能を実現できます。

メソッド

メソッドは、クラス定義に含まれる関数です。クラスのインスタンスが作成されると、メソッドはそのインスタンスにバインドされます。メソッドは、クラス外で宣言された関数とは異なり、その関連付けられているインスタンスと別個に使用することはできません。

メソッドは、function キーワードを使用して定義します。あらゆるクラスプロパティと同様に、どのクラスプロパティ属性も、private、protected、public、internal、static、などのメソッドや、カスタム名前空間に適用できます。次の function ステートメントを使用できます。

```
public function sampleFunction():String {}
```

または、関数式を割り当てる変数を次のように使用することもできます。

```
public var sampleFunction:Function = function () {}
```

多くの場合、関数式ではなく function ステートメントを使用するのは、次のような理由のためです。

- 関数ステートメントは、関数式より簡潔で読みやすくなります。
- 関数ステートメントでは、override および final キーワードを使用できます。
- function ステートメントでは、識別子（関数の名前）とメソッド本体内のコードがより強力に結合されます。変数の値は代入ステートメントで変更できるので、変数とその関数式の結合をいつでも切り離すことができます。var ではなく const で変数を宣言するとこの問題を回避できますが、コードが読みにくくなり、override キーワードと final キーワードが使用できなくなるので、最適な方法とは言えません。

関数式を使用する必要があるのは、関数をプロトタイプオブジェクトに関連付ける場合などです。

コンストラクターメソッド

単にコンストラクターと呼ばれることもあるコンストラクターメソッドは、定義されたクラスと同じ名前を共有する関数です。コンストラクターメソッドに含まれるコードは、クラスのインスタンスが new キーワードで作成されるときに実行されます。例えば、次のコードは、status という名前のプロパティを 1 つ含む単純なクラス Example を定義します。status 変数の初期値は、コンストラクター関数内で設定します。

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

コンストラクターメソッドにはパブリックだけを指定できますが、public 属性を使用するかどうかは任意です。コンストラクターに、private、protected、internal やその他のアクセス制御指定子を使用することはできません。また、ユーザー定義の名前空間をコンストラクターメソッドで使用することもできません。

コンストラクターは、super() ステートメントを使用して直接のスーパークラスのコンストラクターを明示的に呼び出すことができます。スーパークラスのコンストラクターを明示的に呼び出さない場合は、コンパイラーによってコンストラクター本体の最初のステートメントの前に呼び出しが自動的に挿入されます。スーパークラスへの参照として super 接頭辞を使用し、スーパークラスのメソッドを呼び出すこともできます。同じコンストラクター本体で super() と super を使用する場合は、必ず最初に super() を呼び出します。そうしないと、super 参照が意図したとおりに動作しません。super() コンストラクターも、throw または return ステートメントの前に呼び出す必要があります。

次に、super() コンストラクターを呼び出す前に super 参照を使用しようとした場合の例を示します。新しいクラス ExampleEx は Example クラスを拡張します。ExampleEx コンストラクターは、super() を呼び出す前に、スーパークラスで定義された status 変数にアクセスしようとします。ExampleEx コンストラクター内の trace() ステートメントは、値 null を生成します。これは、super() コンストラクターが実行されるまで status 変数を使用できないからです。

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

コンストラクター内で return ステートメントを使用することはできますが、値を返すことはできません。つまり、return ステートメントに式または値を関連付けることはできません。したがって、コンストラクターメソッドは値を返すことができず、戻り値の型を指定できません。

クラスでコンストラクターメソッドを定義しなかった場合、コンパイラーが自動的に空のコンストラクターを作成します。クラスが別のクラスを拡張する場合、コンパイラーには、そのコンパイラーが生成するコンストラクター内の super() 呼び出しが含まれます。

静的メソッド

静的メソッドは、クラスメソッドとも呼ばれ、static キーワードで宣言されたメソッドです。静的メソッドは、クラスのインスタンスではなくクラスに関連付けられ、個々のインスタンスの状態以外のものに影響を与える機能のカプセル化に役立ちます。静的メソッドはクラス全体に関連付けられるため、クラスのインスタンスではなくクラスからのみアクセスできます。

静的メソッドは、クラスインスタンスの状態に影響を与えるだけではない機能のカプセル化に役立ちます。つまり、メソッドにクラスインスタンスの値に直接影響しない機能がある場合、そのメソッドは静的です。例えば、Date クラスには、ストリングを取得して数値に変換する parse() という静的メソッドがあります。このメソッドは、クラスの個々のインスタンスに影響を与えないため静的です。parse() メソッドは、日付値を表すストリングを取得し、そのストリングを解析して、Date オブジェクトの内部表現と互換性がある形式で数値を返します。このメソッドは、Date クラスのインスタンスに適用しても意味がないため、インスタンスメソッドではありません。

静的な parse() メソッドと、getMonth() などの Date クラスのインスタンスメソッドを比較してみます。getMonth() メソッドは、Date インスタンスの特定のコンポーネント月を取得することで、インスタンスの値に対して直接実行されるため、インスタンスメソッドです。

静的メソッドは個々のインスタンスにバインドされていないので、静的メソッドの本体内で this または super キーワードを使用できません。this 参照と super 参照は、インスタンスメソッドのコンテキスト内でのみ有効です。

他のクラスベースのプログラミング言語とは異なり、ActionScript 3.0 では静的メソッドは継承されません。

インスタンスメソッド

インスタンスメソッドは、static キーワードを使用せずに宣言されたメソッドです。インスタンスメソッドは、クラス全体ではなくクラスのインスタンスに関連付けられ、クラスの個々のインスタンスに影響を与える機能の実装に役立ちます。例えば、Array クラスには、Array インスタンスに直接実行される sort() という名前のインスタンスメソッドが含まれています。

インスタンスメソッドの本体内では、静的変数およびインスタンス変数はスコープ内にあります。つまり、同じクラス内で定義された変数は、単純な識別子を使用して参照できます。例えば、次の CustomArray クラスは Array クラスを拡張します。CustomArray クラスは、クラスインスタンスの総数を追跡する静的変数 arrayCountTotal、インスタンスが作成された順序を追跡するインスタンス変数 arrayNumber、およびこれらの変数の値を返すインスタンスメソッド getPosition() を定義します。

```
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);
    }
}
```

クラス外部にあるコードは、CustomArray.arrayCountTotal を使用して、クラスオブジェクトから arrayCountTotal 静的変数にアクセスする必要がありますが、getPosition() メソッドの本体内にあるコードは、arrayCountTotal 静的変数を直接参照できます。これは、スーパークラスの静的変数でも同じです。ActionScript 3.0 では静的プロパティは継承されませんが、スーパークラスの静的プロパティはスコープ内です。例えば、Array クラスにはいくつかの静的変数があり、そのうちの 1 つは DESCENDING という定数です。Array サブクラス内にあるコードは、単純な識別子を使用して静的定数 DESCENDING にアクセスできます。

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

インスタンスメソッドの本体内の this 参照の値は、メソッドが関連付けられているインスタンスへの参照です。次のコードは、this 参照がメソッドを含むインスタンスを参照していることを示します。

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

インスタンスメソッドの継承は、override キーワードと final キーワードを使用して制御できます。override 属性を使用して、継承されたメソッドを再定義できます。また、final 属性を使用して、サブクラスによってメソッドがオーバーライドされないようにできます。

get および set アクセッサーメソッド

getter および **setter** とも呼ばれる get および set アクセッサー関数を使用すると、作成したクラスの使いやすいプログラミングインターフェイスを提供すると同時に、情報の非表示およびカプセル化というプログラミング原則に従うことができます。get および set 関数を使用して、クラスプロパティをクラスに対してプライベートに保持できますが、クラスのユーザーは、クラスメソッドを呼び出すのではなくクラス変数にアクセスしている場合と同じように、これらのプロパティにアクセスできます。

この方法の利点は、getPropertyName()、setPropertyName() など、従来の冗長な名前のアクセッサー関数を使用する必要がないことです。また、getter および setter には、読み取りおよび書き込みアクセスを可能にする各プロパティに対して 2 つの公開関数を持つ必要がないという利点もあります。

次の例の GetSet クラスには、privateProperty という名前のプライベート変数へのアクセスを提供する publicAccess() という名前の get および set アクセッサー関数が含まれています。

