ACTIONSCRIPT® 3.0
Arbeitshandbuch

Rechtliche Hinweise

Rechtliche Hinweise finden Sie unter http://help.adobe.com/de_DE/legalnotices/index.html.

Letzte Aktualisierung 18.5.2011

Inhalt

Kapitel 1: Einführung in ActionScript 3.0
Informationen zu ActionScript 1
Vorzüge von ActionScript 3.0 1
Was ist neu in ActionScript 3.0 2

Kapitel 2: Erste Schritte mit ActionScript
Grundlagen der Programmierung 5
Arbeiten mit Objekten 7
Allgemeine Programmelemente 16
Beispiel: Animationen-Mustermappe (Flash Professional) 18
Erstellen von Anwendungen mit ActionScript 21
Erstellen eigener Klassen 25
Beispiel: Erstellen einer einfachen Anwendung 27

Kapitel 3: ActionScript-Sprache und -Syntax
Die Sprache im Überblick 36
Objekte und Klassen 37
Pakete und Namespaces 38
Variablen 48
Datentypen 51
Syntax 64
Operatoren 69
Bedingungen 75
Schleifen 77
Funktionen 79

Kapitel 4: Objektorientierte Programmierung mit ActionScript
Einführung in die objektorientierte Programmierung 91
Klassen 91
Schnittstellen 106
Vererbung 109
Erweiterte Themen 118
Beispiel: GeometricShapes 125

Kapitel 1: Einführung in ActionScript 3.0

Informationen zu ActionScript

ActionScript ist die Programmiersprache für die Laufzeitumgebungen Adobe® Flash® Player und Adobe® AIR™. Mit dieser Programmiersprache kann in Flash-, Flex- und AIR-Inhalten und -Anwendungen Interaktivität, die Verarbeitung von Daten und vieles andere mehr realisiert werden.

ActionScript-Anweisungen werden von der ActionScript Virtual Machine (AVM) ausgeführt, die Bestandteil von Flash Player und AIR ist. ActionScript-Code wird normalerweise von einem Compiler in das Bytecode-Format umgewandelt. (*Bytecode* ist eine Programmiersprache, die von Computern geschrieben und interpretiert werden kann.) Beispiele für Compiler sind der in Adobe® Flash® Professional integrierte Compiler sowie der Compiler, der in Adobe® Flash® Builder™ integriert ist und im Adobe® Flex™ SDK zur Verfügung steht. Der Bytecode ist in SWF-Dateien eingebettet, die von Flash Player und AIR ausgeführt werden.

ActionScript 3.0 stellt ein robustes Programmiermodell dar, das Entwicklern mit Grundkenntnissen der objektorientierten Programmierung sofort vertraut ist. Zu den wichtigsten Verbesserungen in ActionScript 3.0 im Vergleich mit früheren ActionScript-Versionen zählen:

- Eine neue ActionScript Virtual Machine (mit der Bezeichnung AVM2), die einen neuen Bytecode-Befehlssatz verwendet und deutliche Leistungsverbesserungen bietet.
- Eine modernere Compiler-Codebase, die tief greifendere Optimierungen als frühere Compiler-Versionen vornimmt.
- Eine erweiterte und verbesserte Anwendungsprogrammierschnittstelle (API, Application Programming Interface) mit Objektzugriff auf elementarster Ebene und einem rein objektorientierten Modell.
- Eine XML-API, die auf der E4X-Spezifikation (ECMAScript for XML, ECMA-357 Version 2) beruht. E4X ist eine Spracherweiterung für ECMAScript, mit der XML als nativer Datentyp der Sprache hinzugefügt wird.
- Ein Ereignismodell, das auf der DOM3-Ereignisspezifikation (Document Object Model Level 3) beruht.

Vorzüge von ActionScript 3.0

Die Scripting-Möglichkeiten von ActionScript 3.0 gehen weit über die vorheriger ActionScript-Versionen hinaus. ActionScript 3.0 wurde mit dem Ziel entwickelt, das Erstellen hochkomplexer Anwendungen mit umfangreichen Datensätzen und einer objektorientierten, wiederverwendbaren Codebasis zu ermöglichen. ActionScript 3.0 ist nicht für Inhalt erforderlich, der in Adobe Flash Player ausgeführt wird. Es ermöglicht jedoch Leistungssteigerungen, die nur mit AVM2 (ActionScript 3.0 Virtual Machine) erreicht werden können. ActionScript 3.0-Code kann bis zu zehn Mal schneller ausgeführt werden als älterer ActionScript-Code.

Mit der vorigen Version der ActionScript Virtual Machine, AVM1, wird ActionScript 1.0- und ActionScript 2.0-Code ausgeführt. Flash Player 9 und 10 unterstützen AVM1, um die Abwärtskompatibilität zu gewährleisten.

Was ist neu in ActionScript 3.0

ActionScript 3.0 enthält zahlreiche Klassen und Funktionen, die ActionScript 1.0 und 2.0 ähneln. Dennoch unterscheidet ActionScript 3.0 sich in seiner Architektur und auch konzeptionell von den älteren ActionScript-Versionen. Zu den Verbesserungen in ActionScript 3.0 zählen neue Funktionen der Kernsprache und eine verbesserte API, die eine umfangreichere Steuerung elementarer Objekte ermöglicht.

Funktionsumfang der Kernsprache

Die Kernsprache definiert die elementaren Baublöcke der Programmiersprache, z. B. Anweisungen, Ausdrücke, Bedingungen, Schleifen und Datentypen. ActionScript 3.0 enthält viele Funktionen, die den Entwicklungsprozess beschleunigen.

Ausnahmen zur Laufzeit

ActionScript 3.0 meldet mehr Fehlerzustände als vorherige ActionScript-Versionen. Für allgemeine Fehler kommen Laufzeitausnahmen zur Anwendung. Dies vereinfacht das Debuggen und ermöglicht Ihnen das Entwickeln von Anwendungen mit einer robusten Fehlerverarbeitung. Laufzeitfehler können Stack-Traces bereitstellen, die den Namen der Quellcodedatei und die Zeilennummer enthalten. Dies vereinfacht das Lokalisieren von Fehlerquellen.

Datentypisierung zur Laufzeit

In ActionScript 3.0 werden Typinformationen zur Laufzeit beibehalten. Diese Informationen werden verwendet, um zur Laufzeit Typüberprüfungen durchzuführen und die Typgenauigkeit des Systems zu erhöhen. Typangaben werden auch verwendet, um Variablen im nativen Prozessorformat abzubilden. Dies erhöht die Systemleistung und verringert den Speicherbedarf. Im Gegensatz dazu dienen Typanmerkungen in ActionScript 2.0 hauptsächlich als Hilfe bei der Entwicklung; alle Werte werden zur Laufzeit dynamisch typisiert.

Versiegelte Klassen

ActionScript 3.0 umfasst das Konzept der versiegelten Klassen. Eine versiegelte Klasse verfügt nur über den festen Satz an Eigenschaften und Methoden, die zur Kompilierzeit definiert werden. Es können keine weiteren Eigenschaften und Methoden hinzugefügt werden. Da Klassen nicht zur Laufzeit geändert werden können, ist eine striktere Prüfung zur Kompilierzeit möglich, wodurch robustere Programme entstehen. Außerdem wird die Speicherauslastung verbessert, da nun nicht mehr für jede Objektinstanz eine interne Hash-Tabelle erforderlich ist. Mithilfe des Schlüsselworts `dynamic` sind auch dynamische Klassen möglich. Alle Klassen in ActionScript 3.0 sind standardmäßig versiegelt, können jedoch mit dem Schlüsselwort `dynamic` als dynamische Klasse deklariert werden.

Methodenhüllen

ActionScript 3.0 ermöglicht es, bei Methodenhüllen automatisch die ursprüngliche Objektinstanz zu speichern. Diese Funktion ist bei der Ereignisverarbeitung nützlich. In ActionScript 2.0 wird bei Methodenhüllen nicht gespeichert, von welcher Objektinstanz sie jeweils abgeleitet wurden. Dies führt zu unerwartetem Verhalten beim Aufrufen der Methodenhüllen.

E4X (ECMAScript for XML)

ActionScript 3.0 implementiert die E4X-Spezifikation (ECMAScript for XML), die unlängst als ECMA-357 standardisiert wurde. E4X bietet einen natürlichen, leicht einsetzbaren Satz von Sprachkonstrukten zum Bearbeiten von XML. Im Gegensatz zu herkömmlichen XML-Parsing-APIs wird XML mit E4X hinsichtlich der Verarbeitungsgeschwindigkeit wie ein nativer Datentyp der Programmiersprache behandelt. E4X beschleunigt das Entwickeln von Anwendungen, die XML-Daten bearbeiten, indem der Umfang des dazu erforderlichen Programmcodes auf einen Bruchteil reduziert wird.

Die ECMA E4X-Spezifikation kann unter www.ecma-international.org nachgelesen werden.

Reguläre Ausdrücke
ActionScript 3.0 enthält native Unterstützung für reguläre Ausdrücke. Strings können so schnell gesucht und bearbeitet werden. ActionScript 3.0 implementiert Unterstützung für reguläre Ausdrücke entsprechend der Sprachspezifikation ECMAScript Version 3 (ECMA-262).

Namespaces
Namespaces ähneln den herkömmlichen Zugriffsbezeichnern, die zur Steuerung der Sichtbarkeit von Deklarationen eingesetzt werden (`public`, `private`, `protected`). Sie werden als benutzerdefinierte Zugriffsbezeichner mit frei wählbaren Namen verwendet. Namespaces sind mit einem URI (Universal Resource Identifier) ausgestattet, um Kollisionen zu vermeiden. Wenn Sie E4X einsetzen, werden sie auch zur Abbildung von XML-Namespaces verwendet.

Neue Grunddatentypen
ActionScript 3.0 enthält drei numerische Datentypen: „Number“, „int“ und „uint“. Der Datentyp „Number“ steht für eine Gleitkommazahl mit doppelter Genauigkeit. Der Datentyp „int“ ist eine vorzeichenbehaftete 32-Bit-Ganzzahl, mit der die Vorteile der schnellen Ganzzahlenarithmetik der CPU im ActionScript-Programmcode genutzt werden können. Der Datentyp „int“ eignet sich für Schleifenzähler und Variablen, bei denen Ganzzahlen verwendet werden. Beim Datentyp „uint“ handelt es sich um eine vorzeichenlose 32-Bit-Ganzzahl, die sich gut für RGB-Farbwerte, Bytezähler usw. eignet. In Gegensatz dazu verfügt ActionScript 2.0 nur über einen numerischen Datentyp, nämlich „Number“.

API-Funktionen

Die APIs in ActionScript 3.0 enthalten viele Klassen, mit denen Objekte auf elementarer Ebene gesteuert werden können. Die Sprache kann nun mit einer intuitiveren Architektur als in früheren Versionen aufwarten. Eine detaillierte Beschreibung aller Klassen würde den Rahmen dieser Dokumentation sprengen, doch einige wichtige Unterschiede sollen hier hervorgehoben werden.

DOM3-Ereignismodell
Das DOM3-Ereignismodell (Document Object Model Level 3) bietet eine Standardmethode zum Generieren und Verarbeiten von Ereignismeldungen. Dieses Ereignismodell soll die Interaktion und Kommunikation zwischen Objekten innerhalb von Anwendungen zulassen. Außerdem ermöglicht es Objekten, ihren Status beizubehalten und auf Änderungen zu reagieren. Das ActionScript 3.0-Ereignismodell basiert auf der Ereignisspezifikation nach World Wide Web Consortium DOM Level 3. Dieses Modell bietet einen übersichtlicheren und effizienteren Mechanismus als die Ereignissysteme in früheren ActionScript-Versionen.

Ereignisse und Fehlerereignisse befinden sich im flash.events-Paket. Die Flash Professional-Komponenten und die Flex-Umgebung nutzen dasselbe Ereignismodell, wodurch das Ereignissystem für die ganze Flash-Plattform einheitlich gestaltet ist.

Anzeigelisten-API
Die API für den Zugriff auf die Anzeigeliste – die Baumstruktur, die alle visuellen Elemente einer Anwendung enthält – besteht aus Klassen für den Einsatz visueller Grundtypen.

Die Sprite-Klasse ist ein schlanker Baustein, der als Basisklasse für visuelle Elemente vorgesehen ist, wie beispielsweise Komponenten der Benutzeroberfläche. Die Shape-Klasse repräsentiert unformatierte Vektorformen. Diese Klassen können wie gewohnt mit dem `new`-Operator instanziiert und jederzeit auch wieder dynamisch als übergeordnet deklariert werden.

Die Verwaltung der Tiefe erfolgt automatisch. Zum Festlegen und Verwalten der Stapelreihenfolge von Objekten sind Methoden verfügbar.

Verarbeitung dynamischer Daten und Inhalte

ActionScript 3.0 enthält intuitive und über die gesamte API konsistente Mechanismen zum Laden und Verarbeiten von Ressourcen und Daten in Anwendungen. Die Loader-Klasse stellt einen einheitlichen Mechanismus zum Laden von SWF-Dateien und Bildelementen bereit und bietet eine Möglichkeit, auf detaillierte Informationen der geladenen Inhalte zuzugreifen. Die URLLoader-Klasse stellt einen eigenen Mechanismus zum Laden von Text- und Binärdaten in datengesteuerten Anwendungen bereit. Mit der Socket-Klasse können Binärdaten in beliebigen Formaten von Server-Sockets gelesen oder in diese geschrieben werden.

Datenzugriff auf elementarer Ebene

Verschiedene APIs ermöglichen den Datenzugriff auf elementarer Ebene. Bei Daten, die heruntergeladen werden, ermöglicht die URLStream-Klasse noch während des Downloads den Zugriff auf die Daten als unformatierte Binärdaten. Mithilfe der ByteArray-Klasse können Sie die Lese- und Schreibvorgänge für Binärdaten sowie deren Verarbeitung optimieren. Die Sound-API ermöglicht die exakte Audiosteuerung mithilfe der SoundChannel- und SoundMixer-Klassen. Sicherheitsbezogene APIs stellen Informationen zu den Sicherheitsberechtigungen von SWF-Dateien und geladenen Inhalten bereit. So können durch Sicherheitsverletzungen hervorgerufene Fehler verarbeitet werden.

Arbeiten mit Text

ActionScript 3.0 enthält ein flash.text-Paket für alle textbezogenen APIs. Die TextLineMetrics-Klasse stellt detaillierte Metrikdaten für eine Textzeile in einem Textfeld bereit; sie ersetzt die `TextFormat.getTextExtent()`-Methode aus ActionScript 2.0. Die TextField-Klasse enthält elementare Methoden, die spezifische Informationen über eine Textzeile oder ein einzelnes Zeichen in einem Textfeld liefern. Beispielsweise gibt die `getCharBoundaries()`-Methode ein Rechteck zurück, das den Begrenzungsrahmen eines Zeichens darstellt. Die `getCharIndexAtPoint()`-Methode gibt den Index des Zeichens an einem bestimmten Punkt zurück. Die `getFirstCharInParagraph()`-Methode gibt den Index des ersten Zeichens in einem Absatz zurück. Zu den Methoden auf Zeilenebene zählen beispielsweise `getLineLength()` (gibt die Anzahl der Zeichen der angegebenen Textzeile zurück) und `getLineText()` (gibt den Text der angegebenen Zeile zurück). Die Font-Klasse bietet die Möglichkeit, in SWF-Dateien eingebettete Schriftarten zu verwalten.

Die Klassen im flash.text.engine-Paket bilden das Flash-Textmodul (Flash Text Engine) und ermöglichen die Textsteuerung auf noch elementarer Ebene. Diese Klassen ermöglichen die elementare Textsteuerung und sind für die Erstellung von Textstrukturen und Textkomponenten vorgesehen.

Kapitel 2: Erste Schritte mit ActionScript

Grundlagen der Programmierung

Da es sich bei ActionScript um eine Programmiersprache handelt, ist zum Erlernen von ActionScript das Verständnis einiger allgemeiner Konzepte der Computerprogrammierung hilfreich.

Was machen Computerprogramme?

Zunächst einmal müssen Sie verstehen, was ein Computerprogramm eigentlich ist und was es macht. Es gibt zwei wesentliche Aspekte bei einem Computerprogramm:

- Ein Programm ist eine Abfolge von Anweisungen oder Schritten, die der Computer ausführen soll.
- Jeder Schritt zieht letztlich eine Änderung von Informationen oder Daten nach sich.

Nüchtern betrachtet ist ein Computerprogramm nur eine Liste von Schritt-für-Schritt-Anweisungen, die an einen Computer übergeben und von dem sie dann nacheinander ausgeführt werden. Jeder einzelne Befehl wird auch als eine *Anweisung* (engl. Statement) bezeichnet. In ActionScript wird jede Anweisung mit einem Semikolon abgeschlossen.

Im Grunde genommen ändert ein Befehl in einem Computerprogramm lediglich einige im Speicher abgelegte Datenbits. Ein einfaches Beispiel: Der Computer wird angewiesen, zwei Zahlen zu addieren und das Ergebnis zu speichern. Ein etwas komplexeres Beispiel: Auf dem Bildschirm ist ein Rechteck gezeichnet, und Sie möchten ein Programm schreiben, mit dem das Rechteck an eine andere Stelle des Bildschirms verschoben wird. Der Computer merkt sich bestimmte Informationen zu diesem Rechteck: die x- und y-Koordinaten, die Breite und Höhe, die Farbe und so weiter. Jedes dieser Informationsbits wird an einer bestimmten Stelle im Speicher abgelegt. Ein Programm, mit dem das Rechteck an eine andere Stelle verschoben wird, enthält beispielsweise Schritte wie „die x-Koordinate zu 200 ändern; die y-Koordinate zu 150 ändern“. Anders ausgedrückt: Es gibt neue Werte für die x- und y-Koordinaten an. Hinter den Kulissen manipuliert der Computer diese Daten so, dass die Zahlen in das Bild umgewandelt werden, das auf dem Computerbildschirm angezeigt wird. Grundsätzlich müssen Sie jedoch nur wissen, dass beim „Verschieben eines Rechtecks auf dem Bildschirm“ lediglich Datenbits im Speicher des Computers geändert werden.

Variablen und Konstanten

Die Programmierung hat in erster Linie den Zweck, Informationseinheiten im Speicher des Computers zu ändern. Deshalb muss eine Möglichkeit gegeben sein, eine einzelne Informationseinheit in einem Programm darzustellen. Zu diesen Zweck werden Variablen eingesetzt. Eine *Variable* ist der Name für einen Wert im Speicher des Computers. Beim Schreiben von Anweisungen zum Ändern von Werten verwenden Sie anstelle des tatsächlichen Wertes den Namen der Variablen. Jedes Mal, wenn der Variablenname im Programm erkannt wird, wird der Speicher des Computers durchsucht und der dort gefundene Wert verwendet. Angenommen, Sie arbeiten mit zwei Variablen namens `value1` und `value2`, die beide eine Zahl enthalten. Um diese beiden Zahlen zu addieren, können Sie die folgende Anweisung schreiben:

```
value1 + value2
```

Wenn der Computer die Programmschritte ausführt, sucht er nach dem Wert in jeder Variablen und addiert die Werte.

In ActionScript 3.0 setzt sich eine Variable aus drei unterschiedlichen Teilen zusammen:

- Variablenname

- Datentyp, der in der Variablen gespeichert werden kann
- Tatsächlicher Wert, der im Speicher abgelegt ist

Sie haben gesehen, wie der Computer den Namen als Platzhalter für den Wert verwendet. Auch der Datentyp ist wichtig. Wenn Sie eine Variable in ActionScript erstellen, geben Sie den Typ der Daten an, die die Variable enthalten soll. Nach Angabe des Datentyps können die Programmanweisungen nur Daten mit dem jeweiligen Typ in der Variablen speichern. Sie können den Wert mit den spezifischen Merkmalen des Datentyps ändern. In ActionScript erstellen Sie eine Variable mit der Anweisung *var* (dieser Vorgang wird auch als `Deklarieren` einer Variablen bezeichnet):

```
var value1:Number;
```

In diesem Beispiel wird der Computer angewiesen, eine Variable namens `value1` zu erstellen, die nur numerische Daten (Number-Datentyp) enthalten kann. („Number“ ist ein bestimmter in ActionScript definierter Datentyp.) Sie können auch gleich einen Wert in der Variablen speichern:

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

Adobe Flash Professional
In Flash Professional gibt es noch eine weitere Möglichkeit, eine Variable zu deklarieren. Wenn Sie ein Objekt wie ein Movieclip-Symbol, ein Schaltflächen-Symbol oder ein Textfeld auf der Bühne platzieren, können Sie diesem Objekt im Eigenschafteninspektor einen Instanznamen geben. Hinter den Kulissen erstellt Flash Professional eine Variable mit demselben Namen wie der Instanzname. Sie können diesen Namen in Ihrem ActionScript-Code verwenden, um das jeweilige Bühnenobjekt darzustellen. Angenommen, auf der Bühne befindet sich ein Movieclip-Symbol, dem Sie den Instanznamen `rocketShip` zuweisen. Bei jeder Verwendung der Variablen `rocketShip` im ActionScript-Code ist der jeweilige Movieclip betroffen.

Eine *Konstante* ähnelt einer Variablen. Es handelt sich dabei um einen Namen, der einen Wert im Speicher des Computers mit einem angegebenen Datentyp darstellt. Der Unterschied besteht darin, dass einer Konstanten im Verlauf einer ActionScript-Anwendung nur ein einziges Mal ein Wert zugewiesen werden kann. Wenn der Wert einer Konstanten einmal zugewiesen ist, bleibt er in der gesamten Anwendung stets gleich. Zum Deklarieren einer Konstante gilt fast die gleiche Syntax wie zum Deklarieren einer Variablen. Der einzige Unterschied ist, dass das `const`-Schlüsselwort anstelle von `var` verwendet wird:

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

Eine Konstante ist nützlich, um einen Wert zu definieren, der in einem Projekt mehrfach verwendet wird und sich unter normalen Bedingungen nicht ändert. Die Verwendung einer Konstanten anstelle eines Literalwerts verbessert die Lesbarkeit von Code. Betrachten wir beispielsweise zwei Versionen desselben Codes. Eine Version multipliziert einen Preis mit `SALES_TAX_RATE` (Mehrwertsteuersatz). Die andere Version multipliziert den Preis mit `0.07`. Die Version mit der Konstante `SALES_TAX_RATE` ist verständlicher. Nehmen wir auch an, dass der von der Konstante definierte Wert sich doch ändert. Wenn Sie eine Konstante verwenden, um den Wert im ganzen Projekt darzustellen, können Sie den Wert an einer zentralen Stelle ändern (nämlich in der Deklaration der Konstanten). Bei Verwendung von fest kodierten Literalwerten müssten Sie den Wert dagegen an verschiedenen Stellen ändern.

Datentypen

In ActionScript gibt es verschiedene Datentypen, die Sie Ihren erstellten Variablen zuweisen können. Einige dieser Datentypen sind „einfache“ oder „grundlegende“ Datentypen.

- String: Ein Textwert, wie ein Name oder der Text eines Buchkapitels

- Numerisch: ActionScript3.0 umfasst drei Datentypen für numerische Daten:
  - Number: jeder numerische Wert, ganzzahlig oder als Bruch
  - int: ein Integer-Wert (eine ganze Zahl ohne Kommastelle)
  - uint: ein vorzeichenloser Integer-Wert, also eine ganze Zahl, die nicht negativ sein kann
- Boolean: ein true- oder false-Wert (wahr oder falsch), beispielsweise, ob ein Schalter gesetzt ist oder ob zwei Werte gleich sind

Die einfachen Datentypen stellen nur eine Informationseinheit dar, z. B. eine einzelne Zahl oder eine einzelne Textfolge. Die meisten in ActionScript definierten Datentypen sind jedoch komplexe Datentypen. Sie stellen eine Wertmenge in einem einzelnen Container dar. Beispielsweise repräsentiert eine Variable mit dem Date-Datentyp einen einzelnen Wert (einen bestimmten Augenblick). Trotzdem setzt sich dieser Datumswert aus mehreren Werten zusammen: Tag, Monat, Jahr, Stunden, Minuten, Sekunden usw. – alles einzelne Zahlen. Ein Datum wird meist als einzelner Wert betrachtet und Sie können dies umsetzen, indem Sie eine Date-Variable erstellen. Intern wird das Datum vom Computer jedoch als Gruppe mehrerer Werte behandelt, die gemeinsam ein einzelnes Datum definieren.

Die meisten integrierten Datentypen sind, ebenso wie die von Programmierern definierten Datentypen, komplexe Datentypen. Einige komplexe Datentypen kennen Sie wahrscheinlich schon:

- MovieClip: ein Movieclip-Symbol
- TextField: ein dynamisches oder Eingabetextfeld
- SimpleButton: ein Schaltflächen-Symbol
- Date: Informationen zu einem bestimmten Augenblick (ein Datum und eine Uhrzeit)

Zwei Wörter, die häufig als Synonyme für den Begriff „Datentyp“ verwendet werden, sind „Klasse“ und „Objekt“. Eine *Klasse* ist die Definition eines Datentyps. Sie lässt sich mit einer Vorlage für alle Objekte mit dem jeweiligen Datentyp vergleichen und besagt, dass „alle Variablen des Beispiel-Datentyps die Merkmale A, B und C haben“. Im Gegensatz dazu ist ein *Objekt* eine bestimmte Instanz einer Klasse. Eine Variable mit dem MovieClip-Datentyp kann beispielsweise als MovieClip-Objekt beschrieben werden. Es gibt verschiedene Möglichkeiten, die gleiche Sache auszudrücken:

- Der Datentyp der Variablen `myVariable` ist Number
- Die Variable `myVariable` ist eine Number-Instanz
- Die Variable `myVariable` ist ein Number-Objekt
- Die Variable `myVariable` ist eine Instanz der Number-Klasse

Arbeiten mit Objekten

ActionScript ist eine objektorientierte Programmiersprache. Bei der objektorientierten Programmierung handelt es sich um eine Programmiermethode. Dabei wird der Code eines Programms mithilfe von Objekten organisiert.

Wir haben den Begriff „Computerprogramm“ bereits als Folge von Schritten oder Anweisungen definiert, die vom Computer ausgeführt werden. Analog können Sie sich ein Computerprogramm vereinfacht als eine lange Liste mit Befehlen vorstellen. Bei der objektorientierten Programmierung sind die Programmanweisungen aber in verschiedene Objekte unterteilt. Der Code wird in Funktionalitätsblöcke gruppiert, das heißt, zugehörige Funktionalitätstypen oder Informationseinheiten sind in einem Container gruppiert.

Adobe Flash Professional

Wenn Sie bereits mit Symbolen in Flash Professional gearbeitet haben, ist Ihnen das Konzept der Objekte bereits vertraut. Angenommen, Sie haben ein Movieclip-Symbol definiert, beispielsweise die Zeichnung eines Rechtecks, und eine Kopie davon auf der Bühne platziert. Dieses Movieclip-Symbol ist auch ein Objekt in ActionScript, eine Instanz der MovieClip-Klasse.

Ein Movieclip weist verschiedene Eigenschaften auf, die Sie ändern können. Wenn der Movieclip ausgewählt ist, können Sie Werte im Eigenschafteninspektor ändern, wie die x-Koordinate oder die Breite. Außerdem können Sie die Farbe auf verschiedene Weise anpassen, indem Sie beispielsweise den Alphawert (die Transparenz) ändern oder einen Filter für Schlagschatten anwenden. Andere Werkzeuge in Flash Professional ermöglichen Ihnen weitere Änderungen; so können Sie das Rechteck mit dem Werkzeug für die freie Transformation beispielsweise drehen. Alle diese Methoden zum Ändern eines Movieclip-Symbols in Flash Professional sind auch in ActionScript verfügbar. Sie ändern den Movieclip in ActionScript, indem Sie die Dateneinheiten ändern, die zentral in einem sogenannten MovieClip-Objekt kombiniert werden.