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

privateProperty プロパティに直接アクセスしようとすると、次のエラーが発生します。

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

代わりに、GetSet クラスを使用する場合は、publicAccess というプロパティのように見えて、実際には privateProperty という名前のプライベートプロパティに対して動作する get および set アクセッサー関数のペアを使用します。次の例では、GetSet クラスをインスタンス化し、publicAccess という名前のパブリックアクセッサーを使用して privateProperty の値を設定します。

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

getter および setter 関数でも、スーパークラスから継承されるプロパティをオーバーライドできますが、通常のクラスメンバー変数を使用したときにはオーバーライドできません。var キーワードで宣言されたクラスメンバー変数は、サブクラスではオーバーライドできません。一方、getter および setter 関数を使用して作成されたプロパティにはこの制限はありません。スーパークラスから継承した getter 関数と setter 関数に override 属性を使用できます。

バインドメソッド

バインドメソッドは、メソッドクロージャとも呼ばれ、単にインスタンスから抽出されるメソッドです。バインドメソッドの例には、関数にパラメーターとして渡され、関数から値として返されるメソッドがあります。ActionScript 3.0 で新しく導入されたバインドメソッドは、インスタンスから抽出されたときでもレキシカル環境が保持されるメソッドクロージャに似ています。バインドメソッドとメソッドクロージャの主な違いは、バインドメソッドの this 参照は、バインドメソッドを実装するインスタンスにリンクされたまま、つまりバインドされたままであるという点です。つまり、バインドメソッドの this 参照が、常にこのメソッドを実装する元のオブジェクトを指していることを意味します。関数クロージャの場合、this 参照は汎用です。つまり、この関数が呼び出された時点で関連付けられているオブジェクトが何であっても、そのオブジェクトを参照します。

this キーワードを使用する場合は、バインドメソッドを理解していることが重要です。this キーワードを使用すれば、メソッドの親オブジェクトを参照できます。ほとんどの ActionScript プログラマーは、this キーワードが常にメソッドの定義を含むオブジェクトまたはクラスを表すと考えますが、メソッドのバインディングがないと必ずしもそうなりません。旧バージョンの ActionScript では、例えば、this 参照はメソッドを実装するインスタンスを常に参照するわけではありませんでした。ActionScript 2.0 でインスタンスからメソッドを抽出すると、this 参照が元のインスタンスにバインドされないだけでなく、インスタンスのクラスのメンバー変数もメソッドも使用できません。ActionScript 3.0 では、メソッドをパラメーターとして渡すとバインドメソッドが自動的に作成されるため、これは問題にはなりません。バインドメソッドにより、this キーワードは常に、メソッドが定義されたオブジェクトまたはクラスを参照します。

次のコードは、ThisTest というクラスを定義します。このクラスには、バインドメソッドを定義するメソッド foo()、およびバインドメソッドを返すメソッド bar() が含まれます。クラス外部にあるコードは、ThisTest クラスのインスタンスを作成し、bar() メソッドを呼び出して、myFunc という変数に戻り値を格納します。

```
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 */
```

コードの最後の 2 行は、バインドメソッド foo() の this 参照が、その直前の行の this 参照がグローバルオブジェクトを指していても、依然として ThisTest クラスのインスタンスを指すことを示しています。さらに、myFunc 変数に格納されているバインドメソッドは、引き続き ThisTest クラスのメンバー変数にアクセスできます。この同じコードを ActionScript 2.0 で実行すると、this 参照が一致し、num 変数は undefined になります。

addEventListener() メソッドでは関数またはメソッドをパラメーターとして渡す必要があるため、バインドメソッドの追加が最もわかりやすいのはイベントハンドラーを使用する場合です。

クラスによる列挙

列挙は、値の小さなセットをカプセル化するために作成するカスタムデータ型です。ActionScript 3.0 は、C++ の enum キーワードや Java の Enumeration インターフェイスとは異なり、特定の列挙機能をサポートしません。ただし、クラスと静的定数を使用して列挙を作成できます。例えば、次のコードに示すように、ActionScript 3.0 の PrintJob クラスは PrintJobOrientation という列挙を使用して、"landscape" および "portrait" という値を格納します。

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

慣例では、列挙クラスはクラスを拡張する必要がないので、final 属性で宣言されます。このクラスは静的メンバーのみで構成されます。つまり、クラスのインスタンスを作成しません。代わりに、次の抜粋されたコードに示すように、クラスオブジェクトを通じて直接列挙値にアクセスします。

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

ActionScript 3.0 内のすべての列挙クラスには、String、int、または uint のいずれかの型の変数のみが含まれます。リテラルのストリング値または数値ではなく列挙を使用する利点は、表記ミスを見つけやすいことです。列挙の名前を間違って入力した場合、ActionScript コンパイラーはエラーを生成します。リテラル値を使用した場合、単語を間違って入力するか数値を間違って使用してもコンパイラーに認識されません。前の例では、次のコードの抜粋に示すように、列挙定数の名前が間違っている場合にコンパイラーはエラーを生成します。

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

しかし、次のようにリテラルストリング値にスペルミスがある場合、コンパイラーはエラーを生成しません。

```
    if (pj.orientation == "portrai") // no compiler error
```

列挙を作成するもう 1 つの方法でも、列挙の静的プロパティで別のクラスを作成することが必要になります。ただし、この方法は、静的プロパティにストリング値または整数値ではなくクラスのインスタンスが含まれる点で異なります。例えば、次のコードは、曜日に対する列挙クラスを作成します。

```
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();
}
```

ActionScript 3.0 ではこの方法は使用されていませんが、この方法により型チェックの向上を希望する多くの開発者によって採用されています。例えば、列挙値を返すメソッドは、その戻り値を列挙データ型に制限することができます。次のコードは、曜日を返す関数だけでなく、列挙型を型注釈として使用する関数呼び出しも示します。

```
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();
```

Day クラスを拡張することにより、整数を各曜日に関連付け、曜日のストリング表現を返す toString() メソッドを提供することもできます。

埋め込みアセットクラス

ActionScript 3.0 では、埋め込みアセットを表現するために、埋め込みアセットクラスと呼ばれる特別なクラスを使用します。埋め込みアセットは、コンパイル時に SWF ファイルに含まれるサウンド、イメージ、フォントなどのアセットです。アセットは、動的にロードするのではなく埋め込むことで実行時に確実に利用できるようになりますが、この方法には SWF ファイルのサイズが大きくなるという短所があります。

Flash Professional での埋め込みアセットクラスの使用

アセットを埋め込むには、まずアセットを FLA ファイルのライブラリに保存します。次に、アセットのリンケージプロパティを使用して、アセットの埋め込みアセットクラスの名前を指定します。その名前のクラスがクラスパスに見つからない場合、クラスは自動的に生成されます。これにより、埋め込みアセットクラスのインスタンスを作成し、そのクラスによって定義または継承された任意のプロパティとメソッドを使用できるようになります。例えば、次のコードを使用して、PianoMusic という名前の埋め込みアセットクラスにリンクされた埋め込みサウンドを再生できます。

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

または、次で説明するとおり、[Embed] メタデータタグを使用して Flash Professional プロジェクトにアセットを埋め込むこともできます。コード内に [Embed] メタデータタグがあると、プロジェクトのコンパイル時に、Flash Professional コンパイラーではなく Flex コンパイラーが使用されます。

Flex コンパイラーを使用した埋め込みアセットクラスの使用

Flex コンパイラーを使用してコードをコンパイルする場合、アセットを ActionScript コードに埋め込むには、[Embed] メタデータタグを使用します。メインソースフォルダーか、プロジェクトのビルドパスにある別のフォルダーにアセットを配置します。Flex コンパイラーによって Embed メタデータタグが検出されると、埋め込みアセットクラスが作成されます。このクラスには、[Embed] メタデータタグの直後に宣言するデータ型クラスの変数からアクセスします。例えば、次のコードでは sound1.mp3and というサウンドを埋め込み、soundCls という変数を使用して、そのサウンドに関連付けられている埋め込みアセットクラスに参照を保存します。次に、埋め込みアセットクラスのインスタンスを作成して、そのインスタンスの play() メソッドを呼び出します。

```
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

Flash Builder ActionScript プロジェクトで [Embed] メタデータタグを使用するには、Flex フレームワークから必要なクラスをインポートする必要があります。例えば、サウンドを埋め込むには、mx.core.SoundAsset クラスのインポートが必要です。Flex フレームワークを使用するには、ActionScript ビルドパスに framework.swc ファイルを含めます。これによって SWF ファイルのサイズが増大します。

Adobe Flex

または、Flex 上で MXML タグ定義の @Embed() ディレクティブを使用してアセットを埋め込むこともできます。

インターフェイス

インターフェイスは、関連しないオブジェクトの相互の通信を可能にするメソッド宣言の集合です。例えば、ActionScript 3.0 では、クラスでイベントオブジェクトを処理するために使用できるメソッド宣言を含む IEventDispatcher インターフェイスを定義します。IEventDispatcher インターフェイスは、オブジェクトが相互にイベントオブジェクトを渡し合うための標準的な手段を確立します。次のコードは、IEventDispatcher インターフェイスの定義を示します。

```
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;
}
```

インターフェイスは、メソッドのインターフェイスとその実装との違いに基づいています。メソッドのインターフェイスには、メソッドの名前、すべてのパラメーター、戻り値の型など、そのメソッドを呼び出すために必要なすべての情報が含まれます。メソッドの実装には、インターフェイス情報だけではなく、メソッドのビヘイビアーを実行する実行可能ステートメントも含まれます。インターフェイス定義には、メソッドインターフェイスのみが含まれ、インターフェイスを実装するクラスはメソッドの実装を定義します。

ActionScript 3.0 では、EventDispatcher クラスは、IEventDispatcher インターフェイスのメソッドすべてを定義し、各メソッドにメソッドの本体を追加して、IEventDispatcher インターフェイスを実装します。次のコードは、EventDispatcher クラスの定義からの抜粋です。

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

IEventDispatcher インターフェイスは、プロトコルとして機能します。EventDispatcher インスタンスは、このプロトコルを使用してイベントオブジェクトを処理し、IEventDispatcher インターフェイスを実装する他のオブジェクトに渡します。

インターフェイスは、クラスのようにデータ型を定義すると説明することもできます。したがって、インターフェイスはクラスのように型注釈として使用できます。インターフェイスは、データ型として is 演算子や as 演算子などのデータ型を必要とする演算子と使用することもできます。しかし、クラスとは異なり、インターフェイスをインスタンス化することはできません。この違いから、多くのプログラマーは、インターフェイスを抽象データ型、クラスを具象データ型と見なしています。

インターフェイスの定義

インターフェイス定義の構造は、クラス定義の構造に似ていますが、インターフェイスにはメソッド本体のないメソッドしか含めることができません。インターフェイスに変数や定数を含めることはできませんが、getter および setter は含めることができます。インターフェイスを定義するには、interface キーワードを使用します。例えば IExternalizable は、ActionScript 3.0 の flash.utils パッケージの一部です。IExternalizable インターフェイスは、オブジェクトを直列化するためのプロトコルを定義します。つまり、デバイスへの格納またはネットワーク間の転送に適した形式にオブジェクトを変換します。

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

IExternalizable インターフェイスは、public アクセス制御修飾子で宣言されます。インターフェイス定義は、public および internal アクセス制御指定子でのみ変更できます。インターフェイス定義内のメソッド宣言には、アクセス制御指定子を使用できません。

ActionScript 3.0 では、インターフェイス名は大文字の I で始まるという表記規則に従いますが、インターフェイス名には有効な任意の識別子を使用できます。インターフェイス定義は、通常パッケージの最上位に配置されます。クラス定義内または別のインターフェイス定義内に、インターフェイス定義を配置することはできません。

インターフェイスは、他のインターフェイスを拡張できます。例えば、次のインターフェイス、IExample は、IExternalizable インターフェイスを拡張します。

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

IExample インターフェイスを実装するクラスはいずれも、extra() メソッドの実装だけでなく、IExternalizable インターフェイスから継承される writeExternal() および readExternal() メソッドの実装も含む必要があります。

クラス内でのインターフェイスの実装

クラスは、インターフェイスを実装できる唯一の ActionScript 3.0 言語エレメントです。クラス宣言内で implements キーワードを使用して、1 つまたは複数のインターフェイスを実装します。次の例では、IAlpha および IBeta の 2 つのインターフェイスと、これらの両方を実装する Alpha クラスを定義します。

```
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 {}
}
```

インターフェイスを実装するクラスでは、実装されたメソッドは以下を行う必要があります。

- public アクセス制御識別子を使用する。
- インターフェイスメソッドと同じ名前を使用する。
- 同じ数のパラメーターを含む。各パラメーターのデータ型は、インターフェイスメソッドパラメーターのデータ型と一致する必要があります。
- 同じ戻り値の型を使用する。

```
public function foo(param:String):String {}
```

実装するメソッドのパラメーターには、ある程度自由に名前を付けることができます。実装されるメソッドとインターフェイスメソッドのパラメーター数および各パラメーターのデータ型は一致する必要がありますが、パラメーター名を一致させる必要はありません。例えば、前の例では Alpha.foo() メソッドのパラメーターの名前は param ですが、

IAlpha.foo() インターフェイスメソッドでのパラメーターの名前は str です。

```
function foo(str:String):String;
```

デフォルトのパラメーター値には若干の柔軟性があります。インターフェイス定義は、デフォルトのパラメーター値を備えた関数宣言を含むことができます。このような関数宣言を実装するメソッドは、インターフェイス定義に指定されている値と同じデータ型のメンバーであるデフォルトのパラメーター値を持つ必要がありますが、実際の値が一致する必要はありません。例えば、次のコードは、デフォルトのパラメーター値 3 を持つメソッドを含むインターフェイスを定義します。

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

次のクラス定義は、Igamma インターフェイスを実装しますが、異なるデフォルトパラメーター値を使用します。

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

この柔軟性は、インターフェイスを実装する規則がデータ型の互換性を確保するように特別に設計されているからです。このために、同じパラメーター名およびデフォルトのパラメーター値を要求する必要はありません。

継承

継承は、コード再利用の一形式です。プログラマーは継承を使用して、既存のクラスに基づく新しいクラスを開発することができます。既存のクラスは、基本クラスまたはスーパークラスと呼ばれますが、新しいクラスは、サブクラスと呼ばれます。継承の主な利点は、既存のコードをそのまま維持しながら、基本クラスのコードを再利用できることです。さらに、継承では他のクラスが基本クラスとやり取りする方法を変更する必要がありません。入念にテストされたか、既に使用されている既存クラスを変更するのではなく、継承を使用することでそのクラスを追加プロパティまたはメソッドで拡張できる統合モジュールとして使用できます。したがって、extends キーワードを使用して、クラスが別のクラスを継承することを示すことができます。

継承を使用すると、コード内でポリモーフィズムを利用することもできます。ポリモーフィズムを使用すると、異なるデータ型に適用された場合に異なる動作を実行するメソッドに単一のメソッド名を使用できます。単純な例として、2 つのサブクラス Circle と Square を持つ Shape という名前の基本クラスがあります。Shape クラスは、形状の面積を返す area() というメソッドを定義します。ポリモーフィズムが実装されている場合、Circle 型および Square 型のオブジェクトで area() メソッドを呼び出して、正しい計算を行うことができます。継承では、サブクラスが基本クラスからメソッドを継承し、再定義できるようにする、つまりオーバーライドを許可することで、ポリモーフィズムが有効になります。次の例では、area() メソッドを、Circle クラスと Square クラスによって再定義します。

```
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
```

各クラスはデータ型を定義するため、継承を使用すると、基本クラスと基本クラスを拡張するクラスの特別な関係が作成されます。サブクラスは、基本クラスのすべてのプロパティを保有します。これは、サブクラスのインスタンスは常に基本クラスのインスタンスの代わりに使用できることを意味します。例えば、メソッドが Shape 型のパラメーターを定義する場合、Circle 型は Shape 型を拡張するので Circle 型のパラメーターを渡すことができます。次に例を示します。

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

インスタンスプロパティと継承

インスタンスプロパティは、function、var、または const キーワードのいずれで定義されているかにかかわらず、プロパティが基本クラスの private 属性で宣言されていない限り、すべてのサブクラスに継承されます。例えば、ActionScript 3.0 の Event クラスには、すべてのイベントオブジェクトに共通するプロパティを継承する多数のサブクラスがあります。

一部の型のイベントでは、Event クラスにイベントを定義するために必要なすべてのプロパティが含まれます。これらの型のイベントでは、Event クラスで定義されたもの以外のインスタンスプロパティは必要ありません。このようなイベントの例として、データが正常にロードされたときに発生する complete イベント、ネットワーク接続が確立されたときに発生する connect イベントなどがあります。

次の例は、サブクラスに継承されるプロパティおよびメソッドの一部を示す Event クラスからの抜粋です。プロパティが継承されるので、あらゆるサブクラスのインスタンスは、これらのプロパティにアクセスできます。