Bei der objektorientierten Programmierung in ActionScript weist jede Klasse drei Merkmale auf:

- Eigenschaften
- Methoden
- Ereignisse

Diese Merkmale dienen zum Verwalten der vom Programm verwendeten Datenteile und zur Entscheidung, welche Aktionen in welcher Reihenfolge ausgeführt werden.

Eigenschaften

Eine Eigenschaft stellt einen der Datenteile dar, der mit anderen Datenteilen zu einem Objekt zusammengefasst wird. Beispiel: Ein Song-Objekt kann Eigenschaften namens `artist` und `title` aufweisen; die MovieClip-Klasse enthält Eigenschaften wie `rotation`, `x`, `width` und `alpha`. Sie arbeiten mit Eigenschaften wie mit einzelnen Variablen. Genau genommen lassen sich Eigenschaften mit den „untergeordneten“ Variablen vergleichen, die in einem Objekt enthalten sind.

Im Folgenden sind einige ActionScript-Codebeispiele aufgeführt, in denen Eigenschaften verwendet werden: Mit dieser Codezeile wird der Movieclip namens `square` zur x-Koordinate 100 Pixel verschoben:

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

Im folgenden Code wird die rotation-Eigenschaft verwendet, um den MovieClip `square` so zu drehen, dass er der Drehung des MovieClips `triangle` entspricht:

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

Im folgenden Code wird der horizontale Maßstab des MovieClips `square` so geändert, dass er anderthalb Mal breiter als zuvor ist:

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

Beachten Sie die allgemeine Syntax: Sie verwenden eine Variable (`square`, `triangle`) als Namen des Objekts, gefolgt von einem Punkt (`.`) und dem Namen der Eigenschaft (`x`, `rotation`, `scaleX`). Der Punkt (auch als *Punktoperator* bezeichnet) kennzeichnet, dass Sie auf eines der untergeordneten Elemente des Objekts zugreifen. Die gesamte Struktur, „Variablenname-Punkt-Eigenschaftenname“, wird wie eine einzelne Variable als Name für einen Wert im Speicher des Computers verwendet.

Methoden

Eine *Methode* ist eine Aktion, die ein Objekt ausführen kann. Angenommen, Sie haben in Flash Professional ein Movieclip-Symbol erstellt, dessen Zeitleiste mehrere Schlüsselbilder und Animationen enthält. Dieser Movieclip kann abgespielt, angehalten oder angewiesen werden, den Abspielkopf zu einem bestimmten Bild zu verschieben

Mit dem folgenden Code wird der Movieclip namens `shortFilm` angewiesen, die Wiedergabe zu starten:

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

Mit der folgenden Codezeile wird der Movieclip namens `shortFilm` angehalten (der Abspielkopf hält an, wie das Unterbrechen der Wiedergabe von Video):

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

Mit dem folgenden Code werden der Abspielkopf im Movieclip namens `shortFilm` auf das erste Bild verschoben und die Wiedergabe gestoppt (wie das Zurückspulen eines Video):

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

Der Zugriff auf Methoden erfolgt genauso wie auf Eigenschaften: Sie schreiben den Objektnamen (eine Variable), dann einen Punkt und schließlich den Methodennamen, gefolgt von runden Klammern. Die Klammern geben an, dass Sie die Methode *aufrufen*, also das Objekt anweisen, die jeweilige Aktion auszuführen. Manchmal werden auch Werte (oder Variablen) in den runden Klammern angegeben. Auf diese Weise werden zusätzliche Informationen übergeben, die zum Ausführen der Aktion erforderlich sind. Diese Werte werden als *Parameter* der Methode bezeichnet. Beispielsweise muss die `gotoAndStop()`-Methode wissen, zu welchem Bild sie springen soll, also benötigt sie einen Parameter in den Klammern. Andere Methoden, z. B. `play()` und `stop()`, sind selbsterklärend und benötigen keine zusätzlichen Informationen. Trotzdem werden auch diese Methoden mit Klammern angegeben.

Im Gegensatz zu Eigenschaften (und Variablen) werden Methoden nicht als Platzhalter für Werte verwendet. Dennoch können einige Methoden Berechnungen durchführen und ein Ergebnis zurückgeben, das wie eine Variable verwendet werden kann. Beispielsweise wandelt die Methode `toString()` der Number-Klasse einen numerischen Wert in die entsprechende Textdarstellung um:

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

Beispielsweise können Sie mit der `toString()`-Methode den Wert einer Number-Variablen in einem Textfeld auf dem Bildschirm anzeigen. Die `text`-Eigenschaft der TextField-Klasse wird als String definiert und kann daher nur Textwerte enthalten. (Die `text`-Eigenschaft repräsentiert den Textinhalt, der auf dem Bildschirm angezeigt wird.) Mit dieser Codezeile wird der numerische Wert in der `numericData`-Variablen in Text konvertiert. Anschließend wird der konvertierte Wert auf dem Bildschirm im TextField-Objekt namens `calculatorDisplay` angezeigt:

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

Ereignisse

Ein Computerprogramm ist eine Folge von Anweisungen, die der Computer nacheinander ausführt. Einige einfache Computerprogramme bestehen lediglich aus einigen wenigen Schritten, die der Computer ausführt, und dann ist das Programm beendet. ActionScript-Programme hingegen sind so ausgelegt, dass sie immer weiter ausgeführt werden und auf eine Eingabe durch den Benutzer oder andere Ereignisse warten. Ereignisse sind Mechanismen, mit denen festgelegt wird, welche Befehle der Computer zu welchem Zeitpunkt ausführt.

Einfach erklärt sind *Ereignisse* (Englisch „events“) bestimmte Dinge, die eintreten, ActionScript bekannt sind und eine Reaktion hervorrufen. Viele Ereignisse beziehen sich auf eine Interaktion des Benutzers; beispielsweise klickt der Benutzer auf eine Schaltfläche oder drückt eine Taste auf der Tastatur. Es gibt auch andere Ereignistypen. Angenommen, Sie verwenden ActionScript zum Laden eines externen Bilds. In diesem Fall tritt irgendwann ein Ereignis ein, das Ihnen mitteilt, dass das Bild fertig geladen ist. Während der Ausführung wartet ein ActionScript-Programm genau genommen nur darauf, dass bestimmte Ereignisse auftreten. Sobald diese Ereignisse auftreten, wird der ActionScript-Code, den Sie für die Ereignisse angegeben haben, ausgeführt.

Ereignisverarbeitung

Die Technik zur Angabe von Aktionen, die als Reaktion auf bestimmte Ereignisse ausgeführt werden sollen, wird als *Ereignisverarbeitung* (Englisch „event handling“) bezeichnet. Wenn Sie einen ActionScript-Code zur Ereignisverarbeitung schreiben, müssen Sie drei wichtige Elemente angeben:

- Ereignisquelle: An welchem Objekt wird das Ereignis eintreten? Beispiele hierfür sind die Schaltfläche, auf die geklickt wurde, oder das Loader-Objekt, mit dem das Bild geladen wird. Die Ereignisquelle wird auch als *Ereignisziel* bezeichnet. Dieser Name wird verwendet, da es sich um das Objekt handelt, auf das das Ereignis abzielt (also an dem das Ereignis tatsächlich stattfindet).
- Ereignis: Was wird eintreten, was soll eine Reaktion hervorrufen? Eine korrekte Angabe des genauen Ereignisses ist besonders wichtig, da viele Objekte mehrere Ereignisse auslösen.
- Reaktion: Welche Schritte sollen ausgeführt werden, wenn das Ereignis eintritt?

Diese drei Elemente sind immer erforderlich, wenn Sie ActionScript-Code zur Ereignisverarbeitung schreiben. Der Code weist folgende Grundstruktur auf (Elemente in Fettdruck sind Platzhalter, die Sie von Fall zu Fall angeben):

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

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

Dieser Code bewirkt zweierlei. Zunächst definiert er eine Funktion. Auf diese Weise werden die Aktionen festgelegt, die als Reaktion auf das eingetretene Ereignis ausgeführt werden sollen. Anschließend ruft er die `addEventListener()`-Methode des Quellobjekts auf. Durch den Aufruf von `addEventListener()` wird die Funktion für das angegebene Ereignis festgelegt. Wenn das Ereignis eintritt, werden die Aktionen der Funktion ausgeführt. Betrachten Sie diese Teile genauer.

Mit einer *Funktion* können Sie Aktionen unter einem Namen zusammenfassen, der wie ein verkürzter Name zum Ausführen der Aktionen aufgerufen werden kann. Eine Funktion ist weitgehend mit einer Methode identisch, sie ist jedoch nicht unbedingt einer bestimmten Klasse zugeordnet. (Genau genommen ist eine „Methode“ eine Funktion, die einer bestimmten Klasse zugeordnet ist.) Bei der Erstellung einer Funktion für die Ereignisverarbeitung wählen Sie den Namen für die Funktion aus (`eventResponse` in diesem Fall). Weiterhin geben Sie einen Parameter an (`eventObject` in diesem Beispiel). Die Angabe eines Funktionsparameters erfolgt wie das Deklarieren einer Variablen, Sie müssen also auch den Datentyp des Parameters angeben. (In diesem Beispiel hat der Parameter den Datentyp `EventType`.)

Jedem Ereignistyp, auf den Sie warten möchten, ist eine ActionScript-Klasse zugeordnet. Der Datentyp, den Sie für den Funktionsparameter angeben, ist immer die zugeordnete Klasse des jeweiligen Ereignisses, auf das Sie reagieren möchten. Ein `click`-Ereignis (wird ausgelöst, wenn der Benutzer mit der Maus auf ein Element klickt) ist z. B. der MouseEvent-Klasse zugeordnet. Um eine Listener-Funktion für ein `click`-Ereignis zu schreiben, definieren Sie die Listener-Funktion mit einem Parameter, der den Datentyp „MouseEvent“ aufweist. Abschließend geben Sie in geschweiften Klammern ({ ... }) die Anweisungen an, die beim Eintreten des Ereignisses ausgeführt werden sollen.

Die Funktion für die Ereignisverarbeitung wird geschrieben. Im nächsten Schritt instruieren Sie das Ereignisquellobjekt (das Objekt, bei dem das Ereignis eintritt, beispielsweise die Schaltfläche), dass es die Funktion aufrufen soll, wenn das Ereignis eintritt. Sie registrieren die Funktion mit dem Ereignisquellobjekt, indem Sie die `addEventListener()`-Methode dieses Objekts aufrufen. (Alle Objekte, die Ereignisse enthalten, haben auch eine `addEventListener()`-Methode.) Die `addEventListener()`-Methode hat zwei Parameter:

- Zunächst der Name des Ereignisses, auf das reagiert werden soll. Jedes Ereignis ist einer bestimmten Klasse zugeordnet. Jede Ereignisklasse hat einen besonderen Wert für jedes ihrer Ereignisse. Dieser Wert lässt sich mit einem eindeutigen Namen vergleichen. Sie verwenden diesen Wert für den ersten Parameter.
- Der zweite Parameter ist der Name der Funktion, die als Reaktion auf das Ereignis ausgeführt wird. Beachten Sie, dass ein Funktionsname ohne runde Klammern geschrieben werden muss, wenn er als ein Parameter übergeben wird.

Der Prozess der Ereignisverarbeitung

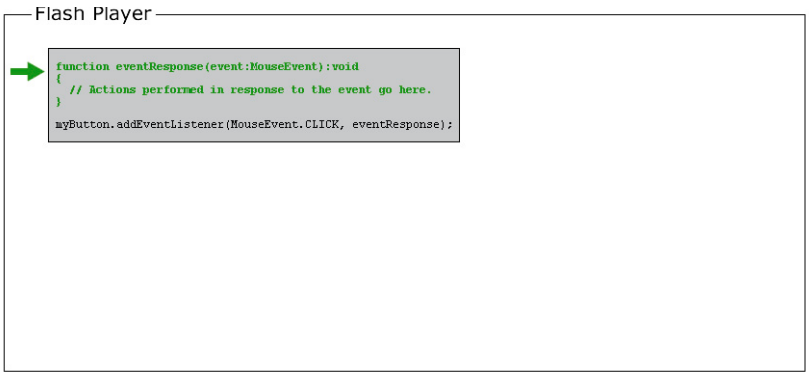
Das Folgende ist eine schrittweise Beschreibung des Prozesses, der beim Erstellen eines Ereignis-Listeners ausgeführt wird. In diesem Fall ist es ein Beispiel für das Erstellen einer Listener-Funktion, die aufgerufen wird, wenn auf ein Objekt namens `myButton` geklickt wird.

Der tatsächliche, vom Programmierer erstellte Code lautet wie folgt:

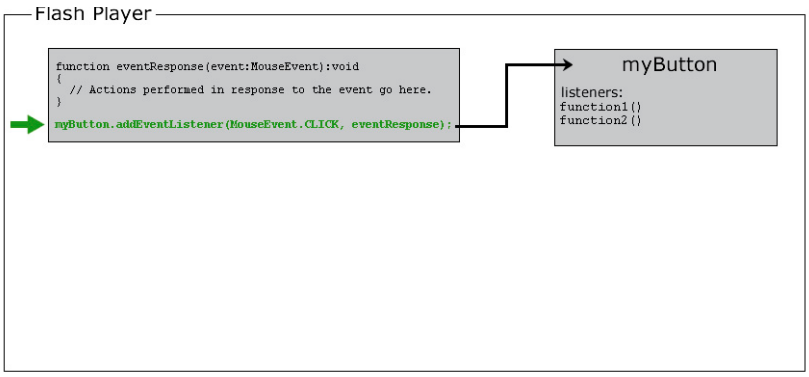
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

Dieser Code würde folgendermaßen funktionieren, wenn er ausgeführt wird:

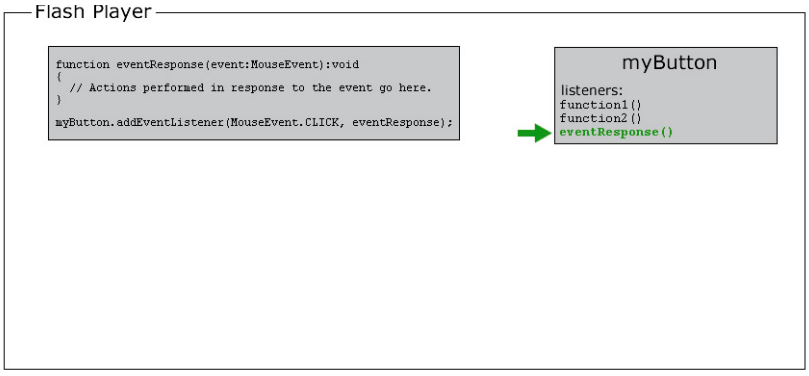
1 Wenn die SWF-Datei geladen wird, erkennt der Computer, dass eine Funktion namens `eventResponse()` vorhanden ist.



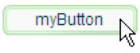
2 Der Code wird dann ausgeführt (genauer die Codezeilen, die sich nicht in einer Funktion befinden). In diesem Fall ist dies nur eine Codezeile: Aufrufen der `addEventListener()`-Methode für das Quellobjekt des Ereignisses (mit dem Namen `myButton`) und Übergeben der `eventResponse`-Funktion als Parameter.



Intern verwaltet `myButton` eine Liste der Funktionen, die auf die einzelnen Ereignisse warten. Wenn die `addEventListener()`-Methode aufgerufen wird, speichert `myButton` die `eventResponse()`-Funktion in der Liste der Ereignis-Listener.

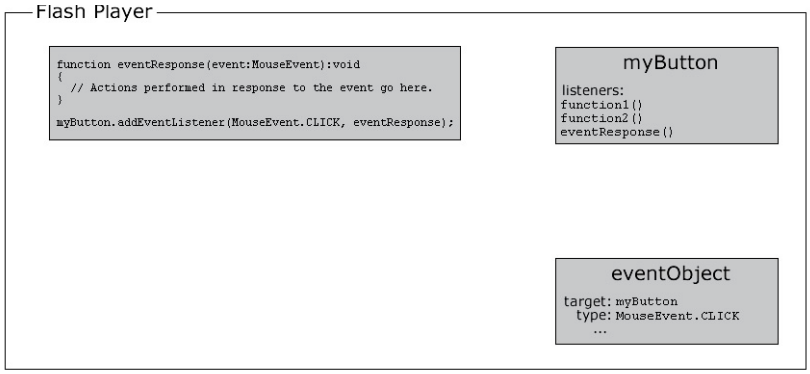


3 Wenn der Benutzer zu einem beliebigen Zeitpunkt auf das `myButton`-Objekt klickt, wird das `click`-Ereignis ausgelöst (im Code als `MouseEvent.CLICK` gekennzeichnet).

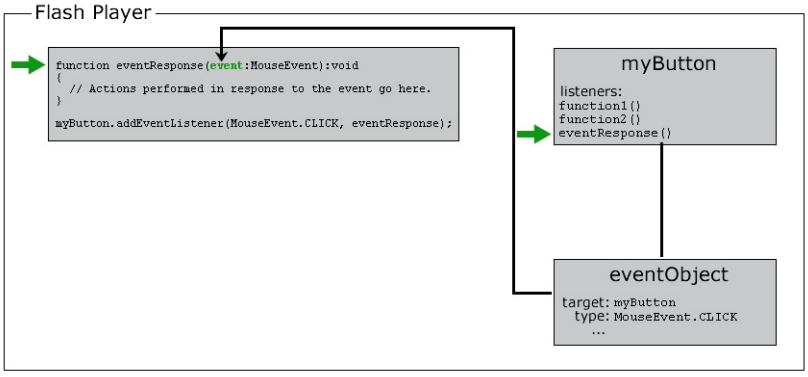


Zu diesem Zeitpunkt geschieht Folgendes:

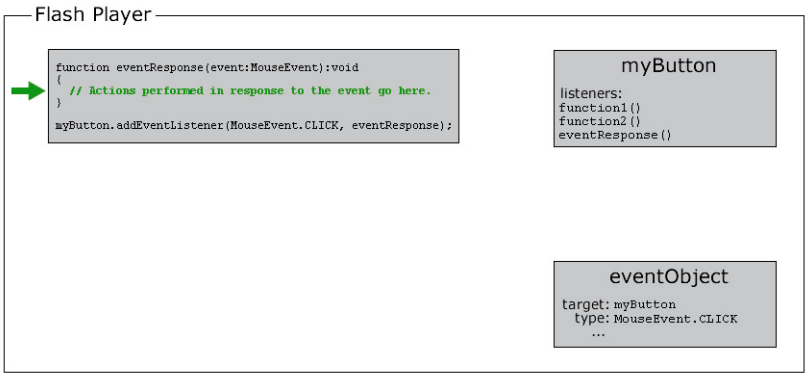
a Ein Objekt wird erstellt, das eine Instanz der Klasse ist, die dem jeweiligen Ereignis zugeordnet ist (MouseEvent in diesem Beispiel). Bei vielen Ereignissen ist dieses Objekt eine Instanz der Event-Klasse. Bei Mausereignissen handelt es sich um eine MouseEvent-Instanz. Bei anderen Ereignissen ist es eine Instanz der Klasse, die dem Ereignis zugeordnet ist. Das hierbei erstellte Objekt wird als *Ereignisobjekt* bezeichnet und enthält bestimmte Informationen zum eingetretenen Ereignis: Ereignistyp, Position des Ereignisses und andere ereignisspezifische Informationen (sofern anwendbar).



b Der Computer prüft dann die Liste der Ereignis-Listener, die von `myButton` gespeichert wurde. Es durchläuft diese Funktionen nacheinander, ruft jede Funktion auf und übergibt das Ereignisobjekt als Parameter an die Funktion. Da die `eventResponse()`-Funktion einer der Listener von `myButton` ist, wird die `eventResponse()`-Funktion als Teil dieses Prozesses aufgerufen.



c Beim Aufrufen der `eventResponse()`-Funktion wird der Code in dieser Funktion ausgeführt; somit werden die von Ihnen angegebenen Aktionen ausgeführt.



Beispiele für die Ereignisverarbeitung

Im Folgenden werden weitere konkrete Beispiele für Code zur Ereignisverarbeitung gezeigt. Diese Beispiele zeigen Ihnen einige übliche Ereigniselemente sowie mögliche Variationen, die beim Schreiben von Code zur Ereignisverarbeitung verfügbar sind:

- Klicken auf eine Schaltfläche zum Starten der Wiedergabe des aktuellen Movieclips. Im folgenden Beispiel ist `playButton` der Instanzname der Schaltfläche und `this` bezieht sich auf das aktuelle Objekt:

```
this.stop();

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

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

- Erfassen einer Eingabe in einem Textfeld. Im folgenden Beispiel ist `entryText` ein Eingabetextfeld und `outputText` ein dynamisches Textfeld:

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

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

- Klicken auf eine Schaltfläche zur Navigation zu einer URL. In diesem Fall ist `linkButton` der Instanzname der Schaltfläche:

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

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

Erstellen von Objektinstanzen

Bevor Sie ein Objekt in ActionScript verwenden können, muss es zunächst einmal existieren. Ein Teil beim Erstellen eines Objekts ist das Deklarieren einer Variablen, jedoch wird hierbei lediglich ein leerer Bereich im Speicher des Computers erstellt. Bevor Sie eine Variable verwenden oder ändern, müssen Sie ihr immer einen Wert zuweisen (ein Objekt erstellen und in der Variablen speichern). Die Erstellung eines Objekts wird als *Instanziieren* des Objekts bezeichnet. Anders ausgedrückt: Sie erstellen eine Instanz einer bestimmten Klasse.

Bei einem einfachen Verfahren zum Erstellen einer Objektinstanz wird kein ActionScript verwendet. Platzieren Sie in Flash Professional ein Movieclip-Symbol, ein Schaltflächensymbol oder ein Textfeld auf der Bühne und weisen Sie ihm einen Instanznamen zu. Flash Professional deklariert automatisch eine Variable mit diesem Instanznamen, erstellt eine Objektinstanz und speichert dieses Objekt in der Variablen. In Flex erstellen Sie auf ähnliche Weise eine Komponente in MXML, indem Sie entweder ein MXML-Tag kodieren oder die Komponente im Entwurfsmodus von Flash Builder im Editor platzieren. Wenn Sie dieser Komponente eine ID zuweisen, wird diese ID als Name einer ActionScript-Variable verwendet, die die jeweilige Komponenteninstanz enthält.

Doch wahrscheinlich möchten Sie Objekte nicht immer auf visuelle Weise erstellen, und bei nichtvisuellen Objekten ist dies auch gar nicht möglich. Mit mehreren weiteren Methoden können Sie Objektinstanzen erstellen, indem Sie ausschließlich ActionScript verwenden.

Bei mehreren ActionScript-Datentypen können Sie eine Instanz mit einem *Literalausdruck* erstellen. Dies ist ein Wert, der direkt in den ActionScript-Code geschrieben wird. Im Folgenden sind einige Beispiele für diese Situationen aufgeführt:

- Numerischer Literalwert (geben Sie die Zahl direkt ein):

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

- Literaler String-Wert (geben Sie den Text zwischen doppelten Anführungszeichen ein):

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

- Literaler Boolean-Wert (verwenden Sie die Literalwerte `true` oder `false`):

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

- Literaler Array-Wert (setzen Sie eine kommagetrennte Werteliste in eckige Klammern):

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

- XML-Literalwert (geben Sie die XML-Daten direkt ein):

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

Darüber hinaus definiert ActionScript auch literale Ausdrücke für die Datentypen Array, RegExp, Object und Function.

Die üblichste Methode zum Erstellen einer Instanz für einen jeden Datentyp ist die Verwendung des `new`-Operators mit dem Klassennamen, wie im Folgenden gezeigt:

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

Die Erstellung eines Objekts mithilfe des `new`-Operators wird häufig als „Aufrufen des Klassenkonstruktors“ bezeichnet. Ein *Konstruktor* ist eine besondere Methode, die als Teil des Prozesses zum Erstellen einer Klasseninstanz aufgerufen wird. Wenn Sie eine Instanz auf diese Weise erstellen, müssen Sie nach dem Klassennamen Klammern einfügen. In einigen Fällen geben Sie in den Klammern Parameterwerte an. Diese beiden Schritte führen Sie auch beim Aufrufen einer Methode aus.

Den `new`-Operator können Sie auch bei den Datentypen, deren Instanzen mit einem Literalausdruck erstellt werden, zum Erstellen einer Objektinstanz verwenden. So haben die beiden folgenden Codezeilen dieselbe Funktionsweise:

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

Beim Erstellen von Objekten müssen Sie auf jeden Fall mit der Syntax `new` *ClassName*`()` vertraut sein. Viele ActionScript-Datentypen haben keine visuelle Darstellung. Sie können diese Datentypen deshalb nicht erstellen, indem Sie ein Element in Flash Professional auf der Bühne oder im Entwurfsmodus von Flash Builder im MXML-Editor platzieren. Instanzen dieser Datentypen können nur in ActionScript mithilfe des `new`-Operators erstellt werden.

Adobe Flash Professional
In Flash Professional können Sie den `new`-Operator auch verwenden, um eine Instanz eines Movieclip-Symbols zu erstellen, das zwar in der Bibliothek definiert ist, aber nicht auf der Bühne platziert wurde.

Verwandte Themen

Verwenden von Arrays

Verwenden von regulären Ausdrücken

Erstellen von MovieClip-Objekten mit ActionScript

Allgemeine Programmelemente

Zum Erstellen eines ActionScript-Programms stehen einige weitere Bausteine zur Verfügung.

Operatoren

Operatoren sind besondere Symbole (gelegentlich auch Wörter), die zum Durchführen von Berechnungen verwendet werden. Sie werden häufig für mathematische Operationen verwendet und dienen auch zum Vergleichen von Werten. Im Allgemeinen „berechnet“ ein Operator anhand von einem oder mehreren Werten ein Ergebnis. Zum Beispiel:

- Der Additionsoperator (`+`) addiert zwei Werte und gibt als Ergebnis eine Zahl zurück:

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

- Der Multiplikationsoperator (`*`) multipliziert zwei Werte und gibt als Ergebnis eine Zahl zurück:

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

- Der Gleichheitsoperator (`==`) vergleicht zwei Werte, um festzustellen, ob sie gleich sind, und gibt als Ergebnis einen booleschen Wert (true oder false) zurück:

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

Wie im obigen Beispiel werden der Gleichheitsoperator und andere „Vergleichs"-Operatoren häufig zusammen mit der `if`-Anweisung verwendet, um zu bestimmen, ob bestimmte Befehle ausgeführt werden sollen oder nicht.

Kommentare

Beim Schreiben von ActionScript ist es häufig sinnvoll, den Code mit Anmerkungen zu versehen. Beispielsweise können Sie so erläutern, wie bestimmte Codezeilen funktionieren oder warum Sie eine bestimmte Wahl getroffen haben. *Codekommentare* ermöglichen Ihnen die Eingabe von Text, der bei der Ausführung des Codes ignoriert wird. ActionScript kennt zwei Arten von Kommentaren:

- Einzeiliger Kommentar: Ein einzeiliger Kommentar wird an einer beliebigen Stelle einer Zeile hinter zwei Schrägstrichen eingefügt. Der gesamte Text ab den Schrägstrichen bis zum Ende der Zeile wird ignoriert:

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

- Mehrzeilige Kommentare: Ein mehrzeiliger Kommentar beginnt mit einer Markierung für den Kommentaranfang (`/*`), dann folgt der eigentliche Kommentar und schließlich eine Markierung für das Kommentarende (`*/`). Der gesamte Inhalt zwischen den Anfangs- und Endmarkierungen wird ignoriert, unabhängig davon, über wie viele Zeilen der Kommentar sich erstreckt:

  ```
  /*
  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.
  */
  ```

Häufig dienen Kommentare auch dazu, eine oder mehrere Codezeilen vorübergehend zu deaktivieren. Beispielsweise können Sie mithilfe von Kommentaren verschiedene Vorgehensweisen testen. Außerdem lässt sich mit Kommentaren möglicherweise ermitteln, warum ein bestimmter ActionScript-Code nicht wie vorgesehen funktioniert.

Ablaufsteuerung

Häufig sollen in einem Programm bestimmte Aktionen wiederholt werden, nur bestimmte Aktionen ausgeführt werden oder abhängig von bestimmten Bedingungen alternative Aktionen ausgeführt werden usw. Mit der *Ablaufsteuerung* können Sie festlegen, welche Aktionen ausgeführt werden. In ActionScript stehen Ihnen verschiedene Elemente zur Ablaufsteuerung zur Verfügung:

- Funktionen: Funktionen ähneln Kurzbefehlen. Sie ermöglichen das Gruppieren mehrerer Aktionen unter einem Namen und können zum Durchführen von Berechnungen verwendet werden. Funktionen sind bei der Verarbeitung von Ereignissen von Bedeutung, werden aber auch als allgemeines Werkzeug zum Gruppieren mehrerer Befehle verwendet.
- Schleifen: Mit Schleifenstrukturen können Sie einen Befehlssatz erstellen, den der Computer bis zu einem vorgegebenen Wert wiederholt oder bis sich eine bestimmte Bedingung ändert. Häufig werden Schleifen zum Ändern von mehreren verwandten Objekten eingesetzt. Dabei wird eine Variable verwendet, deren Wert sich jedes Mal ändert, wenn der Computer die Schleife abgearbeitet hat.

- Bedingungsanweisungen: Mithilfe von Bedingungsanweisungen können Befehle festgelegt werden, die nur unter bestimmten Bedingungen ausgeführt werden. Sie ermöglichen auch die Angabe von Alternativanweisungen für unterschiedliche Bedingungen. Die am häufigsten verwendete Bedingungsanweisung ist die `if`-Anweisung. Die `if`-Anweisung prüft einen Wert oder einen Ausdruck in ihren runden Klammern. Wenn der Wert `true` ist, werden die in geschweiften Klammern eingeschlossenen Codezeilen ausgeführt. Andernfalls werden diese Codezeilen ignoriert. Zum Beispiel:

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

Mit dem Begleiter der `if`-Anweisung, der `else`-Anweisung, können Sie alternative Befehle angeben, die ausgeführt werden, wenn die Bedingung nicht als `true` ausgewertet wird:

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

Beispiel: Animationen-Mustermappe (Flash Professional)

Dieses Beispiel soll Ihnen einen ersten Eindruck vermitteln, wie einzelne ActionScript-Codefragmente zu einer vollständigen Anwendung zusammengefügt werden können. Die Animationen-Mustermappe dient als Beispiel, um zu veranschaulichen, wie Sie einer vorhandenen linearen Animation einige kleinere interaktive Elemente hinzufügen können. Beispielsweise können Sie eine Animation, die für einen Kunden erstellt wurde, in eine Online-Mustermappe einbinden. Das interaktive Verhalten, das der Animation hinzugefügt werden soll, wird mit zwei Schaltflächen realisiert, auf die Benutzer klicken können: eine zum Starten der Animation und eine zum Navigieren zu einer separaten URL (z. B. zum Menü der Mustermappe oder zur Homepage des Autors).

Das Erstellen dieser Animation kann in die folgenden Hauptabschnitte unterteilt werden:

1. Vorbereiten der FLA-Datei für das Einfügen von ActionScript und interaktiven Elementen
2. Erstellen und Hinzufügen der Schaltflächen
3. Programmieren des ActionScript-Codes
4. Testen der Anwendung.

Vorbereiten für das Hinzufügen interaktiver Elemente

Bevor der Animation interaktive Elemente hinzugefügt werden können, sollte die FLA-Datei entsprechend vorbereitet werden, indem Platz zum Hinzufügen der zusätzlichen Inhalte geschaffen wird. Dazu muss Platz auf der Bühne geschaffen werden, damit die Schaltflächen platziert werden können. Weiterhin muss „Platz“ in der FLA-Datei verfügbar gemacht werden, damit unterschiedliche Elemente separat aufbewahrt werden können.

So richten Sie die FLA-Datei für das Hinzufügen interaktiver Elemente ein:

1. Erstellen Sie eine FLA-Datei mit einer einfachen Animation, wie ein einzelnes Bewegungs-Tween oder Form-Tween. Wenn bereits eine FLA-Datei mit der Animation vorhanden ist, die im Projekt gezeigt werden soll, öffnen Sie diese Datei und speichern Sie sie unter einem neuen Namen.
2. Bestimmen Sie, an welcher Stelle auf dem Bildschirm die beiden Schaltflächen angezeigt werden sollen. Eine Schaltfläche dient zum Starten der Animation, die andere ist ein Hyperlink zur Mustermappe oder Homepage des Autors. Schaffen Sie hierzu bei Bedarf auf der Bühne etwas Platz für diese neuen Inhalte. Wenn die Animation noch keinen Startbildschirm enthält, können Sie diesen im ersten Bild erstellen. In diesem Fall ist es sinnvoll, die Animation so zu verschieben, dass sie erst bei Bild 2 oder einem späteren Bild beginnt.
3. Fügen Sie über den anderen Ebenen in der Zeitleiste eine neue Ebene hinzu und geben Sie ihr den Namen **buttons**. Auf dieser Ebene fügen Sie die Schaltflächen hinzu.
4. Fügen Sie über der Ebene „buttons“ eine neue Ebene ein und nennen Sie sie **actions**. Auf dieser Ebene wird der Anwendung ActionScript-Code hinzugefügt.

Erstellen und Hinzufügen von Schaltflächen

Im nächsten Schritt erstellen und positionieren Sie die Schaltflächen, die das Kernstück der interaktiven Anwendung bilden.

So erstellen Sie Schaltflächen und fügen sie der FLA-Datei hinzu:

1. Erstellen Sie auf der Ebene „buttons“ mithilfe der Zeichenwerkzeuge die grafische Darstellung der ersten Schaltfläche („Wiedergabe“). Zeichnen Sie beispielsweise ein horizontales Oval und platzieren Sie darauf Text.
2. Markieren Sie mit dem Auswahlwerkzeug alle Grafikelemente der einzelnen Schaltfläche.
3. Wählen Sie im Hauptmenü die Optionen „Modifizieren“ > „In Symbol konvertieren“ aus.
4. Wählen Sie im angezeigten Dialogfeld als Symboltyp „Button“ aus, weisen Sie dem Symbol einen Namen zu und klicken Sie auf „OK“.
5. Weisen Sie der markierten Schaltfläche im Eigenschafteninspektor den Instanznamen **playButton** zu.
6. Wiederholen Sie die Schritte 1 bis 5, um die Schaltfläche zu erstellen, mit der Benutzer zur Homepage des Autors weitergeleitet werden. Geben Sie dieser Schaltfläche den Namen **homeButton**.

Programmieren des Codes

Der ActionScript-Code für diese Anwendung kann funktionell in drei Abschnitte unterteilt werden, die jedoch alle an derselben Stelle eingefügt werden. Der Code führt die drei folgenden Aktionen aus:

- Anhalten des Abspielkopfs, sobald die SWF-Datei geladen wird (beim Erreichen von Bild 1)
- Überwachen eines Ereignisses (das Klicken des Benutzers auf die Wiedergabeschaltfläche), um daraufhin die Wiedergabe der SWF-Datei zu starten
- Überwachen eines Ereignisses (das Klicken des Benutzers auf die Homepageschaltfläche), um daraufhin im Browser die entsprechende URL zu öffnen

So erstellen Sie Code, um den Abspielkopf beim Erreichen von Bild 1 anzuhalten:

1. Wählen Sie das Schlüsselbild in Bild 1 der Ebene „actions“ aus.
2. Wählen Sie im Hauptmenü die Optionen „Fenster“ > „Aktionen“ aus, um das Bedienfeld „Aktionen“ zu öffnen.
3. Geben Sie im Bedienfeld „Skript“ den folgenden Code ein:

```
stop();
```

So erstellen Sie Code zum Starten der Animation beim Klicken auf die Wiedergabeschaltfläche:

1 Fügen Sie am Ende des in den vorangegangenen Schritten eingegebenen Codes zwei leere Zeilen ein.

2 Geben Sie am Ende des Skripts den folgenden Code ein:

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

Mit diesem Code wird die Funktion `startMovie()` definiert. Beim Aufrufen von `startMovie()` wird die Wiedergabe der Hauptzeitleiste gestartet.

3 Geben Sie in der Zeile nach dem im vorangegangenen Schritt eingefügten Code die folgende Codezeile ein:

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

Mit dieser Codezeile wird die `startMovie()`-Funktion als Listener für das `click`-Ereignis von `playButton` registriert. Das bedeutet, dass bei jedem Klicken auf die Schaltfläche `playButton` die `startMovie()`-Funktion aufgerufen wird.

So erstellen Sie den Code zum Öffnen einer URL im Browser beim Klicken auf die Homepageschaltfläche:

1 Fügen Sie am Ende des in den vorangegangenen Schritten eingegebenen Codes zwei leere Zeilen ein.

2 Geben Sie am Ende des Skripts den folgenden Code ein:

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

Mit diesem Code wird die Funktion `gotoAuthorPage()` definiert. Diese Funktion erstellt zunächst eine URLRequest-Instanz, die die URL http://example.com/ darstellt. Anschließend übergibt sie diese URL an die Funktion `navigateToURL()`, wodurch diese URL im Browser des Benutzers geöffnet wird.

3 Geben Sie in der Zeile nach dem im vorangegangenen Schritt eingefügten Code die folgende Codezeile ein:

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

Mit dieser Codezeile wird die `gotoAuthorPage()`-Funktion als Listener für das `click`-Ereignis von `homeButton` registriert. Das bedeutet, dass bei jedem Klicken auf die Schaltfläche `homeButton` die `gotoAuthorPage()`-Funktion aufgerufen wird.

Testen der Anwendung

Die Anwendung ist nun voll funktionsfähig. Testen Sie sie nun, um sich davon zu überzeugen.

So testen Sie die Anwendung:

1 Wählen Sie im Hauptmenü die Optionen „Steuerung“ > „Film testen“ aus. Flash Professional erstellt die SWF-Datei und öffnet sie in einem Flash Player-Fenster.

2 Klicken Sie auf beide Schaltflächen, um sich zu vergewissern, dass sie wie erwartet funktionieren.

3 Wenn dies nicht der Fall sein sollte, überprüfen Sie folgende Punkte:

- Ist beiden Schaltflächen jeweils ein eigener Instanzname zugeordnet?

- Werden diese Namen in den Aufrufen der `addEventListener()`-Methode als Instanznamen der Schaltflächen verwendet?
- Werden in den Aufrufen der `addEventListener()`-Methode die richtigen Ereignisnamen verwendet?
- Ist für jede der Funktionen der korrekte Parameter angegeben? (Beide Methoden erfordern einen einzelnen Parameter mit dem MouseEvent-Datentyp.)

Bei diesen und den meisten anderen Fehlern wird eine Fehlermeldung angezeigt. Die Fehlermeldung wird entweder angezeigt, wenn Sie den Befehl „Film testen“ wählen, oder wenn Sie beim Testen des Projekts auf die Schaltfläche klicken. Die Kompilierfehler werden im entsprechenden Bedienfeld aufgelistet (dies sind die Fehler, die auftreten, wenn Sie zum ersten Mal „Film testen“ auswählen). Laufzeitfehler werden im Bedienfeld „Ausgabe“ angezeigt. Diese Fehler treten bei der Wiedergabe des Inhalts auf, beispielsweise wenn Sie auf eine Schaltfläche klicken.

Erstellen von Anwendungen mit ActionScript

Zum Erstellen von Anwendungen mit ActionScript müssen Sie mehr als nur die Syntax und die Namen der Klassen kennen, die Sie verwenden möchten. Der Großteil der Dokumentation für die Flash-Plattform befasst sich mit diesen beiden Themen (Syntax und Verwendung der ActionScript-Klassen). Zur Erstellung einer ActionScript-Anwendung benötigen Sie jedoch noch weitere Informationen, wie zum Beispiel:

- Welche Programme können zum Schreiben von ActionScript verwendet werden?
- Wie lässt sich ActionScript-Code organisieren?
- Wie kann ActionScript-Code in eine Anwendung aufgenommen werden?
- Welche Schritte müssen bei der Erstellung einer ActionScript-Anwendung ausgeführt werden?

Optionen für die Codestrukturierung

Mit ActionScript 3.0 können Sie alle Arten von Anwendungen entwickeln, von einfachen Grafikanimationen bis hin zu komplexen Verarbeitungssystemen für Client-Server-Transaktionen. Je nach der von Ihnen erstellten Anwendung verwenden Sie eines oder mehrere der folgenden Verfahren, um ActionScript in Ihr Projekt aufzunehmen.

Speichern von Code in Bildern einer Flash Professional-Zeitleiste
In Flash Professional können Sie jedem Bild in einer Zeitleiste ActionScript-Code hinzufügen. Dieser Code wird ausgeführt, wenn der Film abgespielt wird und der Abspielkopf dieses Bild erreicht.

Durch das Platzieren von ActionScript-Code in Bildern können Sie den in Flash Professional erstellten Anwendungen ganz einfach ein bestimmtes Verhalten hinzufügen. Sie können jedem Bild in der Hauptzeitleiste oder jedem Bild in der Zeitleiste eines beliebigen Movieclip-Symbols Code hinzufügen. Diese Flexibilität hat jedoch ihren Preis. Wenn Sie größere Anwendungen erstellen, verlieren Sie leicht die Übersicht, welche Skripts in welchen Bildern enthalten sind. Als Folge der komplizierten Struktur gestaltet sich die Verwaltung der Anwendung mit der Zeit wahrscheinlich immer schwieriger.

Viele Entwickler vereinfachen die Struktur des ActionScript-Codes in Flash Professional, indem sie Code nur im ersten Bild einer Zeitleiste oder nur auf einer bestimmten Ebene in ein Flash-Dokument einfügen. Durch die Trennung kann der Code in den Flash-FLA-Dateien einfacher gefunden und gepflegt werden. Soll derselbe Code in einem anderen Flash Professional-Projekt verwendet werden, muss er jedoch kopiert und in die neue Datei eingefügt werden.

Damit Ihr ActionScript-Code in Zukunft einfacher in anderen Flash Professional-Projekten verwendet werden kann, sollten Sie ihn in externen ActionScript-Dateien speichern (Textdateien mit der Erweiterung „.as“).

Einbetten von Code in Flex MXML-Dateien

In einer Flex-Entwicklungsumgebung wie Flash Builder können Sie ActionScript-Code innerhalb eines `<fx:Script>`-Tags in einer Flex MXML-Datei einschließen. Durch diese Technik können große Projekte jedoch an Komplexität zunehmen. Außerdem kann es schwieriger werden, denselben Code in anderen Flex-Projekten zu verwenden. Damit Ihr ActionScript-Code in Zukunft einfacher in anderen Flex-Projekten verwendet werden kann, sollten Sie den Code in externen ActionScript-Dateien speichern.

Hinweis: *Sie können einen Quellparameter für ein `<fx:Script>`-Tag angeben. Die Verwendung eines Quellparameters ermöglicht es Ihnen, ActionScript-Code so aus einer externen Datei zu „importieren“, als ob er direkt im `<fx:Script>`-Tag eingegeben worden wäre. Jedoch kann die von Ihnen verwendete Quelldatei keine eigene Klasse definieren, wodurch ihre Wiederverwendbarkeit eingeschränkt ist.*

Speichern von Code in ActionScript-Dateien

Wenn Ihr Projekt viel ActionScript-Code umfasst, sollten Sie Ihren Code am besten in separaten ActionScript-Quelldateien speichern (Textdateien mit der Erweiterung „.as“). Eine ActionScript-Datei kann auf zwei Arten strukturiert werden, je nachdem, wie Sie den Code in Ihrer Anwendung verwenden möchten.

- Unstrukturierter ActionScript-Code: ActionScript-Codezeilen, einschließlich Anweisungen oder Funktionsdefinitionen, werden so geschrieben, als ob sie direkt in das Skript einer Zeitleiste oder in eine MXML-Datei eingegeben werden.

  Auf diese Art programmierter ActionScript-Code kann in ActionScript mit der `include`-Anweisung oder in Flex MXML mit dem `<fx:Script>`-Tag aufgerufen werden. Die ActionScript-Anweisung `include` weist den Compiler an, den Inhalt einer externen ActionScript-Datei an einer bestimmten Position und in einem bestimmten Umfang in ein Skript aufzunehmen. Dies führt zu demselben Ergebnis wie bei direkter Eingabe des Codes. In der MXML-Sprache wird durch Verwendung des `<fx:Script>`-Tags mit einem Quellattribut eine externe ActionScript-Datei identifiziert, die der Compiler an diesem Punkt in der Anwendung lädt. So lädt das folgende Tag eine externe ActionScript-Datei namens „Box.as“:

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

- ActionScript-Klassendefinition: Die Definition einer ActionScript-Klasse, einschließlich ihrer Methoden und Eigenschaftendefinitionen.

  Beim Definieren einer Klasse können Sie auf den ActionScript-Code in der Klasse zugreifen, indem Sie eine Instanz der Klasse erstellen und ihre Eigenschaften, Methoden und Ereignisse verwenden. Die Verwendung von eigenen Klassen unterscheidet sich nicht von der Verwendung der integrierten ActionScript-Klassen. Zwei Teile sind hierzu erforderlich:

  - Verwenden Sie die `import`-Anweisung, um den vollständigen Namen der Klasse anzugeben, sodass der ActionScript-Compiler weiß, wo er die Klasse findet. Wenn Sie beispielsweise die MovieClip-Klasse in ActionScript verwenden möchten, importieren Sie die Klasse mit ihrem vollständigen Namen, einschließlich Paket und Klasse:

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

    Alternativ können Sie das Paket importieren, in dem die MovieClip-Klasse enthalten ist. Dies entspricht dem Schreiben von separaten `import`-Anweisungen für jede Klasse in dem Paket:

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

    Die Klassen der oberen Ebene bilden die einzige Ausnahme zu der Regel, dass eine Klasse importiert werden muss, damit sie im Code verwendet werden kann. Diese Klassen sind nicht in einem Paket definiert.

- Schreiben Sie Code, der konkret den Klassennamen verwendet. Deklarieren Sie beispielsweise eine Variable mit dieser Klasse als Datentyp und erstellen Sie eine Instanz der Klasse zum Speichern in der Variablen. Durch die Verwendung einer Klasse im ActionScript-Code weisen Sie den Compiler an, die Definition dieser Klasse zu laden. Wenn beispielsweise eine externe Klasse namens „Box“ vorhanden ist, erstellt diese Anweisung eine Instanz der Box-Klasse:

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

  Wenn der Compiler zum ersten Mal einen Verweis auf die Box-Klasse erkennt, sucht er im verfügbaren Quellcode nach der Box-Klassendefinition.

Auswählen des richtigen Tools

Zum Schreiben und Bearbeiten von ActionScript-Code stehen mehrere Tools zur Verfügung, die auch in Kombination eingesetzt werden können.

Flash Builder

Adobe Flash Builder ist das bevorzugte Tool zum Erstellen von Projekten in der Flex-Umgebung oder von Projekten, die hauptsächlich aus ActionScript-Code bestehen. Flash Builder umfasst außer einem ActionScript-Editor mit vollem Funktionsumfang auch Funktionen für das visuelle Layout sowie für die MXML-Bearbeitung. Er eignet sich zum Erstellen von Flex-Projekten oder für Projekte, die ausschließlich aus ActionScript bestehen. Flex bietet verschiedene Vorteile, beispielsweise einen umfangreichen Satz an vordefinierten UI-Steuerelementen, flexible dynamische Layoutsteuerungen und integrierte Mechanismen zum Arbeiten mit externen Daten und zum Verknüpfen von externen Daten mit Elementen der Benutzeroberfläche. Da für diese Funktionen jedoch zusätzlicher Code erforderlich ist, sind die SWF-Dateien bei Flex-Projekten meist größer als SWF-Dateien, die ohne Flex erstellt werden.

Verwenden Sie Flash Builder zum Erstellen von funktionsreichen, datenorientierten Internetanwendungen mit Flex. Verwenden Sie es auch zum Bearbeiten von ActionScript- oder MXML-Code sowie für das visuelle Anwendungslayout mit einem einzigen Tool.

Zahlreiche Flash Professional-Benutzer verwenden bei der Entwicklung von ActionScript-lastigen Projekten Flash Professional zur Erstellung der visuellen Elemente und Flash Builder als Editor für den ActionScript-Code.

Flash Professional

Flash Professional bietet nicht nur Tools zur Erstellung von Grafiken und Animationen, sondern auch für die Arbeit mit ActionScript-Code. Der Code kann an Elemente angefügt sein, die sich entweder in FLA-Dateien oder in externen Dateien befinden, die ausschließlich aus ActionScript bestehen. Flash Professional eignet sich ideal für Projekte, die in hohem Maße von Animationen oder Videos Gebrauch machen. Es ist besonders dann von Nutzen, wenn Sie die meisten grafischen Elemente selbst erstellen möchten. Die Erstellung von ActionScript-Projekten mithilfe von Flash Professional bietet außerdem den Vorteil, dass Sie in derselben Anwendung visuelle Elemente erstellen und Code schreiben können. Flash Professional umfasst auch vordefinierte Benutzeroberflächenkomponenten. Mithilfe dieser Komponenten können Sie kleinere SWF-Dateien erzielen. Außerdem haben Sie die Möglichkeit, mit visuellen Werkzeugen Skins für Ihre Projekte anzuwenden.

Flash Professional umfasst zwei Tools für das Schreiben von ActionScript-Code:

- Bedienfeld „Aktionen“: Dieses Bedienfeld steht Ihnen beim Arbeiten in einer FLA-Datei zur Verfügung. Hier können Sie einen ActionScript-Code schreiben, der an die Bilder in einer Zeitleiste angehängt wird.
- Skriptfenster: Das Skriptfenster ist ein spezieller Texteditor für das Arbeiten mit ActionScript-Codedateien (.as-Dateien).

ActionScript-Editoren von Drittanbietern

Da ActionScript-Dateien (.as-Dateien) als einfache Textdateien gespeichert werden, können Sie jedes Programm, das einfache Textdateien bearbeiten kann, zum Schreiben von ActionScript-Dateien verwenden. Neben den ActionScript-Produkten von Adobe gibt es verschiedene Textverarbeitungsprogramme von Drittanbietern, mit denen ActionScript-Code geschrieben werden kann. Sie können MXML-Dateien oder ActionScript-Klassen mit einem beliebigen Texteditor erstellen. Anschließend können Sie mithilfe des Flex SDK anhand dieser Dateien Anwendungen erstellen. Das Projekt kann Flex verwenden oder es kann sich um eine Anwendung handeln, die ausschließlich aus ActionScript besteht. Wieder andere Entwickler verwenden Flash Builder oder den ActionScript-Editor eines Drittanbieters zum Erstellen von ActionScript-Klassen in Kombination mit Flash Professional zum Erstellen der grafischen Inhalte.

Gründe für die Wahl eines ActionScript-Editors eines Drittanbieters:

- Sie möchten ActionScript-Code in einem separaten Programm schreiben und die visuellen Elemente in Flash Professional erstellen.
- Sie verwenden eine bestimmte Anwendung für das Programmieren in einer anderen Sprache als ActionScript (z. B. für das Erstellen von HTML-Seiten oder das Entwerfen von Anwendungen in einer anderen Programmiersprache) und möchten diese Anwendung auch für die ActionScript-Programmierung verwenden.
- Sie möchten mit dem Flex SDK reine ActionScript-Projekte oder Flex-Projekte ohne Flash Professional oder Flash Builder erstellen.

Einige der bekanntesten Code-Editoren, die eine Unterstützung für ActionScript anbieten, sind:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (mit ActionScript- und Flex-Programmpaketen)

ActionScript-Entwicklungsprozess

Unabhängig von der Größe Ihres ActionScript-Projekts empfiehlt es sich, einen Prozess zum Skizzieren und Entwickeln der Anwendung einzurichten, um Ihre Arbeit effizienter zu gestalten. Die folgenden Schritte beschreiben den grundlegenden Entwicklungsprozess beim Erstellen einer Anwendung mit ActionScript 3.0:

1. Skizzieren der Anwendung.

   Skizzieren Sie die grundlegende Idee Ihrer Anwendung, bevor Sie mit der Entwicklung beginnen.

2. Erstellen des ActionScript 3.0-Codes.

   Sie können den ActionScript-Code mithilfe von Flash Professional, Flash Builder, Dreamweaver oder einem Texteditor erstellen.

3. Erstellen eines Flash- oder Flex-Projekts zum Ausführen des Codes.

   In Flash Professional: Erstellen Sie eine FLA-Datei, richten Sie die Einstellungen für die Veröffentlichung ein, fügen Sie der Anwendung Benutzeroberflächenkomponenten hinzu und verweisen Sie auf den ActionScript-Code. In Flex: Definieren Sie die Anwendung, fügen Sie mit MXML Benutzeroberflächenkomponenten hinzu und verweisen Sie auf den ActionScript-Code.

4 Veröffentlichen und Testen der ActionScript-Anwendung.

   Zum Testen führen Sie die Anwendung innerhalb der Entwicklungsumgebung aus, wobei Sie sicherstellen, dass die Anwendung wie vorgesehen funktioniert.

Sie müssen diese Schritte weder in der vorgegebenen Reihenfolge ausführen noch einen Schritt vollständig abschließen, bevor Sie mit dem nächsten Schritt beginnen. Beispielsweise können Sie einen Bildschirm der Anwendung entwerfen (Schritt 1) und dann die Grafiken, Schaltflächen usw. erstellen (Schritt 3), bevor Sie ActionScript-Code schreiben (Schritt 2) und die Anwendung testen (Schritt 4). Oder Sie entwerfen einen Teil der Anwendung, fügen dann eine Schaltfläche oder ein anderes Steuerelement ein, schreiben den ActionScript-Code für die Schaltfläche bzw. das Steuerelement und testen die Funktionsweise direkt nach der Erstellung. Sie sollten sich die vier Schritte des Entwicklungsprozesses merken. Unter echten Arbeitsbedingungen ist es jedoch meist sinnvoller, nach Bedarf zwischen den einzelnen Schritten zu wechseln.

Erstellen eigener Klassen

Das Erstellen von Klassen für Ihre Projekte scheint eine große Herausforderung zu sein. Der schwierigste Teil beim Erstellen der Klasse ist jedoch eher die Strukturierung ihrer Methoden, Eigenschaften und Ereignisse.

Strategien beim Strukturieren einer Klasse

Der objektorientierte Programmentwurf ist ein komplexes Thema. Ganze Karrieren wurden dem akademischen Studium und der professionellen Anwendung dieser Disziplin gewidmet. Dennoch sind an dieser Stelle einige empfohlene Vorgehensweisen aufgeführt, die Ihnen die ersten Schritte erleichtern sollen.

1 Denken Sie an die Rolle, die Instanzen dieser Klasse in der Anwendung spielen. Im Allgemeinen übernehmen Objekte eine der folgenden drei Funktionen:

   - Wertobjekt: Diese Objekte dienen in erster Linie als Datencontainer. Meist haben sie mehrere Eigenschaften, aber nur wenige Methoden (oder gar keine Methoden). Es handelt sich bei diesen Objekten meist um Codedarstellungen von klar definierten Elementen. Beispielsweise könnte eine Musik-Player-Anwendung eine Song-Klasse für einzelne Musiktitel und eine Playlist-Klasse für eine Musiktitelgruppe enthalten.
   - Anzeigeobjekt: Hierbei handelt es sich um Objekte, die auf dem Bildschirm angezeigt werden. Dies sind z. B. Steuerelemente in einer Benutzeroberfläche wie eine Dropdownliste, eine Statusanzeige oder grafische Elemente wie die Figuren in einem Videospiel.
   - Anwendungsstruktur: Diese Objekte sind vielfach in unterstützenden Funktionen in der Logik oder Verarbeitung von Anwendungen zu finden. Beispielsweise können Sie festlegen, dass ein Objekt in einer biologischen Simulation bestimmte Berechnungen vornimmt. Oder Sie erstellen ein Objekt, dessen Aufgabe es ist, die Werte zwischen einem Regler und einer Lautstärkeanzeige in einer Musik-Player-Anwendung zu synchronisieren. Ein anderes Objekt kann für die Verwaltung der Regeln in einem Videospiel zuständig sein. Ein weiteres Beispiel wäre eine Klasse, die ein gespeichertes Bild in einer Zeichnungsanwendung lädt.

2 Überlegen Sie, über welche Funktionalität die Klasse verfügen muss. Diese verschiedenen Funktionstypen werden häufig zu den Methoden der Klasse.

3 Soll die Klasse als Wertobjekt verwendet werden, überlegen Sie, welche Daten die Klasseninstanzen aufnehmen werden. Diese Objekte sind gute Kandidaten für Eigenschaften.

4 Da Ihre Klasse speziell für Ihr Projekt entwickelt wurde, ist es am wichtigsten, dass Sie die Funktionen bereitstellen, die für Ihre Anwendung erforderlich sind. Überlegen Sie sich Antworten auf die folgenden Fragen:

- Welche Informationen werden in Ihrer Anwendung gespeichert, verfolgt und geändert? Durch die Beantwortung dieser Frage können Sie die erforderlichen Wertobjekte und Eigenschaften bestimmen.
- Welche Aktionen führt die Anwendung aus? Was passiert zum Beispiel, wenn die Anwendung zum ersten Mal geladen wird, wenn der Benutzer auf eine bestimmte Schaltfläche klickt oder wenn die Wiedergabe eines Films angehalten wird? Dies sind gute Kandidaten für Methoden. Es kann sich auch um Eigenschaften handeln, wenn die „Aktionen" eine Änderung einzelner Werte nach sich ziehen.
- Welche Informationen sind zur Ausführung der einzelnen Aktionen erforderlich? Diese Informationen werden die Parameter der Methode.
- Was ändert sich in Ihrer Klasse, über das andere Teile der Anwendung informiert sein müssen, während die Anwendung ausgeführt wird? Dies sind gute Kandidaten für Ereignisse.

5 Ist ein Objekt vorhanden, das dem benötigten Objekt ähnelt, aber nicht die gesamte erforderliche Funktionalität bereitstellt? In diesem Fall könnte die Erstellung einer Unterklasse sinnvoll sein. (Eine *Unterklasse* bedient sich der Funktionalität einer vorhandenen Klasse, anstatt ihre eigene Funktionalität komplett selbst zu definieren.) Wenn Sie beispielsweise eine Klasse für ein visuelles Objekt auf dem Bildschirm erstellen möchten, verwenden Sie das Verhalten eines vorhandenen Anzeigeobjekts als Grundlage für die Klasse. In diesem Fall wäre das Anzeigeobjekt (wie MovieClip oder Sprite) die *Basisklasse*, die von Ihrer Klasse erweitert wird.

Schreiben des Codes für eine Klasse

Nachdem Sie ein Konzept Ihrer Klasse erarbeitet haben oder zumindest in groben Zügen wissen, welche Informationen sie speichert und welche Aktionen sie ausführt, ist die tatsächliche Syntax zum Schreiben der Klasse relativ einfach.

Das Erstellen einer eigenen ActionScript-Klasse umfasst in jedem Fall die folgenden Schritte:

1 Öffnen Sie ein neues Textdokument in Ihrem ActionScript-Texteditor.

2 Geben Sie eine `class`-Anweisung ein, um den Namen der Klasse festzulegen. Zum Hinzufügen einer `class`-Anweisung geben Sie die Wörter `public class` und dann den Namen der Klasse ein. Fügen Sie ein geschweiftes Klammernpaar hinzu, das den Inhalt der Klasse einschließen wird (die Methoden- und Eigenschaftsdefinitionen). Zum Beispiel:

```
public class MyClass
{
}
```

`public` gibt an, dass aus beliebigem anderen Code auf die Klasse zugegriffen werden kann. Informationen zu anderen Alternativen finden Sie unter Namespace-Attribute zur Zugriffskontrolle.

3 Geben Sie eine `package`-Anweisung ein, um den Namen des Pakets anzugeben, das Ihre Klasse enthält. Verwenden Sie folgende Syntax: das Wort `package` gefolgt vom vollständigen Paketnamen, gefolgt von einem geschweiften Klammernpaar um den `class`-Anweisungsblock herum. Ändern Sie den Code aus dem vorherigen Schritt beispielsweise wie folgt:

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

4 Definieren Sie die Eigenschaften der Klasse mithilfe der `var`-Anweisung innerhalb des Klassenrumpfs. Die Syntax ist die gleiche, die Sie bereits zum Deklarieren der Variablen verwendet haben (abgesehen von dem hinzugefügten Modifizierer `public`). Wenn Sie beispielsweise die folgenden Zeilen zwischen der öffnenden und der schließenden geschweiften Klammer der Klassendefinition hinzufügen, werden Eigenschaften namens `textProperty`, `numericProperty` und `dateProperty` erstellt:

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

5 Definieren Sie die Methoden der Klasse mit der gleichen Syntax, die Sie zur Definition einer Funktion verwendet haben. Zum Beispiel:

- Geben Sie zum Erstellen einer `myMethod()`-Methode Folgendes ein:

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

- Zum Erstellen eines Konstruktors definieren Sie eine Methode, deren Name exakt dem Klassennamen entspricht. Ein Konstruktor ist eine spezielle Methode, die als Teil des Prozesses zum Erstellen einer Klasseninstanz aufgerufen wird.

```
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);
}
```

  Wenn Sie keine Konstruktormethode in Ihre Klasse aufnehmen, erstellt der Compiler automatisch einen leeren Konstruktor in Ihrer Klasse. (Dies ist ein Konstruktor ohne Parameter und Anweisungen.)

Sie können einige weitere Klassenelemente definieren. Diese Elemente sind komplexer.

- *Accessors* (Zugriffsmethoden) sind ein Mittelding zwischen Methode und Eigenschaft. Beim Schreiben des Codes zum Definieren der Klasse erstellen Sie den Accessor genauso wie eine Methode. Sie können mehrere Aktionen durchführen, anstatt nur einen Wert zu lesen oder zuzuweisen – mehr ist beim Definieren einer Eigenschaft nicht möglich. Wenn Sie jedoch eine Instanz Ihrer Klasse erstellen, behandeln Sie den Accessor wie eine Eigenschaft und verwenden den Namen, um einen Wert zu lesen oder zuzuweisen.
- Ereignisse werden in ActionScript nicht mit einer besonderen Syntax definiert. Stattdessen definieren Sie die Ereignisse in Ihrer Klasse mithilfe der Funktionalität der EventDispatcher-Klasse.

Verwandte Themen

Verarbeiten von Ereignissen

Beispiel: Erstellen einer einfachen Anwendung

ActionScript 3.0 kann in verschiedenen Umgebungen zur Anwendungsentwicklung eingesetzt werden, wie beispielsweise mit Flash Professional und Flash Builder sowie jedem Texteditor.

In diesem Beispiel werden Sie schrittweise durch das Erstellen und Erweitern einer einfachen ActionScript 3.0-Anwendung mit Flash Professional oder Flash Builder geführt. Die von Ihnen erstellte Anwendung ist ein einfaches Muster für das Verwenden von externen ActionScript 3.0-Klassendateien in Flash Professional und Flex.

Entwerfen der ActionScript-Anwendung

Da es sich bei dieser ActionScript-Beispielanwendung um eine standardmäßige Anwendung „Hello World“ handelt, ist ihre Struktur sehr einfach:

- Die Anwendung heißt „HelloWorld“.
- Sie zeigt ein einfaches Textfeld mit den Wörtern „Hello World!“ an.
- Die Anwendung nutzt eine einzelne objektorientierte Klasse namens „Greeter“. Wegen dieser Struktur kann die Anwendung in einem Flash Professional- oder Flex-Projekt verwendet werden.
- In diesem Beispiel erstellen Sie zunächst eine einfache Version der Anwendung. Dann fügen Sie Funktionalität hinzu, über die der Benutzer einen Benutzernamen eingeben kann, den die Anwendung mit einer Liste bekannter Benutzernamen vergleicht.

Mit dieser Kurzbeschreibung können Sie bereits mit dem Erstellen der Anwendung beginnen.

Erstellen des Projekts „HelloWorld“ und der Greeter-Klasse

Die Entwurfsanweisung für die Anwendung „HelloWorld“ besagt, dass der Code so geschrieben werden muss, dass er auch an anderer Stelle eingesetzt werden kann. Um dieses Ziel zu erreichen, nutzt die Anwendung eine einzelne objektorientierte Klasse namens „Greeter“. Sie verwenden diese Klasse innerhalb einer Anwendung, die Sie mit Flash Builder oder Flash Professional erstellen.

So erstellen Sie das HelloWorld-Projekt und die Greeter-Klasse in Flex:

1. Wählen Sie in Flash Builder „File“ > „New“ > „Flex Project“ (Datei > Neu > Flex-Projekt).
2. Geben Sie den Projektnamen „HelloWorld“ ein. Stellen Sie sicher, dass der Anwendungstyp auf „Web (runs in Adobe Flash Player)“ (wird in Adobe Flash Player ausgeführt) eingestellt ist. Klicken Sie dann auf „Finish“ (Fertig stellen).

   Flash Builder erstellt Ihr Projekt und zeigt es im Paket-Explorer an. Standardmäßig enthält das Projekt bereits eine Datei namens „HelloWorld.mxml“, die im Editor geöffnet ist.
3. Um nun in Flash Builder eine benutzerdefinierte ActionScript-Klassendatei zu erstellen, wählen Sie „File“ > „New“ > „ActionScript Class“ (Datei > Neu > ActionScript-Klasse).
4. Geben Sie im Dialogfeld „New ActionScript Class“ (Neue ActionScript-Klasse) **Greeter** als Klassennamen in das Feld „Name“ ein und klicken Sie auf „Finish“ (Fertig stellen).

   Ein neues Fenster zum Bearbeiten von ActionScript wird angezeigt.

   Fahren Sie mit dem Abschnitt Hinzufügen von Code zur Greeter-Klasse fort.

So erstellen Sie die Greeter-Klasse in Flash Professional:

1. Wählen Sie in Flash Professional „Datei“ > „Neu“ aus.
2. Wählen Sie im Dialogfeld „Neues Dokument“ die Option „ActionScript-Datei“ aus und klicken Sie dann auf „OK“.

   Ein neues Fenster zum Bearbeiten von ActionScript wird angezeigt.
3. Wählen Sie „Datei“ > „Speichern“. Wählen Sie den Ordner für die Anwendung aus, geben Sie der ActionScript-Datei den Namen **Greeter.as** und klicken Sie dann auf „OK“.

Fahren Sie mit dem Abschnitt Hinzufügen von Code zur Greeter-Klasse fort.

Hinzufügen von Code zur Greeter-Klasse

Die Greeter-Klasse definiert ein Objekt, `Greeter`, das Sie in Ihrer Anwendung „HelloWorld“ verwenden.

So fügen Sie der Greeter-Klasse Code hinzu:

1 Geben Sie den folgenden Code in die neue Datei ein (möglicherweise wurde ein Teil des Codes bereits hinzugefügt):

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

Die Greeter-Klasse enthält eine einzige `sayHello()`-Methode, die einen String zurückgibt, der die Worte „Hello World!“ enthält.

2 Wählen Sie im Menü „Datei“ den Befehl „Speichern“, um die ActionScript-Datei zu speichern.

Die Greeter-Klasse kann nun in einer Anwendung verwendet werden.

Erstellen einer Anwendung, die den ActionScript-Code verwendet

Die von Ihnen erstellte Greeter-Klasse definiert zwar einen eigenständigen Satz an Softwarefunktionen, stellt jedoch keine vollständige Anwendung dar. Um die Klasse zu verwenden, erstellen Sie ein Flash Professional-Dokument oder ein Flex-Projekt.

Der Code benötigt eine Instanz der Greeter-Klasse. Im Folgenden wird beschrieben, wie Sie die Greeter-Klasse in Ihrer Anwendung verwenden.

So erstellen Sie eine ActionScript-Anwendung mit Flash Professional:

1 Wählen Sie „Datei“ > „Neu“.

2 Wählen Sie im Dialogfeld „Neues Dokument“ die Option „Flash-Datei (ActionScript 3.0)“ aus und klicken Sie dann auf „OK“.

Ein neues Dokumentfenster wird angezeigt.

3 Wählen Sie „Datei“ > „Speichern“. Wählen Sie den Ordner aus, der bereits die Klassendatei „Greeter.as“ enthält, weisen Sie dem Flash-Dokument den Namen **HelloWorld.fla** zu und klicken Sie auf „OK“.

4 Wählen Sie in der Werkzeugpalette von Flash Professional das Textwerkzeug aus. Definieren Sie auf der Bühne durch Ziehen ein neues Textfeld mit einer Breite von ca. 300 Pixel und einer Höhe von ca. 100 Pixel.

5 Weisen Sie dem auf der Bühne ausgewählten Textfeld im Bedienfeld „Eigenschaften“ den Texttyp „Dynamischer Text“ zu. Geben Sie **mainText** als Instanznamen des Textfelds ein.

6 Klicken Sie auf das erste Bild der Hauptzeitleiste. Öffnen Sie das Bedienfeld „Aktionen“, indem Sie „Fenster“ >„Aktionen“ wählen.

7 Geben Sie im Bedienfeld „Aktionen“ das folgende Skript ein:

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

8 Speichern Sie die Datei.

Fahren Sie mit dem Abschnitt Veröffentlichen und Testen der ActionScript-Anwendung fort.

So erstellen Sie eine ActionScript-Anwendung mit Flash Builder:

1 Öffnen Sie die Datei „HelloWorld.mxml“ und fügen Sie Code hinzu, damit die Datei dem folgenden Beispiel entspricht:

```
<?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>
```

Dieses Flex-Projekt beinhaltet vier MXML-Tags:

- Ein `<s:Application>`-Tag, das den Anwendungscontainer definiert
- Ein `<s:layout>`-Tag, das den Layoutstil (vertikales Layout) für das Application-Tag definiert
- Ein `<fx:Script>`-Tag, das ActionScript-Code enthält
- Ein `<s:TextArea>`-Tag, das ein Feld zur Anzeige von Textmeldungen für den Benutzer definiert

Der Code im `<fx:Script>`-Tag definiert eine `initApp()`-Methode, die beim Laden der Anwendung aufgerufen wird. Die `initApp()`-Methode legt den Textwert des `mainTxt`-TextArea auf den String „Hello World!“ fest, der von der `sayHello()`-Methode der benutzerdefinierten Greeter-Klasse zurückgegeben wird, die Sie soeben geschrieben haben.

2 Wählen Sie „File“ > „Save“ (Datei > Speichern), um die Anwendung zu speichern.

Fahren Sie mit dem Abschnitt Veröffentlichen und Testen der ActionScript-Anwendung fort.

Veröffentlichen und Testen der ActionScript-Anwendung

Die Softwareentwicklung läuft immer nach dem gleichen Schema ab. Sie schreiben Code, versuchen, ihn zu kompilieren, und bearbeiten den Code, bis er fehlerfrei kompiliert wird. Die kompilierte Anwendung wird ausgeführt und getestet, um sicherzustellen, dass sie wie vorgesehen funktioniert. Ist dies nicht der Fall, bearbeiten Sie den Code, bis die gewünschte Funktionsweise erreicht ist. Die Entwicklungsumgebungen Flash Professional und Flash Builder bieten verschiedene Möglichkeiten zum Veröffentlichen, Testen und Debuggen Ihrer Anwendungen.

Im Folgenden sind einige der grundlegenden Schritte zum Testen der Anwendung „HelloWorld“ in beiden Umgebungen aufgeführt.

So veröffentlichen und testen Sie eine ActionScript-Anwendung mit Flash Professional:

1. Veröffentlichen Sie die Anwendung und achten Sie auf Kompilierungsfehler. Wählen Sie in Flash Professional die Optionen „Steuerung“ > „Film testen“ aus, um den ActionScript-Code zu kompilieren und die Anwendung „Hello World“ zu starten.
2. Wenn beim Testen der Anwendung Fehler oder Warnungen im Ausgabefenster angezeigt werden, beheben Sie diese Fehler in den Dateien „HelloWorld.fla“ oder „HelloWorld.as“. Testen Sie die Anwendung dann erneut.
3. Wenn keine Kompilierfehler vorliegen, wird ein Flash Player-Fenster mit der Anwendung „Hello World“ geöffnet.

Sie haben eine einfache, aber dennoch vollständige objektorientierte Anwendung erstellt, die ActionScript 3.0 verwendet. Fahren Sie mit dem Abschnitt Erweitern der Anwendung „HelloWorld“ fort.

So veröffentlichen und testen Sie eine ActionScript-Anwendung mit Flash Builder:

1. Wählen Sie „Run“ > „Run HelloWorld“ (Ausführen > HelloWorld ausführen).
2. Die HelloWorld-Anwendung wird gestartet.
   - Wenn beim Testen der Anwendung Fehler oder Warnungen im Ausgabefenster angezeigt werden, beheben Sie diese Fehler in den Dateien „HelloWorld.mxml“ oder „Greeter.as“. Testen Sie die Anwendung dann erneut.
   - Wenn keine Kompilierungsfehler vorliegen, wird ein Browserfenster mit der Anwendung „Hello World“ geöffnet. Der Text „Hello World!“ wird angezeigt.

   Sie haben eine einfache, aber dennoch vollständige objektorientierte Anwendung erstellt, die ActionScript 3.0 verwendet. Fahren Sie mit dem Abschnitt Erweitern der Anwendung „HelloWorld“ fort.

Erweitern der Anwendung „HelloWorld“

Um die Anwendung ein wenig interessanter zu machen, soll sie jetzt zur Eingabe eines Benutzernamens auffordern. Dann wird der eingegebene Benutzername mit einer vordefinierten Namensliste verglichen.

Zunächst aktualisieren Sie die Greeter-Klasse, um die neue Funktionalität hinzuzufügen. Dann aktualisieren Sie die Anwendung, damit die neue Funktionalität verwendet wird.

So aktualisieren Sie die Datei „Greeter.as“:

1. Öffnen Sie die Datei „Greeter.as“.
2. Ändern Sie den Inhalt der Datei wie folgt (neue und geänderte Zeilen werden in Fettschrift angezeigt):

```
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;
            }
        }
    }
}
```

Die Greeter-Klasse verfügt jetzt über mehrere neue Funktionen:

- Das Array `validNames` enthält eine Liste gültiger Benutzernamen. Beim Laden der Greeter-Klasse wird das Array mit einer drei Namen umfassenden Liste initialisiert.

- Die Methode `sayHello()` akzeptiert jetzt einen Benutzernamen und ändert die Begrüßung unter Berücksichtigung von verschiedenen Bedingungen. Wenn `userName` eine leere Zeichenfolge ist (`""`), fordert die Eigenschaft `greeting` den Benutzer zur Eingabe eines Namens auf. Wenn der Benutzername gültig ist, lautet die Begrüßung „`Hello, `*userName*`."` Wenn keine der beiden Bedingungen erfüllt ist, nimmt die Variable `greeting` den folgenden Wert an: „`Sorry `*userName*`, you are not on the list.`".
- Die Methode `validName()` gibt den Wert `true` zurück, wenn der `inputName` im Array `validNames` gefunden wurde, andernfalls `false`. Die Anweisung `validNames.indexOf(inputName)` vergleicht jede Zeichenfolge im Array `validNames` mit dem eingegebenen String `inputName`. Die `Array.indexOf()`-Methode gibt die Indexposition der ersten Instanz eines Objekts in einem Array zurück. Wenn das Objekt im Array nicht gefunden werden kann, wird der Wert -1 zurückgegeben.

Im nächsten Schritt bearbeiten Sie die Anwendungsdatei, die auf diese ActionScript-Klasse verweist.

So ändern Sie die Anwendung mithilfe von Flash Professional:

1. Öffnen Sie die Datei „HelloWorld.fla".
2. Bearbeiten Sie das Skript in Bild 1 so, dass ein leerer String (`""`) an die `sayHello()`-Methode der Greeter-Klasse übergeben wird:

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

3. Wählen Sie das Textwerkzeug in der Werkzeugpalette aus. Erstellen Sie auf der Bühne zwei neue Textfelder. Platzieren Sie sie nebeneinander direkt unter dem vorhandenen Textfeld `mainText`.
4. Geben Sie im ersten neuen Textfeld für die Bezeichnung den Text **Benutzername:** ein.
5. Markieren Sie das andere neue Textfeld und wählen Sie im Eigenschafteninspektor den Textfeldtyp „Eingabetext" aus. Wählen als Zeilentyp „Einzeilig" aus. Geben Sie als Instanzname **textIn** ein.
6. Klicken Sie auf das erste Bild der Hauptzeitleiste.
7. Fügen Sie im Bedienfeld „Aktionen" am Ende des vorhandenen Skripts die folgenden Zeilen ein:

```
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);
    }
}
```

Mit dem neuen Code werden zusätzliche Funktionen hinzugefügt:

- Die ersten beiden Zeilen definieren einfach Rahmen für zwei Textfelder.
- Ein Eingabetextfeld wie das Feld `textIn` verfügt über eine Reihe von Ereignissen, die es auslösen kann. Mit der `addEventListener()`-Methode können Sie eine Funktion festlegen, die beim Auftreten eines bestimmten Ereignistyps ausgeführt wird. Im Beispiel handelt es sich bei dem Ereignis um das Drücken einer Taste auf der Tastatur.

- Die benutzerdefinierte Funktion `keyPressed()` prüft, ob es sich bei der gedrückten Taste um die Eingabetaste handelt. Wenn dies der Fall ist, ruft die Funktion die `sayHello()`-Methode des `myGreeter`-Objekts auf und übergibt dabei als Parameter den Text aus dem Textfeld `textIn`. Auf der Grundlage des übergebenen Strings gibt diese Methode einen Begrüßungsstring zurück. Der zurückgegebene String wird dann der `text`-Eigenschaft des Textfelds `mainText` zugewiesen.

Es folgt der vollständige Skriptcode für Bild 1:

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

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

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

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

8 Speichern Sie die Datei.

9 Wählen Sie „Steuerung“ > „Film testen“ aus, um die Anwendung auszuführen.

Wenn Sie die Anwendung ausführen, werden Sie zur Eingabe eines Benutzernamens aufgefordert. Wenn dieser gültig ist („Sammy“, „Frank“ oder „Dean“), wird in der Anwendung die Bestätigungsmeldung „Hello“ angezeigt.

So ändern Sie die Anwendung mithilfe von Flash Builder:

1 Öffnen Sie die Datei „HelloWorld.mxml“.

2 Ändern Sie das `<mx:TextArea>`-Tag, um den Benutzer darauf hinzuweisen, dass der Textbereich nur zur Anzeige vorgesehen ist. Ändern Sie die Hintergrundfarbe in Hellgrau und stellen Sie das `editable`-Attribut auf `false` ein:

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

3 Fügen Sie nun die folgenden Zeilen direkt nach dem schließenden `<s:TextArea>`-Tag ein. Mit diesen Zeilen wird eine TextInput-Komponente erstellt, die dem Benutzer die Eingabe eines Wertes für den Benutzernamen ermöglicht:

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

Das `enter`-Attribut definiert, was geschieht, wenn der Benutzer im `userNameTxt`-Feld die Eingabetaste drückt. In diesem Beispiel übergibt der Code den im Feld enthaltenen Text an die `Greeter.sayHello()`-Methode. Der Begrüßungstext im `mainTxt`-Feld ändert sich entsprechend.

Die Datei „HelloWorld.mxml“ sieht nun folgendermaßen aus:

```
<?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 Speichern Sie die bearbeitete Datei „HelloWorld.mxml“. Wählen Sie „Run“ > „Run HelloWorld“ (Ausführen > HelloWorld ausführen), um die Anwendung zu starten.

Wenn Sie die Anwendung ausführen, werden Sie zur Eingabe eines Benutzernamens aufgefordert. Wenn dieser gültig ist („Sammy“, „Frank“ oder „Dean“), wird in der Anwendung die Bestätigungsmeldung „ `Hello, Benutzername`“ angezeigt.

Kapitel 3: ActionScript-Sprache und -Syntax

ActionScript 3.0 umfasst die ActionScript-Kernsprache und die Programmierschnittstelle (API, Application Programming Interface) der Adobe Flash-Plattform. Die Kernsprache ist der ActionScript-Teil, der die Sprachsyntax sowie die Datentypen der obersten Ebene definiert. ActionScript 3.0 bietet Programmzugriff auf die Laufzeitumgebungen der Adobe Flash-Plattform: Adobe Flash Player und Adobe AIR.

Die Sprache im Überblick

Objekte bilden den Kern der ActionScript 3.0-Sprache – sie sind die grundlegenden Bausteine. Jede von Ihnen deklarierte Variable, jede von Ihnen geschriebene Funktion und jede von Ihnen erstellte Klasseninstanz ist ein Objekt. Man kann sich ein ActionScript 3.0-Programm als eine Gruppe von Objekten vorstellen, die Aufgaben ausführen, auf Ereignisse reagieren und miteinander kommunizieren.

Programmierer, die mit der objektorientierten Programmierung (OOP) in Java oder C++ vertraut sind, betrachten Objekte ggf. auch als Module, die zwei Arten von Komponenten enthalten: in Mitgliedervariablen oder Eigenschaften gespeicherte Daten und Verhalten, auf die über Methoden zugegriffen wird. In ActionScript 3.0 werden Objekte zwar ähnlich, aber doch etwas anders definiert. In ActionScript 3.0 sind Objekte einfach Sammlungen mehrerer Eigenschaften. Diese Eigenschaften sind Container, die nicht nur Daten, sondern auch Funktionen und andere Objekte enthalten können. Wenn eine Funktion auf diese Weise an ein Objekt angehängt ist, spricht man von einer Methode.

Während Programmierern mit Java- oder C++-Kenntnissen die Definition in ActionScript 3.0 etwas seltsam vorkommen mag, ähnelt das Definieren von Objekttypen mit ActionScript 3.0-Klassen den Verfahren, mit denen Klassen in Java oder C++ definiert werden. Der Unterschied zwischen den zwei Definitionen für ein Objekt wird dann wichtig, wenn das ActionScript-Objektmodell und andere fortgeschrittene Themen behandelt werden. In den meisten anderen Situationen steht der Begriff *Eigenschaften* für Klassenmitgliedervariablen im Gegensatz zu Methoden. Im Referenzhandbuch zu ActionScript 3.0 für die Adobe Flash-Plattform bezieht sich der Begriff *Eigenschaften* beispielsweise auf Variablen oder auf Get/Set-Eigenschaften. Der Begriff *Methoden* steht für Funktionen, die Teil einer Klasse sind.

Ein feiner Unterschied zwischen Klassen in ActionScript und Klassen in Java oder C++ besteht darin, dass Klassen in ActionScript nicht nur abstrakte Einheiten sind. ActionScript-Klassen werden durch *Klassenobjekte* repräsentiert, in denen die Eigenschaften und Methoden der entsprechenden Klasse gespeichert sind. Dies ermöglicht Techniken, die Java- und C++-Programmierern fremd vorkommen werden, wie z. B. das Einschließen von Anweisungen oder ausführbarem Code auf der obersten Ebene einer Klasse oder eines Pakets.