```
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 {}
    ...
}
```

その他の型のイベントは、Event クラスで使用できない固有のプロパティを必要とします。これらのイベントは、Event クラスで定義されたプロパティに新しいプロパティを追加できるように、Event クラスのサブクラスを使用して定義されます。このようなサブクラスには、mouseMove イベント、click イベントなど、マウス操作またはマウスクリックに関連付けられているイベントに固有のプロパティを追加する MouseEvent クラスなどがあります。次の例は、MouseEvent クラスからの抜粋であり、サブクラスには存在するが基本クラスには存在しないプロパティの定義を示しています。

```
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 {}
    ...
}
```

アクセス制御指定子と継承

プロパティが public キーワードで宣言された場合、このプロパティは任意の場所のコードから参照できます。つまり、public キーワードは、private、protected、および internal キーワードとは異なり、プロパティの継承に制限を加えません。

プロパティが private キーワードで宣言された場合、このプロパティは定義されたクラス内でのみ参照でき、サブクラスに継承されません。この動作は、旧バージョンの ActionScript とは異なります。旧バージョンでは、private キーワードは ActionScript 3.0 の protected キーワードと同じように動作しました。

protected キーワードは、定義されたクラス内だけでなく、すべてのサブクラスからもプロパティを参照できることを示します。Java の protected キーワードとは異なり、ActionScript 3.0 の protected キーワードでは、同じパッケージ内の他のすべてのクラスからプロパティを参照できるわけではありません。ActionScript 3.0 では、サブクラスのみが protected キーワードで宣言されたプロパティにアクセスできます。また、protected 指定されたプロパティは、サブクラスが基本クラスと同じパッケージ内にあるか、または異なるパッケージ内にあるかにかかわらず、サブクラスから参照できます。

プロパティが定義されているパッケージからのプロパティへの参照を制限するには、internal キーワードを使用するか、またはアクセス制御指定子を一切使用しないようにします。internal アクセス制御指定子は、アクセス制御指定子が指定されていないときに適用されるデフォルトのアクセス制御指定子です。internal とマークされたプロパティは、同じパッケージ内に存在するサブクラスによってのみ継承されます。

次の例を使用して、各アクセス制御指定子がパッケージ境界を越えてどのように継承に影響するかを確認できます。次のコードは、AccessControl というメインアプリケーションクラスと Base および Extender というその他のクラス 2 つを定義します。Base クラスは foo パッケージ内にあり、Base クラスのサブクラスである Extender クラスは bar パッケージ内にあります。AccessControl クラスは Extender クラスのみを読み込み、Base クラスで定義されている str という変数にアクセスしようとする Extender のインスタンスを作成します。str 変数は public として宣言され、この結果、次の抜粋に示すようにコードがコンパイルされて実行されます。

```
// 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
        }
    }
}
```

その他のアクセス制御指定子が前の例のコンパイルおよび実行に与える影響を確認するには、AccessControl クラスから次の行を削除またはコメントアウトした後、str 変数のアクセス制御指定子を private、protected、または internal に変更します。

```
trace(myExt.str);// error if str is not public
```

変数のオーバーライド禁止

var または const キーワードで宣言されたプロパティは継承されますが、オーバーライドすることはできません。プロパティをオーバーライドすると、サブクラスでプロパティが再定義されます。オーバーライドできるプロパティの型は、get アクセサーと set アクセサー（function キーワードで宣言されるプロパティ）のみです。インスタンス変数をオーバーライドすることはできませんが、インスタンス変数の getter および setter メソッドを作成し、これらのメソッドをオーバーライドすることで、同じ機能を実現できます。

メソッドのオーバーライド

メソッドをオーバーライドすると、継承されたメソッドの動作が再定義されます。静的メソッドは継承されず、オーバーライドすることはできません。ただし、次の 2 つの条件が満たされていれば、インスタンスメソッドはサブクラスに継承され、オーバーライドできます。

- インスタンスメソッドが基本クラス内で final キーワードで宣言されていません。インスタンスメソッドで使用された場合、final キーワードは、サブクラスによってメソッドがオーバーライドされないように、プログラマーが意図的に指定したことを示します。

- インスタンスメソッドが基本クラス内で private アクセス制御指定子で宣言されていません。基本クラス内でメソッドが private としてマークされていると、基本クラスのメソッドはサブクラスから参照できないので、サブクラス内で同じ名前のメソッドを定義するときに override キーワードを使用する必要はありません。

  上記の条件を満たすインスタンスメソッドをオーバーライドするには、サブクラス内のメソッドの定義は override キーワードを使用し、次の方法でメソッドのスーパークラスバージョンと一致する必要があります。

- オーバーライドメソッドには、基本クラスのメソッドと同じレベルのアクセス制御が必要です。internal とマークされたメソッドには、アクセス制御指定子を持たないメソッドと同じレベルのアクセス制御が必要です。
- オーバーライドメソッドには、基本クラスのメソッドと同じ数のパラメーターが必要です。
- オーバーライドメソッドのパラメーターには、基本クラスのメソッドのパラメーターと同じデータ型注釈が必要です。
- オーバーライドメソッドには、基本クラスのメソッドと同じ戻り値の型が必要です。

ただし、オーバーライドメソッドのパラメーターの名前は、パラメーターの数および各パラメーターのデータ型が一致していれば、基本クラスのパラメーターの名前と一致する必要はありません。

super ステートメント

メソッドをオーバーライドするとき、多くのプログラマーが、動作を完全に置き換えるのではなく、オーバーライドするスーパークラスメソッドの動作に追加したいと考えるでしょう。これには、サブクラス内のメソッドがそれ自体のスーパークラスバージョンを呼び出すメカニズムが必要です。super ステートメントは、直接のスーパークラスへの参照が含まれるそのようなメカニズムを提供します。次の例では、thanks() というメソッドを含む Base クラスと、thanks() メソッドをオーバーライドする Extender という Base クラスのサブクラスを定義します。Extender.thanks() メソッドは、super ステートメントを使用して Base.thanks() を呼び出します。

```
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";
    }
}
```

getter と setter のオーバーライド

スーパークラスで定義された変数をオーバーライドすることはできませんが、getter と setter をオーバーライドすることができます。例えば、次のコードでは、ActionScript 3.0 の MovieClip クラスで定義された currentLabel という名前の getter をオーバーライドします。

```
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;
        }
    }
}
```

OverrideExample クラスコンストラクターの trace() ステートメントの出力は、Override: null です。これは、継承された currentLabel プロパティをオーバーライドできたことを示します。

継承されない静的プロパティ

静的プロパティはサブクラスに継承されません。つまり、サブクラスのインスタンスから静的プロパティにアクセスすることはできません。静的プロパティは、そのプロパティが定義されたクラスオブジェクトからのみアクセス可能です。例えば、次のコードは、Base という基本クラスと、Extender という Base クラスを拡張するサブクラスを定義します。静的変数 test は Base クラスで定義されています。次の抜粋で記述されているコードは、strict モードではコンパイルされず、standard モードではランタイムエラーが生成されます。

```
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 { }
```

次のコードに示すように、静的変数 test には、クラスオブジェクトからアクセスする必要があります。

```
Base.test;
```

ただし、静的プロパティと同じ名前を使用してインスタンスプロパティを定義することができます。このインスタンスプロパティは、静的プロパティと同じクラス内またはサブクラス内で定義できます。例えば、前述の例の Base クラスでは、test というインスタンスプロパティを定義できます。次のコードはコンパイルおよび実行されます。これは、インスタンスプロパティが Extender クラスに継承されるからです。test インスタンス変数の定義が Extender クラスにコピーされるのではなく、移動された場合も、コードはコンパイルされ、実行されます。

```
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 {}
```

静的プロパティとスコープチェーン

静的プロパティは継承されませんが、そのプロパティが定義されたクラスおよびそのクラスのサブクラスのスコープチェーン内にあります。このため、静的プロパティは、定義されたクラスおよびサブクラスのスコープ内にあると言います。つまり、静的プロパティが定義されたクラスおよびそのサブクラスの本体内から直接静的プロパティにアクセスできます。

次の例では、前述の例で定義したクラスを変更して、Base クラスで定義された静的変数 test が Extender クラスのスコープ内にあることを示します。つまり、Extender クラスは、test を定義したクラスの名前を接頭辞として変数に付けなくても、静的変数 test にアクセスできます。

```
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
    }

}
```

インスタンスプロパティが、同じクラスまたはスーパークラス内で静的プロパティと同じ名前を使用するように定義されている場合は、インスタンスプロパティはスコープチェーン内で優先順位が高くなります。インスタンスプロパティは、静的プロパティをシャドウすると言います。つまり、静的プロパティの値ではなくインスタンスプロパティの値が使用されます。例えば、次のコードは、Extender クラスが test という名前のインスタンス変数を定義するときに、trace() ステートメントが静的変数の値ではなく、インスタンス変数の値を使用することを示しています。

```
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
    }

}
```

高度なテクニック

ActionScript OOP サポートの歴史

ActionScript 3.0 は旧バージョンの ActionScript をベースに構築されているため、ActionScript オブジェクトモデルがどのように進化してきたかを理解しておくと役立ちます。ActionScript は、Flash Professional の初期バージョン用の単純なスクリプトメカニズムとして誕生しました。その後、プログラマーは ActionScript を使用してより複雑なアプリケーションを作成するようになりました。プログラマーのニーズに応えるために、その後の各リリースには複雑なアプリケーションの作成を容易にする言語機能が追加されてきました。

ActionScript 1.0

ActionScript 1.0 は、Flash Player 6 以前で使用されていたプログラミング言語のバージョンです。この開発初期段階でも、ActionScript オブジェクトモデルは基本的なデータ型としてのオブジェクトの概念をベースにしていました。ActionScript オブジェクトは、プロパティのグループを持つ複合データ型です。オブジェクトモデルについて説明するとき、プロパティという用語には、変数、関数、メソッドなど、オブジェクトに関連付けられるあらゆるものが含まれます。

第 1 世代の ActionScript では、class キーワードによるクラスの定義はサポートされていませんが、プロトタイプオブジェクトと呼ばれる特殊なオブジェクトを使用してクラスを定義できます。class キーワードを使用して具象オブジェクトにインスタンス化する抽象クラスの定義を作成する、Java や C++ などのクラスベース言語の場合と異なり、ActionScript 1.0 の

ようなプロトタイプベース言語では、既存オブジェクトを他のオブジェクトのモデル（またはプロトタイプ）として使用します。クラスベース言語のオブジェクトはテンプレートになるクラスを示す場合がありますが、プロトタイプベース言語のオブジェクトはテンプレートになる別のオブジェクト、つまりプロトタイプを示します。

ActionScript 1.0 でクラスを作成するには、クラスのコンストラクター関数を定義します。ActionScript の関数は、単に抽象的な定義ではなく実際のオブジェクトです。作成したコンストラクター関数は、そのクラスのインスタンスのプロトタイプ的なオブジェクトになります。次のコードでは、Shape という名前のクラスを作成し、デフォルトで true に設定される visible というプロパティを 1 つ定義します。

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

このコンストラクター関数は、new 演算子でインスタンス化できる Shape クラスを次のように定義します。

```
myShape = new Shape();
```

Shape() コンストラクター関数オブジェクトが Shape クラスのインスタンスのプロトタイプとして機能するのと同様に、これは Shape のサブクラスのプロトタイプ、すなわち Shape クラスを拡張する他のクラスとしても機能します。

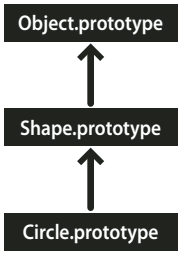
Shape クラスのサブクラスであるクラスを作成するには、次の 2 つの手順を実行します。最初に、次のようにクラスのコンストラクター関数を定義して、クラスを作成します。

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

次に、new 演算子を使用して、Shape クラスが Circle クラスのプロトタイプであると宣言します。デフォルトでは、作成したクラスはそのプロトタイプとして Object クラスを使用します。つまり、現時点では、Circle.prototype には汎用オブジェクト（Object クラスのインスタンス）が含まれています。Circle のプロトタイプに Object ではなく Shape を指定するには、汎用オブジェクトではなく Shape オブジェクトが含まれるように、次のコードを使用して Circle.prototype の値を変更します。

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

これで、Shape クラスと Circle クラスは、プロトタイプチェーンと呼ばれる継承関係内で相互にリンクされました。次の図は、プロトタイプチェーン内の関係を示しています。



各プロトタイプチェーンの最後にある基本クラスは Object クラスです。Object クラスには、ActionScript 1.0 で作成されたすべてのオブジェクトの基本プロトタイプオブジェクトを参照する Object.prototype という静的プロパティが含まれています。このプロトタイプチェーン例の次のオブジェクトは Shape オブジェクトです。これは、Shape.prototype プロパティは明示的に設定されていないので、依然として汎用オブジェクト（Object クラスのインスタンス）を保持しているからです。このプロトタイプチェーンの最後のリンクは Circle クラスで、それ自体のプロトタイプである Shape クラスにリンクされています。Circle.prototype プロパティは Shape オブジェクトを保持します。

次の例に示すように、Circle クラスのインスタンスを作成すると、インスタンスは Circle クラスのプロトタイプチェーンを継承します。

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

以前に、Shape クラスのメンバーとして、visible という名前のプロパティを作成しました。この例では、visible プロパティは、myCircle オブジェクトの一部としてではなく Shape オブジェクトのメンバーとしてのみ存在しますが、次のコード行では true が出力されます。