Ein weiterer Unterschied zwischen ActionScript-Klassen und Java- oder C++-Klassen besteht darin, dass jede ActionScript-Klasse ein so genanntes *Prototypobjekt* aufweist. In früheren Versionen von ActionScript wurden Prototypobjekte zu *Prototypketten* verknüpft, die gemeinsam als Grundlage für die gesamte Klassenvererbungshierarchie dienten. In ActionScript 3.0 spielen Prototypobjekte im Vererbungssystem jedoch nur noch eine untergeordnete Rolle. Das Prototypobjekt kann dennoch als Alternative zu statischen Eigenschaften und Methoden verwendet werden, wenn Sie eine Eigenschaft und deren Wert für alle Instanzen einer Klasse freigeben möchten.

In der Vergangenheit konnten erfahrene ActionScript-Programmierer die Prototypkette mit speziellen integrierten Sprachelementen direkt bearbeiten. Jetzt verfügt die Sprache über eine ausgereiftere Implementierung einer klassenbasierten Programmierschnittstelle, und viele dieser speziellen Sprachelemente (wie `__proto__` und `__resolve`) sind kein Teil der Sprache mehr. Darüber hinaus schließen die Optimierungen des internen Vererbungsmechanismus, der wesentliche Leistungssteigerungen ermöglicht, einen direkten Zugriff auf den Vererbungsmechanismus aus.

Objekte und Klassen

In ActionScript 3.0 wird jedes Objekt durch eine Klasse definiert. Eine Klasse kann man sich als Vorlage oder Entwurf für den Typ eines Objekts vorstellen. Klassendefinitionen können Variable und Konstanten enthalten, in denen Datenwerte gespeichert sind, und Methoden, in denen das an die Klasse gebundene Verhalten eingekapselt ist. Bei den in den Eigenschaften gespeicherten Werten kann es sich um *Grundwerte* (Englisch „primitive values“) oder um andere Objekte handeln. Grundwerte sind Zahlen, Strings oder boolesche Werte.

ActionScript enthält verschiedene integrierte Klassen, die einen Teil der Hauptsprache darstellen. Einige dieser integrierten Klassen, z. B. Number, Boolean und String, stellen die in ActionScript verfügbaren Grundwerte dar. Andere Klassen, wie Array, Math und XML, definieren komplexere Objekte.

Alle Klassen, ob integriert oder benutzerdefiniert, sind von der Object-Klasse abgeleitet. Programmierer mit ActionScript-Erfahrungen müssen wissen, dass der Datentyp „Object“ nicht mehr der Standarddatentyp ist, obwohl alle anderen Klassen weiterhin von diesem Datentyp abgeleitet werden. In ActionScript 2.0 waren die folgenden beiden Codezeilen gleichwertig, da das Fehlen einer Typanmerkung bedeutete, dass eine Variable den Typ „Object“ annahm:

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

Mit ActionScript 3.0 wurde jedoch das Konzept von nicht typisierten Variablen eingeführt, das auf zwei Arten beschrieben werden kann:

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

Eine nicht typisierte Variable ist nicht das Gleiche wie eine Variable des Typs „Object“. Der wesentliche Unterschied besteht darin, dass nicht typisierten Variable den Sonderwert `undefined` enthalten können, während eine Variable des Typs „Object“ diesen Wert nicht annehmen kann.

Sie können eigene Klassen mithilfe des Schlüsselworts `class` definieren. Klasseneigenschaften können auf drei Arten deklariert werden: Konstanten können mit dem Schlüsselwort `const` definiert werden, Variable mit dem Schlüsselwort `var`. Get/Set-Eigenschaften werden mithilfe der Attribute `get` und `set` in einer Methodendeklaration definiert. Methoden können Sie mit dem Schlüsselwort `function` deklarieren.

Eine Klasseninstanz erstellen Sie mit dem Operator `new`. Im folgenden Beispiel wird eine Instanz der Date-Klasse namens `myBirthday` erstellt.

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

Pakete und Namespaces

Pakete und Namespaces sind miteinander verwandte Konzepte. Mit Paketen können Sie Klassendefinitionen bündeln, um die gemeinsame Nutzung von Code zu vereinfachen und Benennungskonflikte zu minimieren. Mit Namespaces steuern Sie die Sichtbarkeit von Bezeichnern, wie z. B. Eigenschaften- und Methodennamen. Namespaces können auf Code unabhängig davon angewendet werden, ob dieser sich innerhalb oder außerhalb eines Pakets befindet. Pakete ermöglichen Ihnen das Strukturieren Ihrer Klassendateien, mit Namespaces verwalten Sie die Sichtbarkeit von einzelnen Eigenschaften und Methoden.

Pakete

In ActionScript 3.0 werden Pakete mit Namespaces implementiert, dennoch sind Pakete und Namespaces keine Synonyme. Beim Deklarieren eines Paketes erstellen Sie implizit einen Sondertyp eines Namespace, der dann bei der Kompilierung garantiert bekannt ist. Explizit erstelle Namespaces hingegen sind bei der Kompilierung nicht unbedingt bekannt.

Im folgenden Beispiel wird die Direktive `package` zum Erstellen eines einfachen Pakets verwendet, das eine Klasse enthält:

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

Der Name der Klasse in diesem Beispiel lautet „SampleCode“. Da sich die Klasse im samples-Paket befindet, wandelt der Compiler den Klassennamen während der Kompilierung automatisch in dessen vollständig qualifizierten Namen (samples.SampleCode) um. Darüber hinaus wandelt der Compiler die Namen aller Eigenschaften oder Methoden um, sodass `sampleGreeting` und `sampleFunction()` zu `samples.SampleCode.sampleGreeting` und `samples.SampleCode.sampleFunction()` werden.

Viele Entwickler, insbesondere diejenigen mit Erfahrungen in der Programmierung mit Java, platzieren Klassen nur in der obersten Ebene eines Pakets. ActionScript 3.0 unterstützt in der obersten Ebene eines Pakets jedoch nicht nur Klassen, sondern auch Variablen, Funktionen und sogar Anweisungen. Eine erweiterte Einsatzmöglichkeit dieses Merkmals ist das Definieren eines Namespace auf der obersten Ebene eines Pakets, sodass er allen Klassen in diesem Paket zur Verfügung steht. Denken Sie jedoch daran, dass nur zwei Zugriffsbezeichner, `public` und `internal`, auf der obersten Ebene eines Pakets zulässig sind. Im Gegensatz zur Programmiersprache Java, bei der Sie verschachtelte Klassen als private Klassen deklarieren können, unterstützt ActionScript 3.0 weder verschachtelte noch private Klassen.

In vielerlei Hinsicht ähneln ActionScript 3.0-Pakete den Paketen in der Programmiersprache Java. Wie Sie im vorangegangenen Beispiel gesehen haben, werden vollständig qualifizierte Paketverweise ebenso wie in Java mit dem Punktoperator (.) ausgedrückt. Mit Paketen können Sie Ihren Code intuitiv und hierarchisch anordnen, sodass er auch von anderen Programmierern verwendet werden kann. Dies vereinfacht die gemeinsame Nutzung von Code, indem von Ihnen erstellte Pakete von anderen Programmierern verwendet und Sie von anderen Programmierern erstellte Pakete in Ihren Code integrieren können.

Darüber hinaus können Sie mit Paketen sicherstellen, dass von Ihnen verwendete Bezeichnernamen einmalig sind und somit nicht zu Konflikten mit anderen Bezeichnernamen führen. Viele sagen sogar, dass dies der wichtigste Vorteil von Paketen ist. Angenommen, zwei Programmierer möchten ihren Code gemeinsam verwenden. Jeder hat eine Klasse mit der Bezeichnung „SampleCode“ erstellt. Ohne Pakete würde dies zu einem Namenskonflikt führen. Die einzige Lösung wäre, eine der beiden Klassen umzubenennen. Mit Paketen lässt sich der Namenskonflikt einfach vermeiden, indem eine Klasse (oder am besten beide Klassen) in Paketen mit einmaligen Namen platziert werden.

Sie können auch eingebettete Punkte in Ihren Paketnamen aufnehmen, um verschachtelte Pakete zu erstellen. Auf diese Weise können Sie eine hierarchische Paketstruktur anlegen. Ein gutes Beispiel hierfür ist das von ActionScript 3.0 bereitgestellte flash.display-Paket, das im flash-Paket verschachtelt ist.

Der Großteil von ActionScript 3.0 ist unter dem flash-Paket organisiert. Beispielsweise enthält das flash.display-Paket die API für die Anzeigeliste und das flash.events-Paket das neue Ereignismodell.

Erstellen von Paketen

ActionScript 3.0 bietet Ihnen eine enorme Flexibilität bei der Strukturierung Ihrer Pakete, Klassen und Quelldateien. Frühere ActionScript-Versionen gestatteten nur eine Klasse pro Quelldatei und verlangten, dass der Name der Quelldatei dem Namen der Klasse entsprach. Mit ActionScript 3.0 können Sie mehrere Klassen in eine Quelldatei aufnehmen, doch für Code außerhalb der Datei ist pro Datei nur eine Klasse verfügbar. Anders ausgedrückt, in einer Paketdeklaration kann in jeder Datei nur eine Klasse deklariert werden. Sie müssen alle zusätzlichen Klassen außerhalb Ihrer Paketdefinition deklarieren. Auf diese Weise werden diese Klassen auch für Code sichtbar, der sich außerhalb dieser Quelldatei befindet. Der Name der innerhalb einer Paketdefinition deklarierten Klasse muss dem Namen der Quelldatei entsprechen.

Darüber hinaus bietet ActionScript 3.0 mehr Flexibilität bei der Deklaration von Paketen. In früheren Versionen von ActionScript waren Pakete lediglich Verzeichnisse, in denen Sie Quelldateien platzierten. Sie haben keine Pakete mit der `package`-Anweisung deklariert, sondern stattdessen den Paketnamen als Teil des vollständig qualifizierten Klassennamens in Ihre Klassendeklaration aufgenommen. Obwohl Pakete in ActionScript 3.0 noch immer Verzeichnisse darstellen, können sie mehr als nur Klassen enthalten. In ActionScript 3.0 deklarieren Sie ein Paket mit der Anweisung `package`, d. h. Sie können auch Variable, Funktionen und Namespaces auf oberster Ebene eines Pakets deklarieren. Es ist sogar möglich, ausführbare Anweisungen in die oberste Ebene eines Pakets aufzunehmen. Wenn Sie Variablen, Funktionen oder Namespaces auf der obersten Ebene eines Pakets deklarieren, stehen Ihnen nur die Attribute `public` und `internal` zur Verfügung. Darüber hinaus kann nur eine Deklaration auf Paketebene pro Datei das Attribut `public` verwenden, unabhängig davon, ob eine Klasse, Variable, Funktion oder ein Namespace deklariert wird.

Pakete eignen sich insbesondere zum Strukturieren Ihres Codes und zum Vermeiden von Namenskonflikten. Verwechseln Sie das Konzept der Pakete jedoch nicht mit dem Konzept der Klassenvererbung. Zwei Klassen im selben Paket haben den gleichen Namespace, müssen aber nicht unbedingt miteinander verwandt sein. Entsprechend hat ein verschachteltes Paket möglicherweise keine semantische Beziehung zu seinem übergeordneten Paket.

Importieren von Paketen

Wenn Sie eine Klasse in einem Paket verwenden möchten, müssen Sie entweder das Paket oder die gewünschte Klasse importieren. Diese Vorgehensweise unterscheidet sich von der in ActionScript 2.0; hier war das Importieren von Klassen optional.

Dies kann anhand des bereits genannten Beispiels der SampleCode-Klasse verdeutlicht werden. Wenn sich die Klasse in einem Paket namens „samples“ befindet, müssen Sie die Klasse mit einer der folgenden Anweisungen importieren, bevor Sie sie verwenden können:

```
import samples.*;
```

oder

```
import samples.SampleCode;
```

Im Allgemeinen müssen `import`-Anweisungen so genau wie möglich angegeben werden. Wenn Sie nur die SampleCode-Klasse aus dem samples-Paket verwenden möchten, so importieren Sie anstelle des gesamten Pakets nur die SampleCode-Klasse. Das Importieren gesamter Pakete kann zu unerwarteten Namenskonflikten führen.

Der Quellcode, der das Paket oder die Klasse definiert, muss in Ihrem *Klassenpfad* platziert werden. Der Klassenpfad ist eine benutzerdefinierte Liste der lokalen Verzeichnispfade, die festlegt, wo der Compiler nach importierten Paketen und Klassen sucht. Manchmal wird der Klassenpfad auch als *Erstellungspfad* oder *Quellpfad* bezeichnet.

Nachdem die Klasse oder das Paket korrekt importiert wurde, können Sie entweder den vollständig qualifizierten Namen der Klasse (samples.SampleCode) oder einfach nur den Klassennamen selbst (SampleCode) verwenden.

Vollständig qualifizierte Namen eignen sich insbesondere dann, wenn identisch benannte Klassen, Methoden oder Eigenschaften zu mehrdeutigem Code führen, sind aber schwierig zu verwalten, wenn sie für alle Bezeichner verwendet werden. Wenn Sie z. B. eine SampleCode-Klasseninstanz instanziieren, wird der Code durch die Verwendung von vollständig qualifizierten Namen sehr lang:

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

Je größer die Anzahl der Ebenen in den verschachtelten Paketen ist, desto schlechter ist die Lesbarkeit des Codes. Wenn Sie sicher sind, dass mehrdeutige Bezeichner kein Problem darstellen, können Sie die Lesbarkeit Ihres Codes verbessern, indem Sie einfache Bezeichner verwenden. Beispielsweise wird das Instanziieren einer neuen Instanz der SampleCode-Klasse weit weniger ausführlich, wenn Sie nur den Klassenbezeichner verwenden:

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

Wenn Sie hingegen versuchen, die Bezeichnernamen zu verwenden, ohne zunächst das entsprechende Paket bzw. die entsprechende Klasse zu importieren, kann der Compiler die Klassendefinitionen nicht finden. Wenn Sie andererseits ein Paket oder eine Klasse importieren, führt jeder Versuch, einen Namen zu definieren, der mit einem importierten Namen in Konflikt steht, zu einer Fehlermeldung.

Nach dem Erstellen eines Pakets lautet der standardmäßige Zugriffsbezeichner für alle Mitglieder dieses Pakets `internal`. Dies wiederum bedeutet, dass alle Paketmitglieder standardmäßig nur für andere Mitglieder des Pakets sichtbar sind. Soll eine Klasse auch für Code außerhalb des Pakets zugänglich sein, müssen Sie die Klasse als `public` (öffentlich) deklarieren. Das folgende Paket enthält beispielsweise zwei Klassen, SampleCode und CodeFormatter:

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

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

Die SampleCode-Klasse ist auch außerhalb des Pakets sichtbar, da sie als `public` deklariert wurde. Die CodeFormatter-Klasse ist jedoch nur innerhalb des samples-Pakets selbst sichtbar. Wenn Sie versuchen, außerhalb des samples-Pakets auf die CodeFormatter-Klasse zuzugreifen, wird eine Fehlermeldung erzeugt. Dies wird im folgenden Beispiel gezeigt:

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

Sollen beide Klassen auch für Code außerhalb des Pakets zugänglich sein, müssen Sie beide Klassen als `public` deklarieren. Sie können das Attribut `public` jedoch nicht auf die Paketdeklaration anwenden.

Vollständig qualifizierte Namen eignen sich zum Auflösen von Namenskonflikten, die bei der Verwendung von Paketen auftreten können. Ein solcher Fall kann entstehen, wenn Sie zwei Pakete importieren, in denen Klassen mit dem gleichen Bezeichner definiert sind. Das folgende Paket enthält beispielsweise eine Klasse mit der Bezeichnung „SampleCode“:

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

Wenn Sie beide Klassen importieren, entsteht bei Verwendung der SampleCode-Klasse ein Namenskonflikt. Dies wird im folgenden Beispiel gezeigt:

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

Der Compiler weiß nicht, welche SampleCode-Klasse verwendet werden soll. Um diesen Konflikt aufzulösen, müssen Sie den vollständig qualifizierten Namen jeder Klasse angeben:

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

Hinweis: *Programmierer mit Erfahrungen in C++ verwechseln häufig die `import`-Anweisung mit `#include`. Die `#include`-Direktive ist in C++ unbedingt erforderlich, da C++-Compiler nur jeweils eine Datei verarbeiten und nicht in anderen Dateien nach Klassendefinitionen suchen, es sei denn, es ist explizit eine Headerdatei enthalten. ActionScript 3.0 enthält eine `include`-Direktive, aber sie ist nicht zum Importieren von Klassen und Paketen vorgesehen. Zum Importieren von Klassen und Paketen in ActionScript 3.0 verwenden Sie die `import`-Anweisung und legen die Quelldatei mit dem Paket im Klassenpfad ab.*

Namespaces

Mit Namespaces haben Sie die Kontrolle über die Sichtbarkeit der von Ihnen erstellten Eigenschaften und Methoden. Stellen Sie sich die Zugriffskontrollbezeichner `public`, `private`, `protected` und `internal` als integrierte Namespaces vor. Wenn diese vordefinierten Zugriffskontrollbezeichner für Ihre Zwecke nicht ausreichen, können Sie eigene Namespaces erstellen.

Wenn Sie mit XML-Namespaces Erfahrung haben, wird Ihnen vieles vertraut vorkommen, obwohl Syntax und Details der ActionScript-Implementierung leicht von denen für XML abweichen. Auch wenn Sie noch nie mit Namespaces gearbeitet haben, können Sie unbesorgt sein – das Konzept ist einfach, doch die Implementierung verwendet einige spezielle Begriffe, die Sie lernen müssen.

Um das Konzept der Namespaces zu verstehen, müssen Sie zunächst einmal wissen, das der Name einer Eigenschaft oder Methode immer zwei Teile enthält: einen Bezeichner und einen Namespace. Der Bezeichner ist das, was Sie sich im Allgemeinen unter dem Namen vorstellen. Beispielsweise lauten die Bezeichner in der folgenden Klassendefinition `sampleGreeting` und `sampleFunction()`:

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

Wenn Definitionen kein Namespace-Attribut vorangestellt ist, sind die entsprechenden Namen immer durch den Standardnamespace `internal` qualifiziert. Das bedeutet, sie sind nur für aufrufende Objekte im gleichen Paket sichtbar. Ist der Compiler auf den strikten Modus gesetzt, gibt er eine Warnung aus, dass der Namespace `internal` für alle Bezeichner ohne Namespace-Attribut gilt. Um sicherzustellen, dass ein Bezeichner überall zur Verfügung steht, müssen Sie dem Bezeichnernamen explizit das Attribut `public` voranstellen. Im vorangegangenen Beispielcode haben sowohl `sampleGreeting` als auch `sampleFunction()` den Namespace-Wert `internal`.

Bei der Verwendung von Namespaces sind drei grundlegende Schritte zu beachten: Zunächst definieren Sie den Namespace mit dem Schlüsselwort `namespace`. Beispielsweise definiert der folgende Code den Namespace `version1`:

```
namespace version1;
```

Dann wenden Sie den Namespace an, indem Sie ihn anstelle eines Zugriffskontrollbezeichners in einer Eigenschafts- oder Methodendeklaration verwenden. Das folgende Beispiel fügt eine Funktion namens `myFunction()` im Namespace `version1` ein:

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

Nachdem Sie den Namespace angewendet haben, können Sie im dritten Schritt mit der Direktive `use` oder durch Qualifizieren des Bezeichnernamens mit einem Namespace auf den angewendeten Namespace verweisen. Das folgende Beispiel verweist über die `use`-Direktive auf die `myFunction()`-Funktion:

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

Sie können auch einen qualifizierten Namen verwenden, um auf die `myFunction()`-Funktion zu verweisen. Dies ist im folgenden Beispiel dargestellt:

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

Definieren von Namespaces

Namespaces enthalten einen Wert, den Uniform Resource Identifier (URI), der manchmal auch als *Namespace-Name* bezeichnet wird. Mit einem URI können Sie sicherstellen, dass Ihre Namespace-Definition einmalig ist.

Ein Namespace wird durch das Deklarieren einer Namespace-Definition erstellt. Dabei stehen Ihnen zwei Verfahren zur Verfügung: Entweder definieren Sie einen Namespace mit einem expliziten URI, als ob Sie einen XML-Namespace definieren würden, oder Sie lassen den URI weg. Das folgende Beispiel zeigt, wie ein Namespace mit einem URI definiert wird:

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

Der URI dient als eindeutiger Identifikationsstring für diesen Namespace. Wenn Sie den URI weglassen, erstellt der Compiler einen eindeutigen internen Identifikationsstring anstelle des URI, wie im folgenden Beispiel dargestellt. Sie können nicht auf diesen internen Identifikationsstring zugreifen.

```
namespace flash_proxy;
```

Nach der Definition eines Namespace – ob mit oder ohne URI – kann dieser Namespace im gleichen Gültigkeitsbereich nicht mehr neu definiert werden. Der Versuch, einen Namespace zu definieren, obwohl schon ein Namespace im gleichen Gültigkeitsbereich existiert, führt zu einem Compiler-Fehler.

Ein in einem Paket oder einer Klasse definierter Namespace ist für Code außerhalb des Pakets oder der Klasse nicht sichtbar, es sei denn, es wird der entsprechende Zugriffskontrollbezeichner verwendet. Beispielsweise wurde der Namespace `flash_proxy` im folgenden Code im flash.utils-Paket definiert. Im folgenden Code bedeutet das Fehlen eines Zugriffskontrollbezeichners, dass der Namespace `flash_proxy` nur für Code innerhalb des flash.utils-Paket und nicht für Code außerhalb des Pakets sichtbar ist:

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

Im folgenden Code wird das `public`-Attribut verwendet, um den `flash_proxy`-Namespace auch für Code außerhalb des Pakets sichtbar zu machen:

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

Anwenden von Namespaces

Unter Anwenden eines Namespace ist das Platzieren einer Definition in einem Namespace zu verstehen. In Namespaces können Funktionen, Variablen und Konstanten platziert werden (das Platzieren einer Klasse in einem benutzerdefinierten Namespace ist nicht möglich).

Betrachten Sie das Beispiel einer Funktion, die mit dem Zugriffskontroll-Namespace `public` deklariert wurde. Mit dem `public`-Attribut in einer Funktionsdefinition wird die Funktion in einem öffentlichen (public) Namespace platziert, wodurch die Funktion für jeden Code verfügbar ist. Ein definierter Namespace kann auf die gleiche Weise wie das `public`-Attribut verwendet werden, und die Definition ist für Code verfügbar, der auf Ihren benutzerdefinierten Namespace verweisen kann. Angenommen Sie haben einen Namespace `example1` definiert, so können Sie eine Methode mit der Bezeichnung `myFunction()` mit dem Attribut `example1` hinzufügen. Dies wird im folgenden Beispiel gezeigt:

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

Das Deklarieren der `myFunction()`-Methode mit dem Namespace `example1` als Attribut bedeutet, dass die Methode zum Namespace `example1` gehört.

Beim Anwenden von Namespaces müssen Sie Folgendes berücksichtigen:

- Sie können in jeder Deklaration nur einen Namespace anwenden.
- Es gibt keine Möglichkeit, ein Namespace-Attribut auf mehrere Definitionen gleichzeitig anzuwenden. Mit anderen Worten, wenn Sie Ihren Namespace auf zehn verschiedene Funktionen anwenden möchten, müssen Sie Ihren Namespace jeder dieser zehn Funktionsdefinitionen hinzufügen.
- Beim Anwenden eines Namespace können Sie keinen Zugriffskontrollbezeichner angeben, da sich Namespaces und Zugriffskontrollbezeichner gegenseitig ausschließen. Sie können also eine Funktion oder Eigenschaft zusätzlich zum Anwenden Ihres Namespace nicht als `public`, `private`, `protected` oder `internal` deklarieren.

Referenzieren von Namespaces

Wenn Sie eine Methode oder Eigenschaft verwenden, die mit einem der Zugriffskontroll-Namespaces wie `public`, `private`, `protected` und `internal` deklariert wurde, ist kein expliziter Verweis auf einen Namespace mehr erforderlich. Dies liegt daran, dass der Zugriff auf diese speziellen Namespaces über den Kontext gesteuert wird. Im Namespace `private` platzierte Definitionen stehen beispielsweise automatisch für Code in der gleichen Klasse zur Verfügung. Für benutzerdefinierte Namespaces existiert eine solche Kontextempfindlichkeit jedoch nicht. Um eine von Ihnen in einem benutzerdefinierten Namespace platzierte Methode oder Eigenschaft zu verwenden, müssen Sie den Namespace referenzieren.

Verweise auf Namespaces erstellen Sie mit der Direktive `use namespace`, oder Sie qualifizieren den Namen mit dem Namensqualifizierter (`::`). Das Referenzieren eines Namespace mit der Direktive `use namespace` „öffnet“ den Namespace, sodass er auf alle nicht qualifizierten Bezeichner angewendet werden kann. Angenommen Sie haben den Namespace `example1` definiert, so können Sie mithilfe von `use namespace example1` auf Namen in diesem Namespace zugreifen:

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

Sie können mehrere Namespaces gleichzeitig öffnen. Nachdem Sie einen Namespace mit `use namespace` geöffnet haben, bleibt er über den gesamten Codeblock, in dem er geöffnet wurde, offen. Es gibt keine Möglichkeit, einen Namespace explizit zu schließen.

Wenn mehrere Namespaces geöffnet sind, erhöht sich jedoch die Wahrscheinlichkeit von Namenskonflikten. Wenn Sie einen Namespace nicht öffnen möchten, können Sie die Direktive `use namespace` vermeiden, indem Sie den Methoden- oder Eigenschaftennamen mit dem Namespace und dem Namensqualifizierer qualifizieren. Das folgende Beispiel zeigt, wie der Name `myFunction()` mit dem Namespace `example1` qualifiziert wird:

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

Verwenden von Namespaces

In der flash.utils.Proxy-Klasse, die einen Teil von ActionScript 3.0 bildet, finden Sie ein realistisches Beispiel eines Namespace, durch dessen Verwendung Namenskonflikte verhindert werden. Die Proxy-Klasse, ein Ersatz für die `Object.__resolve`-Eigenschaft aus ActionScript 2.0, ermöglicht Ihnen das Abfangen von Verweisen auf nicht definierte Eigenschaften oder Methoden, bevor ein Fehler auftritt. Alle Methoden der Proxy-Klasse befinden sich im `flash_proxy`-Namespace, um Namenskonflikte zu verhindern.

Um die Einsatzmöglichkeiten des `flash_proxy`-Namespace besser zu verstehen, müssen Sie wissen, wie die Proxy-Klasse verwendet wird. Die Funktionsmerkmale der Proxy-Klasse stehen nur den Klassen zur Verfügung, die von der Proxy-Klasse erben. Anders ausgedrückt, wenn Sie die Methoden der Proxy-Klasse an einem Objekt anwenden möchten, muss die Klassendefinition des Objekts auf die Proxy-Klasse ausgeweitet werden. Wenn Sie beispielsweise Versuche zum Aufrufen einer nicht definierten Methode abfangen möchten, so müssen Sie die Proxy-Klasse ausweiten und dann die `callProperty()`-Methode der Proxy-Klasse außer Kraft setzen.

Sie werden sich erinnern, dass das Implementieren von Namespaces im Allgemeinen ein dreistufiger Prozess ist, der sich aus dem Definieren, Anwenden und Referenzieren eines Namespace zusammensetzt. Da Sie jedoch keine der Methoden der Proxy-Klasse explizit aufrufen, wird der `flash_proxy`-Namespace nur definiert und angewendet, aber nicht referenziert. ActionScript 3.0 definiert den `flash_proxy`-Namespace und wendet ihn in der Proxy-Klasse an. Ihr Code muss den `flash_proxy`-Namespace nur auf die Klassen anwenden, welche die Proxy-Klasse erweitern.