```
trace(myCircle.visible); // output: true
```

ランタイムでは、プロトタイプチェーン内を移動することにより、myCircle オブジェクトが visible プロパティを継承することを確認できます。このコードを実行すると、ランタイムは最初に myCircle オブジェクトのプロパティから visible というプロパティを検索しますが、このプロパティは見つかりません。次に Circle.prototype オブジェクトを検索しますが、やはり visible というプロパティは見つかりません。ランタイムは、プロトタイプチェーン内を引き続き検索し、最後に Shape.prototype オブジェクトで定義された visible プロパティを見つけ、そのプロパティの値を出力します。

簡潔にするために、ここでは、プロトタイプチェーンの詳細および複雑さに関する説明の多くを省略します。代わりに ActionScript 3.0 オブジェクトモデルの理解に役立つ情報を提供します。

ActionScript 2.0

ActionScript 2.0 では、Java や C++ などのクラスベース言語の経験がある方には使い慣れた方法でクラスを定義できる class、extends、public、private などの新しいキーワードが導入されました。基礎となる継承メカニズムは ActionScript 1.0 と ActionScript 2.0 で変わりがないことを理解する必要があります。ActionScript 2.0 では、クラスの定義に使用する新しいシンタックスが追加されたにすぎません。プロトタイプチェーンは、両方の言語バージョンで同じように機能します。

次の抜粋に示すように ActionScript 2.0 で導入された新しいシンタックスでは、より直観的な方法でクラスを定義できます。

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

ActionScript 2.0 では、コンパイル時の型チェックに使用するための型注釈も導入されました。これにより、前述の例の visible プロパティにはブール値だけが含まれることを宣言できます。また、新しい extends キーワードにより、サブクラスを作成するプロセスが簡略化されます。次の例では、ActionScript 1.0 では 2 つの手順が必要なプロセスが、extends キーワードにより 1 つの手順で実行されています。

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

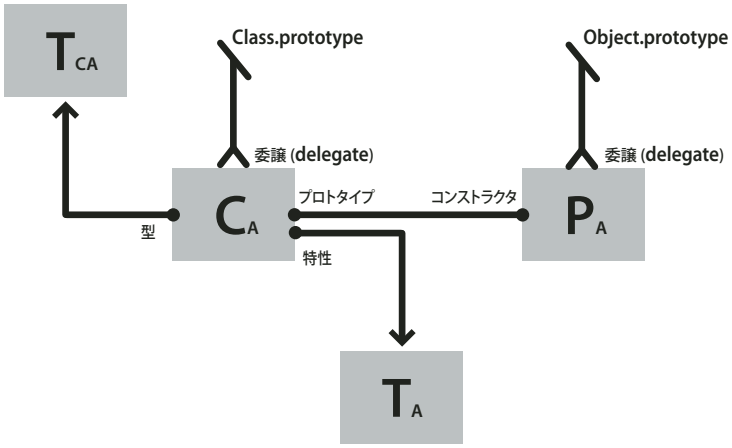
コンストラクターは、クラス定義の一部として宣言されるようになりました。また、クラスプロパティ id および radius は明示的に宣言する必要があります。

ActionScript 2.0 ではインターフェイス定義のサポートも追加され、オブジェクト間の通信用に正式に定義されたプロトコルでオブジェクト指向プログラムをさらに改良できるようになりました。

ActionScript 3.0 のクラスオブジェクト

通常 Java や C++ に関連付けられる一般的なオブジェクト指向プログラミングパラダイムでは、クラスを使用してオブジェクトの型を定義します。このパラダイムを採り入れたプログラミング言語も、クラスにより定義されるデータ型のインスタンスを作成するためにクラスを使用する傾向にあります。ActionScript では、この両方の目的でクラスを使用しますが、プロトタイプベース言語であることから興味深い特徴が付け加えられています。ActionScript ではクラス定義ごとに、動作と状態の両方を共有できる特別なクラスオブジェクトを作成します。とは言え、コーディングする上でこの違いが実質的な影響があると感じる ActionScript プログラマーはほとんどいないでしょう。ActionScript 3.0 は、この特別なオブジェクトを使用しなくても、さらには理解していなくても、高度なオブジェクト指向 ActionScript アプリケーションを作成できるように設計されています。

次の図は、class A {} ステートメントで定義された A という名前の単純なクラスを表すクラスオブジェクトの構造を示しています。



図中の四角形はオブジェクトを表します。図中の各オブジェクトには、クラス A に属していることを表す添え字 A が付いています。クラスオブジェクト（CA）には、その他の重要なオブジェクトへの参照が多数含まれています。インスタンス特性オブジェクト（TA）は、クラス定義内で定義されたインスタンスプロパティを格納します。クラス特性オブジェクト（TCA）はクラスの内部型を表し、そのクラスによって定義された静的プロパティを格納します（添え字 C は「クラス」を表します）。プロパティオブジェクト（PA）は、常に constructor プロパティを通じて関連付けられた元のクラスオブジェクトを意味します。

特性オブジェクト

ActionScript 3.0 で新しく導入された特性オブジェクトは、パフォーマンスを考慮して実装されました。旧バージョンの ActionScript では、名前のルックアップは、Flash Player がプロトタイプチェーン内を移動するため時間のかかるプロセスでした。ActionScript 3.0 では、継承されたプロパティが、スーパークラスからサブクラスの特性オブジェクトにコピーされます。このため、名前のルックアップが効率的になり、所要時間も短縮されています。

特性オブジェクトはプログラマーコードに直接アクセスできませんが、パフォーマンスが向上しメモリ使用量が削減することからオブジェクトの存在がわかります。特性オブジェクトは、AVM2 にクラスのレイアウトと内容に関する詳細情報を提供します。この情報を使って、AVM2 は多くのマシン命令を直接生成することができます。このため、時間のかかる名前のルックアップを行わずにプロパティにアクセスしたり、メソッドを直接呼び出したりすることができ、実行時間を大幅に短縮できます。

特性オブジェクトによって、旧バージョンの ActionScript の類似オブジェクトに比べると、オブジェクトのメモリフットプリントをかなり小さくできます。例えば、クラスが sealed の場合（つまり、クラスが dynamic と宣言されていない場合）、クラスのインスタンスは動的に追加するプロパティにハッシュテーブルを必要とせず、このクラスで定義された固定プロパティの特性オブジェクトおよびスロットへのポインターを保持するだけです。その結果、ActionScript 2.0 で 100 バイトのメモリが必要だったオブジェクトが、ActionScript 3.0 では 20 バイトで済みます。

注意：特性オブジェクトは、内部実装の詳細です。ActionScript の今後のバージョンで変更されない、またはなくならないとは限りません。

プロトタイプオブジェクト

ActionScript のクラスオブジェクトには、クラスのプロトタイプオブジェクトを参照する prototype プロパティがあります。プロトタイプオブジェクトは、ActionScript のプロトタイプベース言語としてのルーツのレガシーです。詳しくは、ActionScript OOP サポートの歴史を参照してください。

prototype プロパティは読み取り専用で、別のオブジェクトを指すように変更することはできません。これは、旧バージョンの ActionScript のクラスの prototype プロパティとは異なります。旧バージョンのプロパティでは、別のクラスを指すようにプロトタイプを再割り当てできました。prototype プロパティは読み取り専用ですが、参照されるプロトタイプオブジェクトは読み取り専用ではありません。つまり、プロトタイプオブジェクトに新しいプロパティを追加することができます。プロトタイプオブジェクトに追加されたプロパティは、クラスのすべてのインスタンス間で共有されます。

ActionScript の以前のバージョンでは唯一の継承メカニズムであったプロトタイプチェーンは、ActionScript 3.0 では二次的な役割を果たすだけです。主な継承メカニズムである固定プロパティの継承は、特性オブジェクトによって内部的に処理されます。固定プロパティは、クラス定義の一部として定義される変数またはメソッドです。固定プロパティの継承は、class、extends、override などのキーワードで関連付けられる継承メカニズムなので、クラス継承とも呼ばれます。

プロトタイプチェーンは、固定プロパティの継承より動的な代替継承メカニズムになります。プロパティは、クラス定義の一部としてだけでなく、実行時にクラスオブジェクトの prototype プロパティからもクラスのプロトタイプオブジェクトに追加できます。ただし、コンパイラーを strict モードに設定した場合は、クラスを dynamic キーワードで宣言しない限り、プロトタイプオブジェクトに追加されたプロパティにアクセスできない場合があります。

いくつものプロパティがプロトタイプオブジェクトに関連付けられているクラスの好例として、Object クラスがあります。Object クラスの toString() および valueOf() メソッドは、実際は Object クラスのプロトタイプオブジェクトのプロパティに割り当てられた関数です。次の例は、これらのメソッドの宣言が理論的にどのように見えるかを示します。ただし、実装の詳細により実際の実装はやや異なります。

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

前述のとおり、プロパティは、クラス定義外部でクラスのプロトタイプオブジェクトに関連付けることができます。例えば、toString() メソッドは、次のように Object クラスの定義外部で定義することもできます。