Der `flash_proxy`-Namespace ist etwa wie folgt in dem flash.utils-Paket definiert:

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

Der Namespace wird auf die Methoden der Proxy-Klasse angewendet, wie im folgenden Codeausschnitt der Proxy-Klasse dargestellt ist:

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

Wie der folgende Code veranschaulicht, müssen Sie zunächst sowohl die Proxy-Klasse als auch den `flash_proxy`-Namespace importieren. Dann deklarieren Sie Ihre Klasse so, dass sie die Proxy-Klasse erweitert (außerdem müssen Sie das Attribut `dynamic` hinzufügen, wenn Sie im strikten Modus kompilieren). Falls Sie die `callProperty()`-Methode außer Kraft setzen, müssen Sie den `flash_proxy`-Namespace verwenden.

```
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);
        }
    }
}
```

Wenn Sie eine Instanz der MyProxy-Klasse erstellen und eine nicht definierte Methode, wie z. B. die `testing()`-Methode, aufrufen (siehe folgendes Beispiel), fängt das Proxy-Objekt den Methodenaufruf ab und führt die Anweisungen in der überschriebenen `callProperty()`-Methode aus (in diesem Fall eine einfache `trace()`-Anweisung).

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

Die Aufnahme der Methoden der Proxy-Klasse in den `flash_proxy`-Namespace hat zwei Vorteile. Zunächst wird durch einen separaten Namespace die öffentliche Schnittstelle jeder Klasse übersichtlicher, welche die Proxy-Klasse erweitert. (Es gibt etwa ein Dutzend Methoden in der Proxy-Klasse, die Sie außer Kraft setzen können. Keine dieser Methoden wird direkt aufgerufen. Das Platzieren aller dieser Methoden im öffentlichen Namespace könnte verwirrend wirken.) Darüber hinaus werden durch Verwenden des `flash_proxy`-Namespace Namenskonflikte vermieden, falls Ihre Proxy-Unterklasse Instanzenmethoden mit Namen enthält, die Methoden in der Proxy-Klasse gleichen. Angenommen Sie möchten eine Ihrer eigenen Methoden `callProperty()` nennen. Der folgende Code wird akzeptiert, da sich Ihre Version der `callProperty()`-Methode in einem anderen Namespace befindet:

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

Mit Namespaces kann auch ein Zugriff auf Methoden oder Eigenschaften bereitgestellt werden, der nicht über die vier Zugriffskontrollbezeichner (`public`, `private`, `internal` und `protected`) möglich ist. Vielleicht haben Sie einige Dienstprogrammmethoden, die über mehrere Pakete verteilt sind. Diese Methoden sollen für alle Pakete zur Verfügung stehen, Sie möchten sie aber nicht als öffentliche Methoden deklarieren. In diesem Fall erstellen Sie einen Namespace und verwenden ihn als Ihren eigenen speziellen Zugriffskontrollbezeichner.

Im folgenden Beispiel werden zwei Funktionen aus unterschiedlichen Paketen in einem benutzerdefinierten Namespace zusammengeführt. Durch Gruppieren dieser Funktionen im gleichen Namespace können Sie beide Funktionen mit nur einer `use namespace`-Anweisung für eine Klasse oder ein Paket sichtbar machen.

Dieses Beispiel verwendet vier Dateien, um diese Technik zu veranschaulichen. Alle Dateien müssen sich in Ihrem Klassenpfad befinden. Die erste Datei, „myInternal.as“, dient zum Definieren des `myInternal`-Namespace. Da sich diese Datei in einem Paket namens „example“ befindet, müssen Sie die Datei in einem Ordner mit der Bezeichnung „example“ speichern. Der Namespace wird als `public` markiert, sodass er in andere Pakete importiert werden kann.

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

Mit der zweiten Datei, „Utility.as“, und der dritten Datei, „Helper.as“, werden die Klassen mit den Methoden definiert, die anderen Paketen zur Verfügung stehen sollen. Die Utility-Klasse befindet sich im Paket „example.alpha“. Dies bedeutet, dass die Datei in einem Ordner namens „alpha“ gespeichert werden muss, der wiederum ein Unterordner von „example“ ist. Die Helper-Klasse befindet sich im Paket „example.beta“. Dies bedeutet, dass die Datei in einem Ordner namens „beta“ gespeichert werden muss, der wiederum ein Unterordner von „example“ ist. Beide Pakete, „example.alpha“ und „example.beta“, müssen den Namespace importieren, bevor sie ihn verwenden können.

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

Die vierte Datei, „NamespaceUseCase.as“, ist die Hauptanwendungsklasse. Sie muss sich in einem Ordner auf gleicher Ebene wie „example“ befinden. In Flash Professional würde diese Klasse als Dokumentklasse für die FLA-Datei verwendet werden. Die NamespaceUseCase-Klasse importiert auch den `myInternal`-Namespace und verwendet ihn zum Aufrufen der beiden statischen Methoden, die sich in den anderen Paketen befinden. In diesem Beispiel werden statische Methoden nur zur Vereinfachung des Codes verwendet. Im Namespace `myInternal` können sowohl statische als auch Instanzmethoden platziert werden.

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

Variablen

Mit Variablen können Sie Werte speichern, die Sie in Ihrem Programm verwenden. Zum Deklarieren einer Variablen verwenden Sie die `var`-Anweisung mit dem Variablennamen. In ActionScript 3.0 ist die `var`-Anweisung immer erforderlich. Beispielsweise wird mit der folgenden Zeile eine Variable namens `i` deklariert:

```
var i;
```

Wenn Sie beim Deklarieren einer Variablen die `var`-Anweisung weglassen, tritt im strikten Modus ein Compiler-Fehler und im Standardmodus ein Laufzeitfehler auf. So führt beispielsweise die folgende Codezeile zu einem Fehler, wenn die `i`-Variable nicht zuvor definiert wurde:

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

Das Zuweisen einer Variablen zu einem Datentyp muss bereits beim Deklarieren der Variablen geschehen. Das Deklarieren einer Variablen, ohne ihr einen Variablentyp zuzuweisen, ist zwar zulässig, erzeugt im strikten Modus jedoch eine Compiler-Warnung. Sie weisen einen Variablentyp zu, indem Sie dem Variablennamen einen Doppelpunkt (:) gefolgt vom Variablentyp anhängen. Im folgenden Code wird beispielsweise eine Variable `i` des Typs „int“ deklariert:

```
var i:int;
```

Mit dem Zuweisungsoperator (`=`) können Sie einer Variablen einen Wert zuweisen. Im folgenden Code wird beispielsweise eine Variable `i` deklariert und der Wert 20 zugewiesen:

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

Vielleicht ist es für Sie einfacher, der Variablen den Wert schon beim Deklarieren zuzuweisen. Dies wird im folgenden Beispiel gezeigt:

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

Das Verfahren, einen Wert schon beim Deklarieren der Variablen zuzuweisen, wird allgemein nicht nur beim Zuweisen von Grundwerten wie Ganzzahlen und Strings, sondern auch beim Erstellen von Arrays oder Instanziieren von Klasseninstanzen eingesetzt. Im folgenden Beispiel wird mit nur einer Codezeile ein Array deklariert und ein Wert zugewiesen.

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

Mit dem Operator `new` können Sie eine Instanz einer Klasse erstellen. Im folgenden Beispiel wird eine Instanz namens `CustomClass` erstellt und einer Variablen namens `customItem` ein Verweis auf die neu erstellte Klasseninstanz zugewiesen:

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

Auch wenn Sie mehrere Variable deklarieren müssen, können Sie mit einer Codezeile arbeiten. In diesem Fall verwenden Sie den Komma-Operator (`,`), um die Variablen voneinander zu trennen. Im folgenden Code werden beispielsweise drei Variable auf einer Codezeile deklariert:

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

Auf der gleichen Codezeile können Sie jeder Variablen auch einen Wert zuweisen. Im folgenden Code werden beispielsweise drei Variable (`a`, `b` und `c`) deklariert, und jeder Variablen wird ein Wert zugewiesen:

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

Der Nachteil beim Verwenden des Komma-Operators zum Gruppieren von Variablendeklarationen in nur einer Anweisung ist jedoch, dass die Lesbarkeit Ihres Codes beeinträchtigt wird.

Gültigkeitsbereich

Der *Gültigkeitsbereich* einer Variablen ist der Bereich Ihres Codes, in dem durch einen lexikalischen Verweis auf die Variable zugegriffen werden kann. Eine *globale* Variable ist in allen Bereichen Ihres Codes definiert, während eine *lokale* Variable nur in einem bestimmten Teil Ihres Codes definiert ist. In ActionScript 3.0 sind Variablen immer dem Gültigkeitsbereich der Funktion oder Klasse zugeordnet, in der sie deklariert wurden. Eine globale Variable ist eine Variable, die außerhalb einer Funktions- oder Klassendefinition definiert ist. Im folgenden Code wird beispielsweise eine globale Variable `strGlobal` erstellt, indem sie außerhalb einer Funktion deklariert wird. Das Beispiel zeigt, dass eine globale Variable sowohl innerhalb als auch außerhalb der Funktionsdefinition verfügbar ist.

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

Sie deklarieren eine lokale Variable, indem Sie die Variable innerhalb einer Funktionsdefinition deklarieren. Der kleinste Codebereich, für den Sie eine lokale Variable definieren können, ist eine Funktionsdefinition. Eine lokale Variable, die in einer Funktion deklariert wurde, existiert nur in dieser Funktion. Wenn Sie beispielsweise eine Variable namens `str2` in einer Funktion namens `localScope()` deklarieren, steht diese Variable außerhalb der Funktion nicht zur Verfügung.

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

Wenn der Name, den Sie für Ihre lokale Variable verwenden, bereits als globale Variable deklariert ist, hat die lokale Definition Vorrang vor der globalen Definition, solange sich die lokale Variable im Gültigkeitsbereich befindet. Die globale Variable existiert weiterhin außerhalb der Funktion. Im folgenden Code wird beispielsweise eine globale Stringvariable namens `str1` und dann eine lokale Variable mit demselben Namen in der Funktion `scopeTest()` erstellt. Die Anweisung `trace` in der Funktion gibt den lokalen Wert der Variablen zurück, und die Anweisung `trace` außerhalb der Funktion gibt den globalen Wert der Variablen zurück.

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

ActionScript-Variablenhaben im Gegensatz zu Variablen in C++ und Java keinen Gültigkeitsbereich auf Blockebene. Ein Codeblock ist eine Gruppe von Anweisungen zwischen einer öffnenden geschweiften Klammer ({) und einer schließenden geschweiften Klammer (}). In einigen Programmiersprachen, wie beispielsweise C++ und Java, stehen Variablen die in einem Codeblock deklariert wurden, außerhalb dieses Codeblocks nicht zur Verfügung. Diese Einschränkung des Gültigkeitsbereichs wird als Gültigkeitsbereich auf Blockebene bezeichnet. Eine solche Einschränkung gibt es in ActionScript nicht. Wenn Sie also eine Variable innerhalb eines Codeblocks deklarieren, steht die Variable nicht nur in diesem Codeblock, sondern auch in anderen Teilen der Funktion zur Verfügung, zu welcher der Codeblock gehört. Beispielsweise enthält die folgende Funktion Variablen die in verschiedenen Block-Gültigkeitsbereichen definiert sind. Alle Variablen stehen in der gesamten Funktion zur Verfügung.

```
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"]);
```

Eine interessante Auswirkung des Fehlens eines Gültigkeitsbereichs auf Blockebene ist, dass Sie eine Variable schon vor dem Deklarieren für Lese- und Schreibvorgänge verwenden können, solange sie vor dem Ende der Funktion deklariert wird. Dies beruht auf der so genannten *Hoisting*-Technik (etwa: Hochziehen), die dafür sorgt, dass der Compiler alle Variablendeklarationen an den Anfang der Funktion verschiebt. Beispielsweise kann der folgende Code kompiliert werden, obwohl die erste `trace()`-Funktion für die Variable `num` schon vor der Deklaration der `num`-Variable ausgeführt wird.

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

Der Compiler verschiebt jedoch keine Zuweisungsanweisungen an den Anfang einer Funktion. Dies erklärt, warum die erste `trace()`-Funktion für `num` zum Ergebnis `NaN` (not a number) führt, dem Standardwert für Variablen des Datentyps „Number“. Aus diesem Grund können Sie Variablen schon vor ihrer Deklaration Werte zuweisen. Dies wird im folgenden Beispiel gezeigt:

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

Standardwerte

Ein *Standardwert* ist der Wert, den eine Variable enthält, bevor Sie ihren Wert einstellen. Sie *initialisieren* eine Variable, wenn Sie ihren Wert das erste Mal einstellen. Wenn Sie eine Variable deklarieren, ohne ihren Wert einzustellen, ist diese Variable *nicht initialisiert*. Der Wert einer nicht initialisierten Variablen hängt von ihrem Datentyp ab. In der folgenden Tabelle sind die Standardwerte der Variablen nach Datentyp aufgeführt:

Datentyp	**Standardwert**
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`

Datentyp	Standardwert
uint	0
Nicht deklariert (entspricht der Typanmerkung `*`)	`undefined`
Alle weiteren Klassen, einschließlich benutzerdefinierter Klassen.	`null`

Bei Variablen des Typs „Number“ lautet der Standardwert `NaN` (not a number). Dies ist ein Sonderwert, der vom IEEE-754-Standard definiert wurde, um einen Wert auszudrücken, der keine Zahl darstellt.

Wenn Sie beim Deklarieren einer Variablen keinen Datentyp angeben, gilt der Standarddatentyp `*`. Dies bedeutet, dass die Variable nicht typisiert ist. Wenn Sie darüber hinaus eine nicht typisierte Variable nicht mit einem Wert initialisieren, lautet der Standardwert `undefined`.

Bei anderen Datentypen als „Boolean“, „Number“, „int“ und „uint“ lautet der Standardwert einer nicht initialisierten Variable `null`. Dies gilt nicht nur für alle von ActionScript 3.0 definierten Klassen, sondern auch für alle von Ihnen erstellten benutzerdefinierten Klassen.

Der Wert `null` ist für Variable des Typs „Boolean“, „Number“, „int“ oder „uint“ nicht zulässig. Wenn Sie versuchen, einer solchen Variablen den Wert `null` zuzuweisen, wird der Wert in den Standardwert dieses Datentyps umgewandelt. Sie können den Wert `null` für Variablen des Typs „Object“ zuweisen. Wenn Sie versuchen, einer Variablen des Typs „Object“ den Wert `undefined` zuzuweisen, wird der Wert zu `null` umgewandelt.

Für Variable des Typs „Number“ gibt es eine besondere Funktion auf oberster Ebene namens `isNaN()`, die den booleschen Wert `true` zurückgibt, wenn die Variable keine Zahl ist. Andernfalls gibt sie `false` zurück.

Datentypen

Ein *Datentyp* definiert Werte. Beispielsweise lässt der Datentyp „Boolean“ zwei Werte zu: `true` und `false`. Neben dem Datentyp „Boolean“ definiert ActionScript 3.0 weitere häufig verwendete Datentypen wie „String“, „Number“ und „Array“. Sie können Ihre eigenen Datentypen definieren, indem Sie benutzerdefinierte Werte mit Klassen oder Schnittstellen definieren. Alle Werte in ActionScript 3.0 sind Objekte, unabhängig davon, ob es sich um Grundwerte oder um komplexe Werte handelt.

Ein *Grundwert* ist ein Wert, der einem der folgenden Datentypen angehört: „Boolean“, „int“, „Number“, „String“ und „uint“. Vorgänge mit Grundwerten sind in der Regel schneller als Vorgänge mit komplexen Werten, da Grundwerte in ActionScript so gespeichert werden, dass Speicher- und Geschwindigkeitsoptimierungen möglich sind.

Hinweis: *Falls Sie Interesse an den technischen Details haben: ActionScript speichert Grundwerte intern als unveränderliche Objekte. Da sie als unveränderliche Objekte gespeichert werden, ist die Übergabe über einen Verweis genauso effektiv wie die Übergabe über einen Wert. Dies belegt weniger Speicher und erhöht die Ausführungsgeschwindigkeit, da Verweise in der Regel deutlich kleiner sind als die Werte selbst.*

Ein *komplexer Wert* ist ein Wert, bei dem es sich nicht um einen Grundwert handelt. Zu den Datentypen, die komplexe Werte definieren, gehören „Array“, „Date“, „Error“, „Function“, „RegExp“, „XML“ und „XMLList“.

Viele Programmiersprachen unterscheiden zwischen Grundwerten und deren Wrapper-Objekten. Beispielsweise verfügt Java über einen int-Grundwert und eine java.lang.Integer-Klasse, die als Wrapper dient. Java-Grundwerte sind im Gegensatz zu ihren Wrappern keine Objekte. Dadurch eignen sich Grundwerte für einige Vorgänge gut, während Wrapper-Objekte für andere Vorgänge besser geeignet sind. In ActionScript 3.0 wird aus praktischen Gründen nicht zwischen Grundwerten und deren Wrapper-Objekten unterschieden. Alle Werte, auch Grundwerte, sind Objekte. Die Laufzeit behandelt diese Grundtypen als Sonderfälle, die sich wie Objekte verhalten, aber nicht den normalen Aufwand zum Erstellen von Objekten erfordern. Das heißt, dass die beiden folgenden Anweisungen gleichwertig sind:

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

Alle oben aufgeführten Grund- und komplexen Datentypen sind durch die ActionScript 3.0-Kernklassen definiert. Mit den Kernklassen können Sie Objekte mit Literalwerten erstellen, ohne dass Sie den `new`-Operator verwenden müssen. Beispielsweise können Sie ein Array mit einem Literalwert oder dem Array-Klassenkonstruktor erstellen, wie im Folgenden dargestellt:

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

Typüberprüfung

Die Typüberprüfung kann während der Kompilierung oder zur Laufzeit erfolgen. Statisch typisierte Sprachen wie C++ und Java führen die Typüberprüfung während der Kompilierung durch. Dynamisch typisierte Sprachen wie Smalltalk und Python führen die Typüberprüfung zur Laufzeit durch. Als dynamisch typisierte Sprache führt ActionScript 3.0 die Typüberprüfung zur Laufzeit durch, unterstützt mit einem speziellen Compiler-Modus namens *strikter Modus* aber auch eine Typüberprüfung während der Kompilierung. Im strikten Modus wird die Typüberprüfung sowohl während der Kompilierung als auch zur Laufzeit durchgeführt, im Standardmodus wird sie nur zur Laufzeit durchgeführt.

Dynamisch typisierte Sprachen bieten zwar eine hohe Flexibilität beim Strukturieren Ihres Codes, dafür werden Typfehler erst zur Laufzeit festgestellt. Statisch typisierte Sprachen melden Typfehler bereits während der Kompilierung. Dazu müssen die Typinformationen jedoch bereits zur Kompilierung bekannt sein.

Typüberprüfung während der Kompilierung

Die Typüberprüfung während der Kompilierung wird insbesondere bei größeren Projekten bevorzugt, da die Datentypflexibilität mit wachsender Projektgröße weniger wichtig als das rechtzeitige Erfassung von Typfehlern wird. Aus diesem Grund ist der ActionScript-Compiler in Flash Professional und Flash Builder standardmäßig auf den strikten Modus eingestellt.

Adobe Flash Builder

Sie können den strikten Modus in Flash Builder über die Einstellungen des ActionScript-Compilers im Dialogfeld für Projekteigenschaften deaktivieren.

Der Compiler muss die Datentypinformationen der Variablen oder Ausdrücke in Ihrem Code kennen, um eine Typüberprüfung während der Kompilierung durchführen zu können. Um einen Datentyp für eine Variable explizit zu deklarieren, fügen Sie den Doppelpunktoperator (`:`) gefolgt vom Datentyp als Suffix zum Variablennamen hinzu. Um einem Parameter einen Datentyp zuzuordnen, verwenden Sie den Doppelpunktoperator gefolgt vom Datentyp. Im folgenden Code werden dem Parameter `xParam` Datentypinformationen hinzugefügt. Zudem wird eine Variable `myParam` mit einem expliziten Datentyp deklariert:

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

Im strikten Modus meldet der ActionScript-Compiler Typdiskrepanzen als Compiler-Fehler. Im folgenden Code wird ein Funktionsparameter `xParam` des Typs „Object“ deklariert, im weiteren Verlauf wird jedoch versucht, diesem Parameter Werte des Typs „String“ und „Number“ zuzuweisen. Im strikten Modus führt dies zu einem Compiler-Fehler.

```
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);
    }
}
```

Auch im strikten Modus können Sie die Typüberprüfung während der Kompilierung selektiv deaktivieren, indem Sie die rechte Seite einer Zuweisungsanweisung nicht typisieren. Um eine Variable oder einen Ausdruck als nicht typisiert zu kennzeichnen, lassen Sie entweder die Typanmerkung weg oder verwenden die Sondertypanmerkung `*` (Sternchen). Wenn beispielsweise der `xParam`-Parameter aus dem vorherigen Beispiel so geändert wird, dass er keine Typanmerkung mehr aufweist, wird der Code im strikten Modus kompiliert:

```
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");
```

Typüberprüfung zur Laufzeit

Die Typüberprüfung zur Laufzeit findet in ActionScript 3.0 unabhängig davon statt, ob Sie im strikten Modus oder im Standardmodus kompilieren. Stellen Sie sich eine Situation vor, bei der der Wert 3 als Argument an eine Funktion übergeben wird, die ein Array erwartet. Im strikten Modus erzeugt der Compiler einen Fehler, weil der Wert 3 nicht mit dem Datentyp „Array“ kompatibel ist. Wenn Sie den strikten Modus deaktivieren und den Standardmodus ausführen, meldet der Compiler keine Typdiskrepanzen, aber eine Typüberprüfung zur Laufzeit führt zu einem Laufzeitfehler.

Im folgenden Beispiel ist eine Funktion namens `typeTest()` dargestellt, die ein Array-Argument erwartet, stattdessen jedoch den Wert 3 empfängt. Dies führt im Standardmodus zu einem Laufzeitfehler, da der Wert 3 kein Mitglied des deklarierten Datentyps des Parameters (Array) ist.

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

Es gibt auch Situationen, in denen ein Laufzeitfehler erzeugt wird, obwohl Sie im strikten Modus arbeiten. In einem solchen Fall haben Sie den strikten Modus verwendet, die Typüberprüfung während der Kompilierung jedoch durch das Verwenden einer nicht typisierten Variablen ausgeschaltet. Durch Verwenden einer nicht typisierten Variablen wird die Typüberprüfung nicht deaktiviert, sondern bis zur Laufzeit zurückgestellt. Angenommen, die `myNum`-Variable aus dem vorhergehenden Beispiel weist keinen deklarierten Datentyp auf. In diesem Fall kann der Compiler keine Typdiskrepanz erkennen, es wird jedoch ein Laufzeitfehler erzeugt, da der Laufzeitwert von `myNum` (der als Ergebnis der Zuweisungsanweisung auf 3 festgelegt ist) mit dem Datentyp von `xParam` verglichen wird, der auf „Array“ gesetzt ist.

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

Die Typüberprüfung zur Laufzeit ermöglicht auch eine flexiblere Verwendung der Vererbung als die Typüberprüfung während der Kompilierung. Durch das Zurückstellen der Typüberprüfung bis zur Laufzeit können Sie im Standardmodus auch dann auf die Eigenschaften einer Unterklasse verweisen, wenn Sie *verallgemeinern (upcast)*. Eine Verallgemeinerung findet statt, wenn Sie eine Basisklasse zum Deklarieren des Typs einer Klasseninstanz verwenden, jedoch eine Unterklasse zum Instanziieren angeben. Beispielsweise können Sie eine Klasse mit dem Namen „ClassBase“ erstellen, die erweitert werden kann (Klassen mit dem Attribut `final` können nicht erweitert werden):

```
class ClassBase
{
}
```

Anschließend können Sie eine Unterklasse von „ClassBase“ namens „ClassExtender“ erstellen, die eine Eigenschaft mit der Bezeichnung `someString` aufweist. Dies wird im folgenden Code gezeigt:

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

Mit beiden Klassen können Sie eine Klasseninstanz erstellen, die mit dem Datentyp „ClassBase“ deklariert ist, jedoch mit dem ClassExtender-Konstruktor instanziiert wird. Eine Verallgemeinerung wird als sichere Operation angesehen, da die Basisklasse keine Eigenschaften oder Methoden enthält, die nicht in der Unterklasse vorhanden sind.

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

Eine Unterklasse kann jedoch Eigenschaften oder Methoden enthalten, die in der Basisklasse nicht enthalten sind. Beispielsweise enthält die ClassExtender-Klasse die Eigenschaft `someString`, die in der ClassBase-Klasse nicht vorhanden ist. Im Standardmodus von ActionScript 3.0 können Sie mit der `myClass`-Instanz auf diese Eigenschaft verweisen, ohne einen Fehler während der Kompilierung zu erzeugen. Dies wird im folgenden Beispiel gezeigt:

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

is-Operator

Mit dem `is`-Operator können Sie testen, ob eine Variable oder ein Ausdruck Mitglied eines bestimmten Datentyps ist. In früheren Versionen von ActionScript wurde diese Funktion vom `instanceof`-Operator bereitgestellt, in ActionScript 3.0 sollte der `instanceof`-Operator jedoch nicht mehr zum Testen der Datentypmitgliedschaft verwendet werden. Stattdessen sollten Sie zur manuellen Typüberprüfung den `is`-Operator anstelle des `instanceof`-Operators verwenden, da der Ausdruck `x instanceof y` lediglich die Prototypkette von `x` auf das Vorhandensein von `y` überprüft (und die Prototypkette in ActionScript 3.0 kein vollständiges Bild der Vererbungshierarchie bietet).

Der `is`-Operator überprüft die genaue Vererbungshierarchie. Er kann also nicht nur überprüfen, ob ein Objekt eine Instanz einer bestimmten Klasse ist, sondern auch, ob ein Objekt eine Instanz einer Klasse ist, die eine bestimmte Schnittstelle implementiert. In dem folgenden Beispielcode wird eine Instanz der Sprite-Klasse mit der Bezeichnung `mySprite` erstellt und mit dem Operator `is` überprüft, ob `mySprite` eine Instanz der Sprite- und DisplayObject-Klassen ist und ob es die IEventDispatcher-Schnittstelle implementiert:

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

Der `is`-Operator überprüft die Vererbungshierarchie und meldet unverzüglich, ob `mySprite` mit den Sprite- und DisplayObject-Klassen kompatibel ist (die Sprite-Klasse ist eine Unterklasse der DisplayObject-Klasse). Außerdem überprüft der `is`-Operator, ob `mySprite` von einer der Klassen erbt, welche die IEventDispatcher-Schnittstelle implementieren. Da die Sprite-Klasse von der EventDispatcher-Klasse erbt, die wiederum die IEventDispatcher-Schnittstelle implementiert, meldet der `is`-Operator folgerichtig, dass `mySprite` die gleiche Schnittstelle implementiert.

Das folgende Beispiel zeigt die gleichen Tests wie das vorangegangene Beispiel, diesmal jedoch mit dem `instanceof`-Operator anstelle des `is`-Operators. Der `instanceof`-Operator erkennt richtig, dass `mySprite` eine Instanz von Sprite oder DisplayObject ist, gibt jedoch den Wert `false` zurück, wenn getestet wird, ob `mySprite` die IEventDispatcher-Schnittstelle implementiert.

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

as-Operator

Mit dem `as`-Operator können Sie auch testen, ob ein Ausdruck Mitglied eines bestimmten Datentyps ist. Im Gegensatz zum `is`-Operator liefert der `as`-Operator keinen booleschen Wert. Der `as`-Operator gibt den Wert des Ausdrucks anstelle von `true` und `null` anstelle von `false` zurück. Das folgende Beispiel zeigt, was geschieht, wenn in einem einfachen Fall mit dem `as`-Operator anstelle des `is`-Operators überprüft werden soll, ob eine Sprite-Instanz ein Mitglied der Datentypen „DisplayObject“, „IEventDispatcher“ und „Number“ ist.

```
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
```

Wenn Sie den `as`-Operator verwenden, muss der Operand auf der rechten Seite ein Datentyp sein. Der Versuch, einen anderen Ausdruck als einen Datentyp als Operanden auf der rechten Seite zu verwenden, führt zu einem Fehler.