```
Object.prototype.toString = function()
{
    // statements
};
```

ただし、固定プロパティの継承とは異なり、プロトタイプの継承では、サブクラスでメソッドを再定義する場合、override キーワードは必要ありません。以下に例を挙げます。Object クラスのサブクラスで valueOf() メソッドを再定義する場合は、次の 3 つのオプションがあります。1 つ目は、クラス定義内部のサブクラスのプロトタイプオブジェクトで、valueOf() メソッドを定義します。次のコードは、Foo という Object のサブクラスを作成し、Foo のプロトタイプオブジェクトで、クラス定義の一部として valueOf() メソッドを再定義します。すべてのクラスは Object から継承されるので、extends キーワードを使用する必要はありません。

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

2 番目に、クラス定義外部の Foo のプロトタイプオブジェクトで valueOf() メソッドを次のコードのように再定義できます。

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

3 番目に、Foo クラスの一部として、valueOf() という名前の固定プロパティを定義できます。この方法は、固定プロパティの継承とプロトタイプの継承が混在するという点で他の 2 つとは異なります。valueOf() を再定義する Foo のサブクラスでは、override キーワードを使用する必要があります。次のコードは、Foo で固定プロパティとして定義された valueOf() を示します。

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

AS3 名前空間

固定プロパティの継承およびプロトタイプの継承という 2 つの別個の継承メカニズムが混在することで、コアクラスのプロパティおよびメソッドに関して留意すべき互換性の問題が生じます。ActionScript が基づく ECMAScript 言語仕様との互換性からは、プロトタイプの継承を使用する必要があります。つまり、コアクラスのプロパティおよびメソッドは、そのクラスのプロトタイプオブジェクトで定義されます。一方、ActionScript 3.0 との互換性では、固定プロパティの継承を使用することが要求されます。つまり、コアクラスのプロパティおよびメソッドは、const、var、および function のキーワードを使用してクラス定義で定義されます。さらに、プロトタイプではなく固定プロパティの継承を使用すると、ランタイムパフォーマンスの大幅な向上につながります。

ActionScript 3.0 では、この問題を解決するために、コアクラスにプロトタイプの継承と固定プロパティの継承の両方を使用しています。各コアクラスに、2 セットのプロパティおよびメソッドが含まれます。1 セットは、ECMAScript 仕様との互換性のためにプロトタイプオブジェクトで定義され、残りの 1 セットは、ActionScript 3.0 との互換性のために固定プロパティおよび AS3 名前空間で定義されます。

AS3 名前空間は、この 2 セットのプロパティおよびメソッド間の選択のために便利なメカニズムを提供します。AS3 名前空間を使用しない場合、コアクラスのインスタンスは、コアクラスのプロトタイプオブジェクトに定義されているプロパティおよびメソッドを継承します。AS3 名前空間を使用することに決めた場合は、固定プロパティがプロトタイププロパティよりも常に優先されるため、コアクラスのインスタンスは AS3 バージョンを継承します。つまり、固定プロパティが利用可能な場合には、同じ名前のプロトタイププロパティではなく、その固定プロパティが必ず使用されます。

AS3 名前空間で修飾することによって、AS3 名前空間バージョンのプロパティまたはメソッドを選択的に使用することができます。例えば、次のコードでは、AS3 バージョンの Array.pop() メソッドを使用しています。

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

または、use namespace ディレクティブを使用して、コードブロック内のすべての定義に対して AS3 名前空間を開くことができます。例えば、次のコードでは、use namespace ディレクティブを使用して pop() メソッドと push() メソッドの両方に対して AS3 名前空間を開いています。

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

また、ActionScript 3.0 では、プロパティセットごとにコンパイラーオプションを備えており、AS3 名前空間をプログラム全体に適用できます。-as3 コンパイラーオプションは AS3 名前空間を表し、-es コンパイラーオプション（es は ECMAScript の略）はプロトタイプ継承オプションを表します。プログラム全体に対して AS3 名前空間を開くには、-as3 コンパイラーオプションを true に、-es コンパイラーオプションを false に設定します。プロトタイプバージョンを使用するには、両方のコンパイラーオプションを反対の値に設定します。Flash Builder および Flash Professional のデフォルトのコンパイラー設定は、-as3 = true および -es = false です。

いずれかのコアクラスを拡張してメソッドのいずれかをオーバーライドする計画がある場合には、オーバーライドメソッドの宣言方法に AS3 名前空間がどのように影響する可能性があるかを理解しておく必要があります。AS3 名前空間を使用する場合は、コアクラスメソッドのいずれのメソッドオーバーライドも、override 属性で AS3 名前空間を使用する必要があります。AS3 名前空間を使用せずにコアクラスメソッドをサブクラスで再定義する場合は、AS3 名前空間も override キーワードも使用しないでください。

例：GeometricShapes

サンプルアプリケーション GeometricShapes では、ActionScript 3.0 を使用して次のようにオブジェクト志向の概念および機能をいろいろと適用できることがわかります。

- クラスの定義
- クラスの拡張
- ポリモーフィズムと override キーワード
- インターフェイスの定義、拡張、実装

さらに、クラスインスタンスを作成する「ファクトリメソッド」も含まれ、インターフェイスのインスタンスとして戻り値を宣言し、返されたオブジェクトを汎用的な方法で使用する方法が示されます。

このサンプルのアプリケーションのファイルを入手するには、www.adobe.com/go/learn_programmingAS3samples_flash_jp を参照してください。GeometricShapes アプリケーションのファイルは、"Samples/GeometricShapes" フォルダーにあります。このアプリケーションは次のファイルで構成されています。

ファイル	説明
GeometricShapes.mxml または GeometricShapes.fla	Flash（FLA）または Flex（MXML）のメインアプリケーションファイル。
com/example/programmingas3/geometricshapes/IGeometricShape.as	すべての GeometricShapes アプリケーションクラスによって実装されるメソッドを定義する基本インターフェイス。
com/example/programmingas3/geometricshapes/IPolygon.as	複数の辺を持つ GeometricShapes アプリケーションクラスによって実装されるメソッドを定義するインターフェイス。
com/example/programmingas3/geometricshapes/RegularPolygon.as	シェイプの中心から対称の位置に等しい長さの辺を有する一種の幾何学図形。
com/example/programmingas3/geometricshapes/Circle.as	円を定義する一種の幾何学図形。
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	正三角形を定義する RegularPolygon のサブクラス。
com/example/programmingas3/geometricshapes/Square.as	正四角形を定義する RegularPolygon のサブクラス。
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	指定された種類およびサイズでシェイプを作成するためのファクトリメソッドを含むクラス。

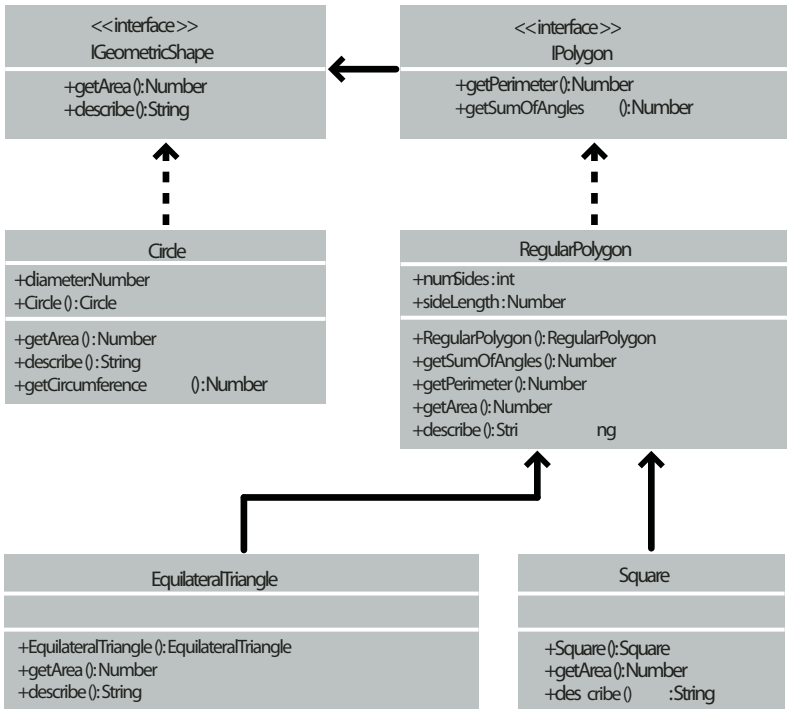
GeometricShapes クラスの定義

GeometricShapes アプリケーションを使用するユーザーは、幾何学図形の種類とサイズを指定することができます。それに対しアプリケーションは、シェイプの説明、その面積、および周辺の長さで応答します。

アプリケーションユーザーインターフェイスは、シェイプの種類を選択、サイズを設定、説明を表示するための小数のコントロールを含み、ごく普通です。このアプリケーションで最も興味深い部分は、内部にありそれはクラスおよびインターフェイス自体の構造に関係します。

このアプリケーションは幾何学図形を扱いますが、それを図では表示しません。

次の図に、この例で幾何学図形を定義するクラスとインターフェイスを UML（Unified Modeling Language）表記で示します。



GeometricShapes クラスの例

インターフェイスでの一般的な動作の定義

この GeometricShapes アプリケーションでは、円、四角形、および等辺三角形の 3 種類のシェイプを処理します。GeometricShapes クラス構造は、非常に単純なインターフェイスである、3 種類すべてのシェイプに共通するメソッドをリストする IGeometricShape で開始します。

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

インターフェイスでは、シェイプの領域を計算して返す getArea() メソッドと、シェイプのプロパティのテキスト説明を構成する describe() メソッドの 2 つのメソッドを定義します。

各シェイプの周囲の長さを知りたいこともあります。ただし、円の周囲は円周と呼ばれ、その計算方法は固有であるため、三角形または正方形とは動作が異なります。それでもなお、三角形、四角形、およびその他の多角形の間には十分な類似性が存在します。従って、これらに新しいインターフェイスクラス IPolygon を定義することは意味があります。IPolygon インターフェイスも、次に示すように、どちらかと言えば単純です。

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

このインターフェイスは、すべての多角形に共通する 2 つのメソッドを定義します。1 つは、すべての辺の長さの合計を測定する getPerimeter() メソッドで、もう 1 つはすべての内角の和を計算する getSumOfAngles() メソッドです。

IPolygon インターフェイスは、IGeometricShape インターフェイスを拡張します。これは、IPolygon インターフェイスを実装するクラスはいずれも、4 つのすべてのメソッド（IGeometricShape インターフェイスから 2 つ、IPolygon インターフェイスから 2 つ）を宣言する必要があることを意味します。

シェイプクラスの定義

各種類のシェイプに共通するメソッドに関する考えがまとまったら、シェイプクラス自体を定義することができます。実装する必要のあるメソッドの数で見た場合、最も単純なシェイプはここに示すように、Circle クラスです。