Dynamische Klassen

Eine *dynamische* Klasse definiert ein Objekt, das zur Laufzeit durch das Hinzufügen oder Ändern von Eigenschaften und Methoden verändert werden kann. Eine nicht dynamische Klasse, wie z. B. die String-Klasse, wird als *versiegelte* Klasse bezeichnet. Einer versiegelten Klasse können keine Eigenschaften oder Methoden zur Laufzeit hinzugefügt werden.

Dynamische Klassen werden mit dem Attribut `dynamic` beim Deklarieren einer Klasse erstellt. Im folgenden Code wird eine dynamische Klasse namens `Protean` erstellt:

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

Wenn Sie später eine Instanz der `Protean`-Klasse instanziieren, können Sie dieser Instanz außerhalb der Klassendefinition Eigenschaften oder Methoden hinzufügen. Im folgenden Code wird beispielsweise eine Instanz der `Protean`-Klasse erstellt und der Instanz eine Eigenschaft namens `aString` sowie eine Eigenschaft namens `aNumber` hinzugefügt:

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

Eigenschaften, die Sie der Instanz einer dynamischen Klasse hinzufügen, sind Laufzeit-Entitäten. Daher wird die Typüberprüfung zur Laufzeit durchgeführt. Einer auf diese Weise hinzugefügten Eigenschaft können Sie keine Typanmerkung hinzufügen.

Sie können der `myProtean`-Instanz auch eine Methode hinzufügen, indem Sie eine Funktion definieren und diese Funktion an eine Eigenschaft der `myProtean`-Instanz anhängen. Im folgenden Code wird die trace-Anweisung in eine Methode namens `traceProtean()` verschoben:

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

Auf diese Art erstellte Methoden haben jedoch keinen Zugriff auf private Eigenschaften oder Methoden der Protean-Klasse. Darüber hinaus müssen sogar Verweise auf öffentliche Eigenschaften oder Methoden der `Protean`-Klasse entweder durch das Schlüsselwort `this` oder durch den Klassennamen qualifiziert werden. Das folgende Beispiel zeigt, wie die `traceProtean()`-Methode versucht, auf die privaten und öffentlichen Variablen der `Protean`-Klasse zuzugreifen.

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

Datentypbeschreibungen

Die Grunddatentypen umfassen „Boolean“, „int“, „Null“, „Number“, „String“, „uint“ und „void“. Darüber hinaus definieren die ActionScript-Kernklassen die folgenden komplexen Datentypen: Object, Array, Date, Error, Function, RegExp, XML und XMLList.

Boolean-Datentyp

Der Boolean-Datentyp beinhaltet zwei Werte: `true` und `false`. Für Variable des Typ „Boolean“ sind keine anderen Werte zulässig. Der Standardwert einer Boolean-Variablen, die zwar deklariert, jedoch nicht initialisiert wurde, lautet `false`.

int-Datentyp

Der Datentyp „int“ wird intern als Ganzzahl mit 32 Bit gespeichert und umfasst Ganzzahlen im Bereich von

-2.147.483.648 (-2^{31}) bis einschließlich 2.147.483.647 ($2^{31} - 1$). Frühere Versionen von ActionScript boten lediglich den Datentyp „Number“, der sowohl für Ganzzahlen als auch für Gleitkommazahlen verwendet wurde. In ActionScript 3.0 können Sie jetzt auch auf untergeordnete Maschinentypen für 32-Bit-Ganzzahlen mit und ohne Vorzeichen zugreifen. Wenn für eine Variable keine Gleitkommazahlen erforderlich sind, bietet der Datentyp „int“ eine höhere Geschwindigkeit und Effizienz als der Datentyp „Number“.

Für ganzzahlige Werte außerhalb des Bereichs für int-Werte verwenden Sie den Datentyp „Number“, der Werte zwischen plus/minus 9.007.199.254.740.992 (ganzzahlige Werte mit 53 Bit) verarbeiten kann. Der Standardwert für Variable des Datentyps „int“ lautet 0.

Null-Datentyp

Der Null-Datentyp umfasst nur den Wert `null`. Dies ist der Standardwert für den Datentyp „String“ und alle Klassen, die komplexe Datentypen definieren. Hierzu gehört auch die Object-Klasse. Keiner der anderen Grunddatentypen, wie „Boolean“, „Number“, „int“ und „uint“, enthält den Wert `null`. Wenn Sie versuchen, Variablen des Typs „Boolean“, „Number“, „int“ oder „uint“ den Wert `null` zuzuweisen, wird `null` zur Laufzeit in den entsprechenden Standardwert umgewandelt. Sie können diesen Datentyp nicht als Typanmerkung verwenden.

Number-Datentyp

In ActionScript 3.0 kann der Datentyp „Number“ Ganzzahlen, vorzeichenlose Ganzzahlen und Gleitkommazahlen darstellen. Um die Leistung zu maximieren, sollten Sie den Datentyp „Number“ jedoch nur für ganzzahlige Werte größer als die Speichergrenze für `int`- und `uint`-Typen mit 32 Bit oder für Gleitkommazahlen verwenden. Zum Speichern als Gleitkommazahl müssen Sie mit der Zahl ein Dezimalzeichen angeben. Ohne das Dezimalzeichen wird die Zahl als Ganzzahl gespeichert.

Der Datentyp „Number“ verwendet eine Gleitkommazahl nach dem IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754) mit doppelter Genauigkeit (64 Bit). Dieser Standard legt fest, wie Gleitkommazahlen mit 64 verfügbaren Bits gespeichert werden. Ein Bit legt fest, ob die Zahl positiv oder negativ ist. 11 Bit werden für den Exponenten verwendet, der als Basis 2 gespeichert wird. Die verbleibenden 52 Bit werden zum Speichern des *Signifikand* (auch als *Mantisse* bezeichnet) verwendet. Dies ist die Zahl, die durch den Exponenten potenziert wird.

Indem einige Bits zum Speichern eines Exponenten verwendet werden, kann der Datentyp „Number“ wesentlich größere Gleitkommazahlen speichern, als wenn alle Bits für die Mantisse verwendet werden. Würde der Datentyp „Number“ alle 64 Bit zum Speichern der Mantisse verwenden, könnte lediglich eine Zahl in der Größe von 2^{65} -1 gespeichert werden. Indem 11 Bit zum Speichern eines Exponenten verwendet werden, kann der Datentyp „Number“ die Mantisse bis zu einer Potenz von 2^{1023} anheben.

Die Höchst- und Mindestwerte, die der Datentyp „Number“ darstellen kann, sind in den statischen Eigenschaften der Number-Klasse namens `Number.MAX_VALUE` und `Number.MIN_VALUE` gespeichert.

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

Dieser enorm große Zahlenbereich geht jedoch zu Lasten der Genauigkeit. Der Datentyp „Number“ verwendet 52 Bit zum Speichern der Mantisse. Dadurch sind Zahlen, die mehr als 52 Bit zur exakten Darstellung benötigen, wie z. B. der Bruch 1/3, nur Annäherungen. Wenn die Anwendung Dezimalzahlen mit absoluter Präzision erfordert, benötigen Sie Software, die anstelle von Binär-Gleitkommaarithmetik die Dezimal-Gleitkommaarithmetik verwendet.

Wenn Sie ganzzahlige Werte mit dem Datentyp „Number“ speichern, werden nur die 52 Bit der Mantisse verwendet. Der Datentyp „Number“ verwendet diese 52 Bit sowie ein spezielles verborgenes Bit, um Ganzzahlen im Bereich von -9.007.199.254.740.992 (-2^{53}) bis 9.007.199.254.740.992 (2^{53}) darzustellen.

Der Wert `NaN` wird nicht nur als Standardwert für Variablen mit dem `Number`-Typ verwendet, sondern auch als Ergebnis für jeden Vorgang, der eine Zahl zurückgeben sollte, aber einen anderen Wert liefert. Wenn Sie beispielsweise versuchen, die Quadratwurzel einer negativen Zahl zu berechnen, lautet das Ergebnis `NaN`. Weitere spezielle Number-Werte sind *positive infinity* (positive Unendlichkeit) und *negative infinity* (negative Unendlichkeit) .

Hinweis: *Das Ergebnis einer Division durch `0` ist nur dann `NaN`, wenn der Divisor ebenfalls `0` ist. Eine Division durch `0` ergibt den Wert `infinity` (positive Unendlichkeit), wenn der Dividend positiv ist, oder `-infinity` (negative Unendlichkeit), wenn der Dividend negativ ist.*

String-Datentyp

Der Datentyp „String“ stellt eine Zeichenfolge von 16 Bit dar. Strings werden intern als Unicode-Zeichen im Format UTF-16 gespeichert. Strings sind, ebenso wie in der Programmiersprache Java, unveränderliche Werte. Eine Operation mit einem Stringwert liefert eine neue Instanz des Strings. Der Standardwert für eine Variable, die mit dem Datentyp „String“ deklariert ist, lautet `null`. Der Wert `null` ist nicht mit einem leeren String (`""`) identisch. Beim Wert `null` ist in der Variablen kein Wert gespeichert; bei einem leeren String enthält die Variable einen Wert, der ein String ohne Zeichen ist.

uint-Datentyp

Der Datentyp „uint“ wird intern als vorzeichenlose Ganzzahl mit 32 Bit gespeichert und enthält Ganzzahlen im Bereich von 0 bis einschließlich 4.294.967.295 (2^{32} - 1). Sie verwenden den Datentyp „uint“ für Sonderfälle, die nicht-negative Ganzzahlen verlangen. Beispielsweise müssen Sie den Datentyp „uint“ zur Darstellung von Farbwerten für Pixel verwenden, da der Datentyp „int“ ein internes Vorzeichen-Bit umfasst, das für die Verarbeitung von Farbwerten nicht geeignet ist. Für ganzzahlige Werte größer als der uint-Höchstwert verwenden Sie den Datentyp „Number“, der ganzzahlige Werte mit 53 Bit verarbeiten kann. Der Standardwert für Variablen des Datentyps „uint“ lautet 0.

Void-Datentyp

Der Datentyp „void“ umfasst nur den Wert `undefined`. In früheren Versionen von ActionScript war `undefined` der Standardwert für Instanzen der Object-Klasse. In ActionScript 3.0 lautet der Standardwert für Object-Instanzen `null`. Wenn Sie versuchen, einer Instanz der Object-Klasse den Wert `undefined` zuzuweisen, wird der Wert in `null` umgewandelt. Der Wert `undefined` kann nur nicht typisierten Variablen zugewiesen werden. Nicht typisierte Variablen sind Variablen, denen entweder eine Typanmerkung fehlt oder die das Sternchen-Symbol (`*`) als Typanmerkung verwenden. Sie können `void` nur als Rückgabe-Typanmerkung verwenden.

Object-Datentyp

Der Object-Datentyp wird von der Object-Klasse definiert. Die Object-Klasse dient als Basisklasse für alle Klassendefinitionen in ActionScript. Der Datentyp „Object“ in ActionScript 3.0 unterscheidet sich vom Datentyp „Object“ in früheren Versionen in dreifacher Hinsicht. Zunächst einmal ist der Datentyp „Object“ nicht mehr der Standarddatentyp, der Variablen ohne Typanmerkung zugewiesen wird. Zweitens enthält der Datentyp „Object“ nicht mehr den Wert `undefined`, der als Standardwert für Object-Instanzen verwendet wurde. Drittens lautet der Standardwert für Instanzen der Object-Klasse in ActionScript 3.0 jetzt `null`.

In früheren Versionen von ActionScript wurde einer Variablen ohne Typanmerkung automatisch der Datentyp „Object“ zugewiesen. Dies findet in ActionScript 3.0 nicht mehr statt, da ActionScript 3.0 jetzt das Konzept einer wahrhaft nicht typisierten Variablen enthält. Variablen ohne Typanmerkung werden jetzt als nicht typisiert betrachtet. Wenn Sie anderen Programmierern, die Ihren Code lesen, deutlich machen möchten, dass Sie eine Variable absichtlich nicht typisiert haben, können Sie das Sternchen-Symbol (`*`) als Typanmerkung verwenden. Dies entspricht dem Weglassen einer Typanmerkung. Im folgenden Beispiel sind zwei gleichwertige Anweisungen dargestellt. Beide deklarieren eine nicht typisierte Variable `x`:

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

Nur nicht typisierte Variable können den Wert `undefined` annehmen. Wenn Sie versuchen, einer Variablen mit einem Datentyp den Wert `undefined` zuzuweisen, wandelt die Laufzeit den Wert `undefined` in den Standardwert dieses Datentyps um. Bei Instanzen des Object-Datentyps lautet der Standardwert `null`. Wenn Sie also versuchen, einer Object-Instanz den Wert `undefined` zuzuweisen, wird der Wert in `null` umgewandelt.

Typumwandlungen

Eine Typumwandlung tritt auf, wenn ein Wert in den Wert eines anderen Datentyps umgewandelt wird. Typumwandlungen können entweder *implizit* oder *explizit* erfolgen. Die implizite Umwandlung, auch als *Erzwingung* bezeichnet, wird manchmal zur Laufzeit durchgeführt. Wenn beispielsweise einer Variablen mit dem Datentyp „Boolean“ der Wert 2 zugeordnet wird, wird der Wert 2 in den booleschen Wert `true` umgewandelt, bevor er der Variablen zugewiesen wird. Eine explizite Umwandlung (auch als *Casting* bezeichnet) tritt auf, wenn der Compiler im Code angewiesen wird, eine Variable eines Datentyps so zu behandeln, als gehöre sie zu einem anderen Datentyp. Sind auch Grundwerte involviert, wandelt Casting die Werte von einem Datentyp in den anderen um. Um ein Objekt in einen anderen Typ umzuwandeln, schließen Sie den Objektnamen in runde Klammern ein und stellen ihm den Namen des neuen Typs voran. Im folgenden Codebeispiel wird ein boolescher Wert in eine Ganzzahl umgewandelt:

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

Implizite Typumwandlungen

Implizite Umwandlungen treten zur Laufzeit in verschiedenen Kontexten auf:

- In Zuweisungsanweisungen

- Wenn Werte als Funktionsargumente übergeben werden
- Wenn Werte von Funktionen zurückgegeben werden
- In Ausdrücken, die bestimmte Operatoren verwenden, z. B. den Additionsoperator (+)

  Implizite Umwandlungen von benutzerdefinierten Typen sind dann erfolgreich, wenn der umzuwandelnde Wert eine Instanz der Zielklasse oder eine von der Zielklasse abgeleitete Klasse ist. Ist eine implizite Umwandlung nicht erfolgreich, tritt ein Fehler auf. Der folgende Code enthält beispielsweise eine erfolgreiche implizite Umwandlung und eine nicht erfolgreiche implizite Umwandlung:

```
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.
```

  Bei Grundtypen werden implizite Umwandlungen durch Aufrufen der gleichen internen Umwandlungsalgorithmen bearbeitet, die auch für explizite Umwandlungen verwendet werden.

Explizite Umwandlungen

Beim Kompilieren im strikten Modus sollten Sie explizite Umwandlungen (Casting) verwenden, da u. U. verhindert werden soll, dass eine Typdiskrepanz einen Kompilierungsfehler erzeugt. Dies ist z. B. der Fall, wenn Sie wissen, dass die Coercion Ihre Werte zur Laufzeit korrekt umwandelt. Wenn Sie beispielsweise mit Daten arbeiten, die von einem Formular übergeben werden, soll die Umwandlung von bestimmten Stringwerten zu numerischen Werten von der Coercion ausgeführt werden. Der folgende Code erzeugt einen Laufzeitfehler, obwohl er im Standardmodus korrekt ausgeführt werden würde:

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

Wenn Sie weiterhin den strikten Modus verwenden, den String aber in eine Ganzzahl umwandeln möchten, können Sie die explizite Umwandlung verwenden. Dazu verwenden Sie folgenden Code:

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

Umwandlung in die Datentypen „int“, „uint“ und „Number“

Sie können jeden Datentyp in einen der drei Datentypen für Zahlen (int, uint und Number) umwandeln. Wenn die Zahl nicht umgewandelt werden kann, wird für die Datentypen „int“ und „uint“ der Standardwert 0 und für den Datentyp „Number“ der Standardwert `NaN` zugewiesen. Wenn Sie einen booleschen Wert in eine Zahl umwandeln, wird `true` in den Wert 1 und `false` in den Wert 0 umgewandelt.

```
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
```

Stringwerte, die nur Ziffern enthalten, können in einen der anderen Datentypen für Zahlen umgewandelt werden. Die Datentypen für Zahlen können auch Strings umwandeln, die wie negative Zahlen aussehen oder einen Hexadezimalwert darstellen (z. B. `0x1A`). Beim Umwandlungsprozess werden vor- und nachgestellte Leerzeichen in Stringwerten ignoriert. Mit `Number()` können Sie auch Strings umwandeln, die Textdarstellungen von Gleitkommazahlen sind. Beim Einfügen eines Dezimalzeichens wird mit `uint()` und `int()` eine Ganzzahl zurückgegeben, bei der die Ziffern und Zeichen hinter der Dezimalstelle abgeschnitten sind. Die folgenden Stringwerte können beispielsweise in Zahlen umgewandelt werden:

```
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
```

Stringwerte, die keine numerischen Zeichen enthalten, geben 0 zurück, wenn sie mit `int()` oder `uint()` umgewandelt werden, und `NaN`, wenn die Umwandlung mit `Number()` erfolgt. Der Umwandlungsprozess ignoriert vor- und nachgestellte Leerzeichen, gibt aber 0 oder `NaN` zurück, wenn die Zeichenfolge Leerstellen enthält, die zwei Zahlen voneinander trennen.

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

In ActionScript 3.0 unterstützt die `Number()`-Funktion keine Oktal- oder Basis 8-Zahlen. Wenn Sie in ActionScript 2.0 eine Zeichenfolge mit einer vorgestellten Null an die `Number()`-Funktion übergeben, wird die Zahl als Oktalzahl interpretiert und in den entsprechenden Dezimalwert umgewandelt. Dies ist in ActionScript 3.0 bei der `Number()`-Funktion nicht der Fall. Hier wird die vorgestellte Null stattdessen ignoriert. Mit dem folgenden Code wird beispielsweise je nach verwendeter ActionScript-Version eine andere Ausgabe erzeugt:

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

Eine Typumwandlung (Casting) ist nicht erforderlich, wenn ein Wert eines numerischen Typs einer Variablen mit einem anderen numerischen Typ zugewiesen wird. Auch im strikten Modus werden numerische Datentypen implizit in andere numerische Datentypen umgewandelt. Dies führt in einigen Fällen dazu, dass sich unerwartete Werte einstellen, wenn der Bereich eines Datentyps überschritten wird. Die folgenden Beispiele werden alle im strikten Modus kompiliert, obwohl einige unerwartete Werte erzeugen:

```
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
```

In der folgenden Tabelle sind die Ergebnisse der Umwandlung von Datentypen in den Datentyp „Number“, „int“ oder „uint“ aufgeführt.

Datentyp oder Wert	Ergebnis der Umwandlung in „Number“, „int“ oder „uint“
Boolean	Wenn der Wert `true` ist, 1; andernfalls 0.
Date	Die interne Darstellung des Date-Objekts gibt die Anzahl der Millisekunden an, die seit dem 1. Januar 1970, 0:00 Uhr Weltzeit verstrichen sind.
`null`	0
Object	Wenn die Instanz `null` lautet und in den Datentyp „Number“ umgewandelt wird, `NaN`; andernfalls 0.
String	Eine Zahl, wenn der String in eine Zahl umgewandelt werden kann; andernfalls `NaN` bei der Umwandlung in den Datentyp „Number“ oder 0 bei der Umwandlung in „int“ oder „uint“.
`undefined`	Wenn in den Datentyp „Number“ umgewandelt wird, `NaN`; wenn in den Datentyp „int“ oder „uint“ umgewandelt wird, 0.

Umwandlung in den Boolean-Datentyp

Durch die Umwandlung eines beliebigen numerischen Datentyps (uint, int und Number) in den Datentyp „Boolean“ wird `false` zurückgegeben, wenn der numerische Wert 0 lautet. Andernfalls wird `true` zurückgegeben. Beim Datentyp „Number“ gibt der Wert `NaN` ebenfalls `false` zurück. Im folgenden Beispiel sind die Ergebnisse der Umwandlung der Zahlen -1, 0 und 1 dargestellt:

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

Im folgenden Beispiel wird veranschaulicht, dass nur eine der drei Zahlen (0) den Wert `false` zurückgibt:

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

Beim Umwandeln eines Stringwerts in einen booleschen Wert wird `false` zurückgegeben, wenn der String `null` lautet oder es sich um einen leeren String (`""`) handelt. Andernfalls wird `true` zurückgegeben.

```
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
```

Bei der Umwandlung der Instanz einer Object-Klasse in den Datentyp „Boolean“ wird `false` zurückgegeben, wenn die Instanz `null` lautet. Andernfalls wird `true` zurückgegeben:

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

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

Variablen des Typs „Boolean“ erfahren im strikten Modus eine Sonderbehandlung, d. h. Sie können einer booleschen Variablen ohne Umwandlung (Casting) Werte jedes Datentyps zuweisen. Die implizite Coercion aus allen Datentypen in den Datentyp „Boolean“ tritt sogar im strikten Modus auf. Anders ausgedrückt, im Gegensatz zu fast allen anderen Datentypen ist eine Umwandlung in den Datentyp „Boolean“ nicht erforderlich, um Fehler im strikten Modus zu vermeiden. Die folgenden Beispiele werden alle im strikten Modus kompiliert und verhalten sich zur Laufzeit wie erwartet:

```
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
```

In der folgenden Tabelle sind die Ergebnisse der Umwandlung eines Datentyps in den Datentyp „Boolean“ aufgeführt:

Datentyp oder Wert	**Ergebnis der Umwandlung in den Datentyp „Boolean“**
String	`false`, wenn der Wert `null` lautet oder eine leere Zeichenfolge ist (`""`); andernfalls `true`.
`null`	`false`
Number, int oder uint	`false`, wenn der Wert `NaN` oder 0 ist, andernfalls `true`.
Object	`false`, wenn die Instanz `null` ist, andernfalls `true`.

Umwandlung in einen String-Datentyp

Durch die Umwandlung eines beliebigen numerischen Datentyps in den Datentyp „String“ wird eine Stringdarstellung der Zahl zurückgegeben. Die Umwandlung eines booleschen Werts in den Datentyp „String“ gibt den String `true` zurück, wenn der Wert `true` lautet, oder den String `false`, wenn der Wert `false` lautet.

Die Umwandlung einer Instanz der Object-Klasse in den Datentyp „String“ gibt den String `null` zurück, wenn die Instanz `null` lautet. Andernfalls gibt die Umwandlung einer Object-Klasse in den Datentyp „String“ den String `[object Object]` zurück.

Die Umwandlung einer Instanz der Array-Klasse in den Datentyp „String“ gibt einen String zurück, der aus einer kommagetrennten Liste aller Array-Elemente besteht. Beispielsweise gibt die folgende Umwandlung in den Datentyp „String“ einen String zurück, der alle drei Elemente des folgenden Arrays enthält:

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

Die Umwandlung einer Instanz der Date-Klasse in den Datentyp „String“ gibt eine Stringdarstellung des Datums zurück, das die Instanz enthält. Im folgenden Code wird eine Stringdarstellung der Date-Klasseninstanz zurückgegeben (die Ausgabe zeigt das Ergebnis für die Pacific Daylight Time):

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

In der folgenden Tabelle sind die Ergebnisse der Umwandlung eines Datentyps in den Datentyp „String“ aufgeführt:

Datentyp oder Wert	Ergebnis der Umwandlung in den Datentyp „String“
Array	Ein String, der aus allen Array-Elementen besteht.
Boolean	`"true"` oder `"false"`
Date	Die Stringdarstellung des Date-Objekts.
`null`	`"null"`
Number, int oder uint	Die Stringdarstellung der Zahl.
Object	Wenn die Instanz „null“ ist, `null`; andernfalls `[object Object]`.

Syntax

Die Syntax einer Sprache definiert einen Regelsatz, der beim Schreiben von ausführbarem Code eingehalten werden muss.

Groß-/Kleinschreibung

ActionScript 3.0 unterscheidet zwischen Groß- und Kleinschreibung. Bezeichner, die sich in der Groß-/Kleinschreibung unterscheiden, werden als unterschiedliche Bezeichner angesehen. Im folgenden Code werden beispielsweise zwei verschiedene Variable erstellt:

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

Punktsyntax

Der Punktoperator (.) ermöglicht den Zugriff auf die Eigenschaften und Methoden eines Objekts. Mit der Punktsyntax können Sie mit einem Instanznamen, gefolgt vom Punktoperator und dem Namen der Methode oder Eigenschaft, auf eine Eigenschaft der Klasse oder Methode verweisen. Betrachten Sie die folgende Klassendefinition:

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

Mit der Punktsyntax können Sie mit dem Instanznamen auf die `prop1`-Eigenschaft und die `method1()`-Methode zugreifen. Der Instanzname wird im folgenden Code erstellt:

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

Sie können die Punktsyntax beim Definieren von Paketen verwenden. Sie können den Punktoperator für Verweise auf verschachtelte Pakete verwenden. Beispielsweise befindet sich die EventDispatcher-Klasse in einem Paket namens „events“, das wiederum in einem Paket namens „flash“ verschachtelt ist. Der Verweis auf das events-Paket erfolgt mit dem folgenden Ausdruck:

```
flash.events
```

Der Verweis auf die EventDispatcher-Klasse erfolgt mithilfe dieses Ausdrucks:

```
flash.events.EventDispatcher
```

Schrägstrichsyntax

Die Schrägstrichsyntax wird in ActionScript 3.0 nicht unterstützt. Sie wurde in früheren Versionen von ActionScript verwendet, um den Pfad eines Movieclips oder einer Variablen anzugeben.

Literale

Ein *Literal* ist ein Wert, der direkt im Code angezeigt wird. Bei den folgenden Beispielen handelt es sich um Literale:

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

Literale können auch zu zusammengesetzten Literalen zusammengefasst werden. Array-Literale werden in eckige Klammern (`[]`) eingeschlossen. Zum Trennen mehrerer Array-Elemente wird ein Komma verwendet.

Ein Array-Literal kann zum Initialisieren eines Arrays verwendet werden. Die folgenden Beispiele zeigen zwei Arrays, die mit Array-Literalen initialisiert werden. Mit der `new`-Anweisung können Sie das zusammengesetzte Literal als Parameter an den Konstruktor der Array-Klasse übergeben. Es ist jedoch auch möglich, Literalwerte beim Erstellen von Instanzen der ActionScript-Hauptklassen (Object, Array, String, Number, int, uint, XML, XMLList und Boolean) direkt zuzuweisen.

```
// 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];
```

Literale können auch zum Initialisieren generischer Objekte verwendet werden. Ein generisches Objekt ist eine Instanz der Object-Klasse. Objektliterale werden in geschweifte Klammern (`{}`) eingeschlossen. Zum Trennen mehrerer Objekteigenschaften wird ein Komma verwendet. Jede Eigenschaft wird mit einem Doppelpunkt (`:`) deklariert, der den Namen der Eigenschaft von ihrem Wert trennt.