```
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;
        }
    }
}
```

Circle クラスは、IGeometricShape インターフェイスを実装するので、getArea() メソッドと describe() メソッドの両方に対するコードを提供する必要があります。さらに、Circle クラスに固有の getCircumference() メソッドを定義します。Circle クラスでは、他の多角形クラスでは見られない diameter プロパティの宣言もあります。

残りの 2 つの種類のシェイプの正方形および等辺三角形には、他の点で共通することがあります。つまり、等しい長さの辺を有する点で、両方について共通の式を使用して周辺と内角の総和を計算することができます。実際、これらの共通の式はまた、今後定義するあらゆる正多角形に適用されます。

RegularPolygon クラスは、Square クラスと EquilateralTriangle クラスの両方のスーパークラスです。スーパークラスを使用すると、共通メソッドを 1 か所で定義できるので、サブクラスごとに別々に共通メソッドを定義する必要がありません。RegularPolygon クラスのコードを次に示します。

```
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;
        }
    }
}
```

最初に、RegularPolygon クラスは、すべての正多角形に共通する 2 つのプロパティを宣言します。つまり、各辺の長さの sideLength プロパティと辺の数の numSides プロパティです。

RegularPolygon クラスは、IPolygon インターフェイスを実装し、IPolygon インターフェイスメソッドを 4 つすべて宣言します。そのうち getPerimeter() と getSumOfAngles() の 2 つのメソッドを共通の式を使用して実装します。

getArea() メソッドの式はシェイプによって異なるので、基本クラスバージョンのメソッドは、サブクラスメソッドで継承可能な共通の論理を含むことができません。代わりに、面積を計算しなかったことを示すには、単にデフォルト値 0 を返します。各シェイプの面積を正しく計算するには、RegularPolygon クラスのサブクラスが getArea() メソッド自体をオーバーライドする必要があります。

EquilateralTriangle クラスの次のコードは、getArea() メソッドをオーバーライドする方法を示しています。

```
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;
        }
    }
}
```

override キーワードは、EquilateralTriangle.getArea() メソッドが、RegularPolygon スーパークラスからの getArea() メソッドを意図的にオーバーライドすることを示します。EquilateralTriangle.getArea() メソッドが呼び出されると、前記のコードの式を使用して面積が計算されます。RegularPolygon.getArea() メソッドのコードが実行されることは決してありません。

対照的に、EquilateralTriangle クラスは独自のバージョンの getPerimeter() メソッドを定義しません。EquilateralTriangle.getPerimeter() メソッドが呼び出されると、継承チェーンに進み、RegularPolygon サブクラスの getPerimeter() メソッドのコードが実行されます。

EquilateralTriangle() コンストラクターは super() ステートメントを使用して、そのスーパークラスの RegularPolygon() コンストラクターを明示的に呼び出します。両方のコンストラクターのパラメーターセットが同じ場合は、EquilateralTriangle() コンストラクターを完全に省略しておくこともできますが、代わりに RegularPolygon() コンストラクターが実行されます。ただし、RegularPolygon() コンストラクターは追加パラメーターの numSides を必要とします。したがって、EquilateralTriangle() コンストラクターは、三角形が 3 辺であることを示すために len 入力パラメーターと値 3 を渡す super(len, 3) を呼び出します。

describe() メソッドも super() ステートメントを使用しますが、方法が異なります。describe() メソッドの RegularPolygon スーパークラスのバージョンを呼び出します。EquilateralTriangle.describe() メソッドは、最初に desc ストリング変数をシェイプの種類に関する記述に設定します。次に、super.describe() を呼び出して RegularPolygon.describe() メソッドの結果を取得し、その結果を desc ストリングに追加します。

ここでは Square クラスについて詳細は説明しません。しかし、このクラスは、EquilateralTriangle クラスとよく似ていて、コンストラクターと getArea() および describe() メソッドの独自の実装を提供します。

ポリモーフィズムとファクトリメソッド

インターフェイスと継承を有効に利用するクラスセットは、いろいろな興味深い方法で使用することができます。例えば、これまでに説明したすべてのシェイプクラスは、IGeometricShape インターフェイスを実装するか、またはそのようにスーパークラスを拡張します。したがって、IGeometricShape のインスタンスであるように変数を定義する場合は、インスタンスが実際には Circle のインスタンスであるか Square クラスのインスタンスであるかがわからなくても、describe() メソッドを呼び出すことができます。

次のコードは、この働きを示します。

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

myShape.describe() が呼び出された場合、変数が IGeometricShape インターフェイスのインスタンスと定義されていても Circle が基礎となるクラスであるので、Circle.describe() メソッドが実行されます。

この例からわかることは、ポリモーフィズムの動作原理です。つまり、正確に同じメソッド呼び出しの場合も、メソッドが呼び出されるオブジェクトのクラスに応じて、実行されるコードは異なります。

GeometricShapes アプリケーションでは、この種類のインターフェイスベースのポリモーフィズムを適用するために、ファクトリメソッドとして知られている簡略化バージョンの設計パターンを使用します。ファクトリメソッドという用語は、基になるデータ型または内容が状況によって変わる可能性のあるオブジェクトを返す関数を指します。

ここに示す GeometricShapeFactory クラスは、createShape() というファクトリメソッドを次のように定義します。

```
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();
        }
    }
}
```

createShape() ファクトリメソッドを使用すると、シェイプサブクラスのコンストラクターで、作成するインスタンスの詳細を定義できます。一方、IGeometricShape インスタンスとして新しいオブジェクトを返すと、さらに一般的な方法でアプリケーションによる処理ができるようになります。

前述の例の describeShape() メソッドは、さらに具体的なオブジェクトへの汎用的な参照を取得するために、アプリケーションでファクトリメソッドをどのように使用できるかを示しています。次のように、新しく作成された Circle オブジェクトの説明を取得することができます。

```
GeometricShapeFactory.describeShape("Circle", 100);
```

次に、describeShape() メソッドは同じパラメーター付きで createShape() ファクトリメソッドを呼び出しますが、IGeometricShape オブジェクトとして型指定された currentShape という静的変数には、新しい Circle オブジェクトが格納されています。次に、currentShape オブジェクトで describe() メソッドが呼び出され、その呼び出しは自動的に解決されて Circle.describe() を実行し、円の詳細な説明を返します。

サンプルアプリケーションの機能拡張

インターフェイスおよび継承の実際の威力は、アプリケーションを機能拡張または変更したときに明らかになります。

新しいシェイプとして五角形をこのサンプルアプリケーションに追加するとします。RegularPolygon クラスを拡張して、独自のバージョンの getArea() メソッドと describe() メソッドを定義する Pentagon クラスを作成します。次に、アプリケーションのユーザーインターフェイスのコンボボックスに新しい Pentagon オプションを追加します。以上です。Pentagon クラスは、継承によって RegularPolygon クラスから getPerimeter() メソッドと getSumOfAngles() メソッドの機能を自動的に取得します。IGeometricShape インターフェイスを実装するクラスから継承するため、Pentagon インスタンスは IGeometricShape インスタンスとしても扱うことができます。つまり、シェイプの新しい型を追加するために GeometricShapeFactory クラスのメソッドのメソッドシグネチャを変更する必要がないので、GeometricShapeFactory クラスを使用するコードも変更する必要がありません。

実習として Pentagon クラスを Geometric Shapes の例に追加してみると、アプリケーションに新機能を追加する負荷がインターフェイスおよび継承によってどの程度軽減できるかがわかります。