Ein generisches Objekt erstellen Sie mit der Anweisung `new` und übergeben das Objekt-Literal als Parameter an den Konstruktor der Object-Klasse. Es ist jedoch auch möglich, das Objekt-Literal der deklarierten Instanz direkt zuzuweisen. Im folgenden Beispiel werden zwei alternative Methoden veranschaulicht, um ein neues generisches Objekt zu erstellen und das Objekt mit drei Eigenschaften (`propA`, `propB` und `propC`) und den Werten 1, 2 und 3 zu initialisieren:

```
// 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};
```

Verwandte Themen

Arbeiten mit Strings

Verwenden von regulären Ausdrücken

Initialisieren von XML-Variablen

Semikola

Mit dem Semikolon (`;`) beenden Sie eine Anweisung. Wenn Sie das Semikolon weglassen, geht der Compiler davon aus, dass jede Codezeile eine einzelne Anweisung darstellt. Viele Programmierer haben sich angewöhnt, das Ende einer Anweisung mit einem Semikolon zu kennzeichnen. Ihr Code wird leichter lesbar, wenn Sie zum Beenden Ihrer Anweisungen konsistent Semikola verwenden.

Durch die Verwendung von Semikola können Sie zwar mehrere Anweisungen in eine Codezeile platzieren, jedoch wirkt sich dies meist negativ auf die Lesbarkeit des Codes aus.

Runde Klammern

Sie können runde Klammern (`()`) in ActionScript 3.0 auf drei Arten verwenden. Zum einen können Sie mithilfe von runden Klammern die Reihenfolge von Operationen in einer Anweisung ändern. Operationen, die zwischen runden Klammern gruppiert sind, werden immer zuerst ausgeführt. Runde Klammern können beispielsweise verwendet werden, um die Reihenfolge der Operationen im folgenden Code zu ändern:

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

Dann können Sie mit runden Klammern und dem Kommaoperator (`,`) eine Reihe von Ausdrücken auswerten und das Ergebnis des endgültigen Ausdrucks zurückzugeben. Diese Anwendung wird im folgenden Beispiel gezeigt:

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

Schließlich können Sie mit runden Klammern einen oder mehrere Parameter an Funktionen oder Methoden übergeben. Diese Anwendung wird im folgenden Code vorgestellt, in dem ein Stringwert an die `trace()`-Funktion übergeben wird:

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

Kommentare

ActionScript 3.0-Code unterstützt zwei Arten von Kommentaren: einzeilige und mehrzeilige Kommentare. Die Kommentarmechanismen ähneln denen in C++ und Java. Der Compiler ignoriert Text, der als Kommentar gekennzeichnet ist.

Einzeilige Kommentare beginnen mit zwei Schrägstrichen (`//`) und werden dann bis zum Ende der Zeile fortgeführt. Der folgende Code enthält einen einzeiligen Kommentar:

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

Mehrzeilige Kommentare beginnen mit einem Schrägstrich und einem Sternchen (`/*`) und enden mit einem Sternchen und einem Schrägstrich (`*/`).

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

Schlüsselwörter und reservierte Wörter

Reservierte Wörter können im Code nicht als Bezeichner verwendet werden, da sie für die Verwendung durch ActionScript reserviert sind. Reservierte Wörter umfassen *lexikalische Schlüsselwörter*, die vom Compiler aus dem Namespace des Programms entfernt werden. Wenn Sie ein lexikalisches Schlüsselwort als Bezeichner verwenden, meldet der Compiler einen Fehler. In der folgenden Tabelle sind die lexikalischen Schlüsselwörter in ActionScript 3.0 aufgeführt:

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

Es gibt noch eine kleine Gruppe von Schlüsselwörtern, die so genannten *syntaktischen Schlüsselwörter*, die als Bezeichner verwendet werden können, in bestimmten Kontexten jedoch eine spezielle Bedeutung haben. In der folgenden Tabelle sind die syntaktischen Schlüsselwörter von ActionScript 3.0 aufgeführt:

each	get	set	namespace
include	dynamic	final	nativ
override	static		

Darüber hinaus gibt es verschiedene Bezeichner, die manchmal als *für zukünftige Verwendung reservierte Wörter* bezeichnet werden. Diese Bezeichner sind zwar nicht durch ActionScript 3.0 reserviert, einige dieser Wörter können jedoch von Software, die ActionScript 3.0 integriert, als Schlüsselwörter behandelt werden. Zahlreiche dieser Bezeichnung können problemlos in Ihrem Code verwendet werden, Adobe rät jedoch von deren Verwendung ab, da sie in einer der nachfolgenden Versionen der Programmiersprache evtl. als Schlüsselwörter genutzt werden könnten.

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

Konstanten

ActionScript 3.0 unterstützt die `const`-Anweisung, mit der Sie Konstanten erstellen können. Konstante sind Eigenschaften mit einem festen Wert, der nicht geändert werden kann. Einer Konstanten wird nur einmal ein Wert zugewiesen, und die Zuweisung muss in unmittelbarer Nähe zur Deklaration der Konstanten erfolgen. Wenn Sie eine Konstante beispielsweise als Mitglied einer Klasse deklarieren, können Sie der Konstanten einen Wert nur als Teil der Deklaration oder innerhalb des Klassenkonstruktors zuweisen.

Im folgenden Code werden zwei Konstanten deklariert. Der ersten Konstante (`MINIMUM`) wird als Teil der Deklarationsanweisung ein Wert zugewiesen. Der zweiten Konstante (`MAXIMUM`) wird im Konstruktor ein Wert zugewiesen. Beachten Sie, dass dieses Beispiel nur im Standardmodus kompiliert wird. Im strikten Modus kann der Wert einer Konstanten nur bei der Initialisierung zugewiesen werden.

```
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
```

Der Versuch, einer Konstanten mit einem anderen Verfahren einen Ursprungswert zuzuweisen, führt zu einer Fehlermeldung. Wenn Sie beispielsweise versuchen, den Ursprungswert von `MAXIMUM` außerhalb der Klasse zuzuweisen, tritt ein Laufzeitfehler auf.

```
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 definiert eine Vielzahl von Konstanten. Üblicherweise werden Konstanten in ActionScript in Großbuchstaben geschrieben, und einzelne Wörter werden durch einen Unterstrich (_) voneinander getrennt. Die MouseEvent-Klassendefinition verwendet beispielsweise diese Namenskonvention für ihre Konstanten. Dabei steht jede Konstante für ein Ereignis, das einer Mauseingabe zugeordnet ist:

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

Operatoren

Operatoren sind spezielle Funktionen, die mindestens einen Operanden umfassen und einen Wert zurückgeben. Ein *Operand* ist ein Wert (in der Regel ein Literal, eine Variable oder ein Ausdruck), den ein Operator als Eingabe verwendet. Im folgenden Beispielcode werden der Additionsoperator (`+`) und der Multiplikationsoperator (`*`) mit drei literalen Operanden (`2`, `3` und `4`) verwendet, um einen Wert zurückzugeben. Dieser Wert wird dann vom Zuweisungsoperator (`=`) verwendet, um den zurückgegebenen Wert (14) der Variablen `sumNumber` zuzuweisen.

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

Operatoren können unär, binär oder ternär sein. Ein *unärer* Operator umfasst einen Operanden. Ein Beispiel für einen unären Operator ist der Inkrementoperator (`++`), da er nur einen Operanden hat. Ein *binärer* Operator umfasst zwei Operanden. Ein Beispiel ist der Divisionsoperator (`/`, der zwei Operanden umfasst. Ein *ternärer* Operator umfasst drei Operanden. Ein Beispiel für diesen Operator ist der Bedingungsoperator (`?:`, der drei Operanden hat.

Einige Operatoren sind *überladen*, d. h. dass sie sich je nach Typ oder Anzahl der übergebenen Operanden unterschiedlich verhalten. Ein Beispiel für einen überladenen Operator ist der Additionsoperator (`+`), da er sich je nach Datentyp der Operanden unterschiedlich verhält. Handelt es sich bei beiden Operanden um Zahlen, liefert der Additionsoperator die Summe der beiden Werte. Sind beide Operanden Strings, gibt der Additionsoperator eine Verkettung der beiden Operanden zurück. Im folgenden Beispielcode wird gezeigt, wie sich der Operator je nach Operanden unterschiedlich verhält:

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

Operatoren können sich auch je nach Anzahl der angegebenen Operanden unterschiedlich verhalten. Der Subtraktionsoperator (`-`) ist sowohl ein unärer als auch ein binärer Operator. Wenn nur ein Operand angegeben ist, negiert der Subtraktionsoperator den Operanden und gibt das Ergebnis zurück. Sind zwei Operanden angegeben, gibt der Subtraktionsoperator die Differenz zwischen den Operanden zurück. Im folgenden Beispiel wird der Subtraktionsoperator gezeigt, wie er zuerst als unärer Operator und dann als binärer Operator verwendet wird.

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

Rangfolge und Assoziativität von Operatoren

Rangfolge und Assoziativität der Operatoren bestimmen die Reihenfolge, in der die Operatoren verarbeitet werden. Obwohl es für Personen mit arithmetischen Kenntnissen selbstverständlich erscheinen mag, dass der Compiler den Multiplikationsoperator (`*`) vor dem Additionsoperator (`+`) ausführt, braucht der Compiler konkrete Anweisungen dafür, in welcher Reihenfolge Operatoren verarbeitet werden sollen. Diese Anweisungen werden zusammenfassend als *Operatorrangfolge* bezeichnet. In ActionScript gilt eine Standardrangfolge für Operatoren, die Sie jedoch mithilfe von runden Klammern (`()`) ändern können. Beispielsweise wird mit dem folgenden Code die Standardrangfolge aus dem vorangegangenen Beispiel geändert, um den Compiler zu zwingen, den Additionsoperator vor dem Multiplikationsoperator auszuführen:

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

In manchen Fällen können sich zwei oder mehr Operatoren mit derselben Rangfolge im gleichen Ausdruck befinden. In diesen Fällen ermittelt der Compiler anhand der Regeln der *Assoziativität*, welcher Operator zuerst verarbeitet wird. Für alle binären Operatoren, mit Ausnahme der Zuweisungsoperatoren, gilt die *Linksassoziativität*, d. h. die Operatoren auf der linken Seite werden vor den Operatoren auf der rechten Seite verarbeitet. Für die Zuweisungsoperatoren und den Bedingungsoperator (`?:`) gilt die *Rechtsassoziativität*, sodass die Operatoren auf der rechten Seite vor den Operatoren auf der linken Seite verarbeitet werden.

Der Kleiner als-Operator (<) und der Größer als-Operator (>) haben beispielsweise dieselbe Rangfolge. Sind beide Operatoren im gleichen Ausdruck enthalten, wird der linke Operator zuerst verarbeitet, da für beide Operatoren die Linksassoziativität gilt. Die beiden folgenden Anweisungen generieren somit dieselbe Ausgabe:

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

Der Größer als-Operator wird zuerst verarbeitet. Als Ergebnis entsteht der Wert `true`, da der Operand 3 größer ist als der Operand 2. Dann wird der Wert `true` mit dem Operanden 1 an den Kleiner als-Operator übergeben. Dieser Zwischenzustand ist im folgenden Code dargestellt:

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

Der Kleiner als-Operator wandelt den Wert `true` in den numerischen Wert 1 um und vergleicht diesen numerischen Wert mit dem zweiten Operanden 1. Es wird der Wert `false` zurückgegeben, da der Wert 1 nicht kleiner ist als 1.

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

Sie können die standardmäßige Linksassoziativität mit dem Klammernoperator ändern. Durch Einschließen des Operators und seiner Operanden in runden Klammern können Sie den Compiler anweisen, zuerst den Kleiner als-Operator zu verarbeiten. Im folgenden Beispielcode wird der Klammernoperator eingesetzt, um bei Verwendung der Zahlen aus dem vorangegangenen Beispiel eine andere Ausgabe zu erzeugen:

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

Der Kleiner als-Operator wird zuerst verarbeitet. Als Ergebnis entsteht der Wert `false`, da der Operand 2 nicht kleiner ist als der Operand 1. Dann wird der Wert `false` mit dem Operanden 3 an den Größer als-Operator übergeben. Dieser Zwischenzustand ist im folgenden Code dargestellt:

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

Der Größer als-Operator wandelt den Wert `false` in den numerischen Wert 0 um und vergleicht diesen numerischen Wert mit dem zweiten Operanden 3. Es wird der Wert `true` zurückgegeben, da der Wert 3 größer ist als 0.

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

In der folgenden Tabelle sind die Operatoren für ActionScript 3.0 in absteigender Rangfolge aufgeführt. Jede Tabellenzeile enthält Operatoren der gleichen Rangfolge. Jede Operatorenzeile hat einen höheren Rang als die Zeile, die darunter folgt.

Gruppe	**Operatoren**
Primär	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Suffix	`x++ x--`
Unär	`++x --x + - ~ ! delete typeof void`
Multiplikativ	`* / %`
Additiv	`+ -`
Bitweise Verschiebung	`<< >> >>>`
Relational	`< > <= >= as in instanceof is`
Gleichheit	`== != === !==`
Bitweises AND	`&`
Bitweises XOR	`^`
Bitweises OR	`|`

Gruppe	Operatoren
Logisches AND	`&&`
Logisches OR	`||`
Bedingung	`?:`
Zuweisung	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Komma	`,`

Primäre Operatoren

Die primären Operatoren umfassen die Operatoren zum Erstellen von Array- und Object-Literalen, zum Gruppieren von Ausdrücken, zum Aufrufen von Funktionen, zum Instanziieren von Klasseninstanzen und zum Zugreifen auf Eigenschaften.

Alle in der folgenden Tabelle aufgeführten primären Operatoren haben dieselbe Rangfolge. Operatoren, die Teil der E4X-Spezifikation sind, werden durch die Notation (E4X) gekennzeichnet.

Operator	Aktion
`[]`	Initialisiert ein Array
`{x:y}`	Initialisiert ein Objekt
`()`	Gruppiert Ausdrücke
`f(x)`	Ruft eine Funktion auf
`new`	Ruft einen Konstruktor auf
`x.y x[y]`	Greift auf eine Eigenschaft zu
`<></>`	Initialisiert ein XMLList-Objekt (E4X)
`@`	Greift auf ein Attribut zu (E4X)
`::`	Qualifiziert einen Namen (E4X)
`..`	Greift auf ein XML-Nachfolgeelement zu (E4X)

Suffix-Operatoren

Die Suffix-Operatoren umfassen einen Operator und erhöhen oder verringern dessen Wert. Diese Operatoren sind zwar unär, werden jedoch nicht zusammen mit den anderen unären Operatoren klassifiziert, da sie eine höhere Rangfolge und ein besonderes Verhalten aufweisen. Wenn Sie einen Suffix-Operator in einem größeren Ausdruck verwenden, wird der Wert des Ausdrucks noch vor der Verarbeitung des Suffix-Operators zurückgegeben. Beispielsweise wird im folgenden Code der Wert des Ausdrucks `xNum++` zurückgegeben, bevor der Wert erhöht wird:

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

Alle in der folgenden Tabelle aufgeführten Suffix-Operatoren haben dieselbe Rangfolge:

Operator	Aktion
`++`	Inkrementiert (Suffix)
`--`	Dekrementiert (Suffix)

Unäre Operatoren

Unäre Operatoren umfassen einen Operanden. Die Operatoren zum Inkrementieren (`++`) und Dekrementieren (`--`) in dieser Gruppe sind *Präfix-Operatoren*, da sie in einem Ausdruck vor dem Operanden stehen. Präfix-Operatoren unterscheiden sich insofern von ihren Gegenstücken, den Suffix-Operatoren, als dass das Inkrementieren oder Dekrementieren abgeschlossen ist, bevor der Wert des Gesamtausdrucks zurückgegeben wird. Im folgenden Beispielcode wird der Wert des Ausdrucks `++xNum` zurückgegeben, nachdem dieser inkrementiert wurde:

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

Alle in der folgenden Tabelle aufgeführten unären Operatoren haben dieselbe Rangfolge:

Operator	Aktion
`++`	Inkrementiert (Präfix)
`--`	Dekrementiert (Präfix)
`+`	Unär +
`-`	Unär - (Negation)
`!`	Logisches NOT
`~`	Bitweises NOT
`delete`	Löscht eine Eigenschaft.
`typeof`	Gibt Typinformationen zurück
`void`	Gibt einen undefinierten Wert zurück

Multiplikative Operatoren

Multiplikative Operatoren nehmen mit zwei Operanden Multiplikationen, Divisionen und Restwertberechnungen vor.

Alle in der folgenden Tabelle aufgeführten multiplikativen Operatoren haben dieselbe Rangfolge:

Operator	Aktion
`*`	Multiplikation
`/`	Division
`%`	Restwert

Additionsoperatoren

Additionsoperatoren umfassen zwei Operanden und führen Additionen und Subtraktionen durch. Alle in der folgenden Tabelle aufgeführten Additionsoperatoren haben dieselbe Rangfolge:

Operator	Aktion
+	Addition
-	Subtraktion

Operatoren für bitweise Verschiebung

Die Operatoren für bitweise Verschiebung haben zwei Operanden. Sie verschieben die Bits des ersten Operanden um den Wert, der mit dem zweiten Operanden angegeben wird. Alle in der folgenden Tabelle aufgeführten Operatoren für bitweise Verschiebung haben dieselbe Rangfolge:

Operator	Aktion
<<	Bitweise Verschiebung nach links
>>	Bitweise Verschiebung nach rechts
>>>	Vorzeichenlose bitweise Verschiebung nach rechts

Relationale Operatoren

Relationale Operatoren umfassen zwei Operanden, vergleichen deren Werte und geben einen booleschen Wert zurück. Alle in der folgenden Tabelle aufgeführten relationalen Operatoren haben dieselbe Rangfolge:

Operator	Aktion
<	Kleiner als
>	Größer als
<=	Kleiner als oder gleich
>=	Größer als oder gleich
as	Überprüft den Datentyp
in	Überprüft Objekteigenschaften
instanceof	Überprüft die Prototypkette
is	Überprüft den Datentyp

Gleichheitsoperatoren

Gleichheitsoperatoren umfassen zwei Operanden, vergleichen deren Werte und geben einen booleschen Wert zurück. Alle in der folgenden Tabelle aufgeführten Gleichheitsoperatoren haben dieselbe Rangfolge:

Operator	Aktion
==	Gleichheit
!=	Ungleichheit
===	Strikte Gleichheit
!==	Strikte Ungleichheit

Bitweise logische Operatoren

Die bitweisen logischen Operatoren umfassen zwei Operanden und führen logische Operationen auf Bit-Ebene aus. Die bitweisen logischen Operatoren unterscheiden sich hinsichtlich der Rangfolge. In der folgenden Tabelle sind sie in absteigender Rangfolge aufgeführt:

Operator	Aktion
`&`	Bitweises AND
`^`	Bitweises XOR
`|`	Bitweises OR

Logische Operatoren

Die logischen Operatoren umfassen zwei Operanden und geben einen booleschen Ergebniswert zurück. Die logischen Operatoren unterscheiden sich hinsichtlich der Rangfolge. In der folgenden Tabelle sind sie in absteigender Rangfolge aufgeführt:

Operator	Aktion
`&&`	Logisches AND
`||`	Logisches OR

Bedingungsoperator

Der Bedingungsoperator ist ternär, d. h. er umfasst drei Operanden. Mit dem Bedingungsoperator lässt sich die Bedingungsanweisung `if..else` auf verkürzte Weise anwenden.

Operator	Aktion
`?:`	Bedingung

Zuweisungsoperatoren

Zuweisungsoperatoren haben zwei Operanden. Sie weisen einem Operanden einen Wert zu, der auf dem Wert des anderen Operanden basiert. Alle in der folgenden Tabelle aufgeführten Zuweisungsoperatoren haben dieselbe Rangfolge:

Operator	Aktion
`=`	Zuweisung
`*=`	Multiplikationszuweisung
`/=`	Divisionszuweisung
`%=`	Modulo-Zuweisung (Restwert)
`+=`	Additionszuweisung
`-=`	Subtraktionszuweisung
`<<=`	Zuweisung einer bitweisen Verschiebung nach links
`>>=`	Zuweisung einer bitweisen Verschiebung nach rechts

Operator	Aktion
>>>=	Zuweisung einer vorzeichenlosen bitweisen Verschiebung nach rechts
&=	Zuweisung von bitweisem AND
^=	Zuweisung von bitweisem XOR
|=	Zuweisung von bitweisem OR

Bedingungen

ActionScript 3.0 umfasst drei Bedingungsanweisungen, mit denen Sie den Programmablauf steuern können.

if..else

Mit der Bedingungsanweisung `if..else` können Sie eine Bedingung testen und dann in Abhängigkeit davon, ob die Bedingung erfüllt ist, unterschiedliche Codeblocks ausführen. Im folgenden Beispielcode wird getestet, ob der Wert von `x` größer ist als 20. Ist dies der Fall, wird eine `trace()`-Funktion generiert. Andernfalls wird eine andere `trace()`-Funktion generiert:

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

Wenn kein alternativer Codeblock ausgeführt werden soll, können Sie die Anweisung `if` auch ohne die Anweisung `else` verwenden.

if..else if

Mit der Bedingungsanweisung `if..else if` lassen sich mehrere Bedingungen überprüfen. Im folgenden Beispielcode wird getestet, ob der Wert `x` größer ist als 20 und ob `x` einen negativen Wert hat:

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

Wenn einer `if`- oder `else`-Anweisung nur eine Anweisung folgt, muss die Anweisung nicht in geschweifte Klammern eingeschlossen werden. Im folgenden Code werden beispielsweise keine geschweiften Klammern verwendet:

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

Dennoch sollten Sie stets geschweifte Klammern verwenden, da andernfalls ein unerwartetes Verhalten auftreten kann, wenn später Anweisungen zu einer Bedingungsanweisung hinzugefügt werden, die keine geschweiften Klammern enthält. Im folgenden Beispielcode wird Wert von `positiveNums` um 1 erhöht, unabhängig davon, ob die Bedingung als `true` ausgewertet wird:

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

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

trace(positiveNums); // 1
```

switch

Die `switch`-Anweisung eignet sich insbesondere dann, wenn mehrere Ausführungspfade vom gleichen Bedingungsausdruck abhängen. Sie bietet eine ähnliche Funktionalität wie eine lange Folge von `if..else if`-Anweisungen, die Lesbarkeit ist jedoch besser. Anstatt eine Bedingung auf einen booleschen Wert zu testen, wertet die `switch`-Anweisung einen Ausdruck aus und ermittelt den auszuführenden Codeblock anhand des Ergebnisses. Codeblöcke beginnen mit einer `case`-Anweisung und enden mit einer `break`-Anweisung. In der folgenden `switch`-Anweisung wird anhand der von der `Date.getDay()`-Methode zurückgegebenen Zahl der entsprechende Wochentag ausgegeben:

```
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;
}
```

Schleifen

Mit Schleifenanweisungen können Sie einen bestimmten Codeblock mit einer Reihe von Werten oder Variablen wiederholt ausführen. Sie sollten den Codeblock immer in geschweifte Klammern ({}) setzen. Wenn der Codeblock nur eine Anweisung enthält, können Sie die geschweiften Klammern theoretisch auch weglassen. Hiervon wird jedoch abgeraten, da dies, ähnlich wie bei Bedingungen, dazu führen kann, dass später hinzugefügte Anweisungen versehentlich aus dem Codeblock ausgeschlossen werden. Wenn Sie später eine Anweisung hinzufügen, die in den Codeblock aufgenommen werden soll, jedoch die notwendigen geschweiften Klammern vergessen, wird die Anweisung nicht als Teil der Schleife ausgeführt.

for

Mit der `for`-Schleife können Sie eine Variable in einem bestimmten Wertebereich durchlaufen. Sie müssen drei Ausdrücke in einer `for`-Anweisung angeben: eine Variable, die auf einen Ausgangswert eingestellt ist, eine Bedingungsanweisung, die das Ende der Schleife bestimmt, und einen Ausdruck, der den Wert der Variablen bei jedem Durchlauf ändert. Der Code im folgenden Beispiel wird fünf Mal ausgeführt. Die `i`-Variable hat am Anfang den Wert 0 und endet mit dem Wert 4. Die Ausgabe besteht aus den Zahlen 0 bis 4, die jeweils in einer separaten Zeile angezeigt werden.

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

for..in

Die `for..in`-Schleife durchläuft die Eigenschaften eines Objekts oder die Elemente eines Arrays. Sie können eine `for..in`-Schleife beispielsweise verwenden, um die Eigenschaften eines generischen Objekts zu durchlaufen (da für Objekteigenschaften keine feste Reihenfolge gilt, werden sie in willkürlicher Reihenfolge angezeigt):

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

Sie können auch die Elemente eines Arrays durchlaufen:

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

Dagegen ist es nicht möglich, die Eigenschaften eines Objekts zu durchlaufen, wenn es sich um eine Instanz einer versiegelten Klasse handelt (einschließlich integrierter und benutzerdefinierter Klassen). Sie können nur die Eigenschaften einer dynamischen Klasse durchlaufen. Und selbst bei Instanzen von dynamischen Klassen können Sie nur die Eigenschaften durchlaufen, die dynamisch hinzugefügt werden.

for each..in

Die `for each..in`-Schleife durchläuft die Objekte einer Sammlung. Bei diesen Objekten kann es sich um Tags in einem XML- oder XMLList-Objekt, um die in Objekteigenschaften gespeicherten Werte oder um die Elemente eines Arrays handeln. Wie im folgenden Codeauszug zu sehen ist, können Sie eine `for each..in`-Schleife beispielsweise verwenden, um die Eigenschaften eines generischen Objekts zu durchlaufen. Im Gegensatz zur `for..in`-Schleife enthält die Iterationsvariable in einer `for each..in`-Schleife den in der Eigenschaft gespeicherten Wert anstelle des Eigenschaftsnamens:

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

Sie können ein XML- oder XMLList-Objekt durchlaufen. Dies wird im folgenden Code gezeigt:

```
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
*/
```

Sie können auch die Elemente eines Arrays durchlaufen, wie das folgende Beispiel zeigt:

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

Die Eigenschaften eines Objekts, das eine Instanz einer versiegelten Klasse ist, können nicht durchlaufen werden. Auch bei Instanzen von dynamischen Klassen können Sie keine festen Eigenschaften durchlaufen, die als Teil der Klassendefinition definiert sind.

while

Die `while`-Schleife ähnelt einer `if`-Anweisung, da sie wiederholt wird, solange die Bedingung `true` ist. Im folgenden Codebeispiel wird die gleiche Ausgabe wie bei der `for`-Schleife erzeugt:

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

Das Verwenden von `while`-Schleifen hat im Gegensatz zu `for`-Schleifen den Vorteil, dass sich Endlosschleifen leichter mit `while`-Schleifen schreiben lassen. Der Code einer `for`-Schleife wird nicht kompiliert, wenn der Ausdruck fehlt, mit dem die Zählervariable erhöht wird. Der Code einer `while`-Schleife wird jedoch auch ohne diesen Ausdruck kompiliert. Wenn der Ausdruck zum Erhöhen von `i` nicht vorhanden ist, entsteht eine Endlosschleife.

do..while

Die `do..while`-Schleife ist eine `while`-Schleife, die garantiert, dass ein Codeblock mindestens einmal ausgeführt wird, da die Bedingung erst nach dem Ausführen des Codeblocks überprüft wird. Das folgende Codebeispiel zeigt eine einfache `do..while`-Schleife, durch die eine Ausgabe generiert wird, obwohl die Bedingung nicht erfüllt ist:

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

Funktionen

Funktionen sind Codeblöcke, die bestimmte Aufgaben ausführen und an anderer Stelle in einem Programm wiederverwendet werden können. In ActionScript 3.0 wird zwischen zwei Funktionstypen unterschieden: *Methoden* und *Funktionshüllen*. Ob eine Funktion als eine Methode oder Funktionshülle bezeichnet wird, hängt von dem Kontext ab, in dem die Funktion definiert ist. Eine Funktion wird als eine Methode bezeichnet, wenn Sie sie als Teil der Klassendefinition definieren oder an eine Instanz eines Objekts anhängen. Eine Funktion wird als ein Funktionshülle bezeichnet, wenn sie auf eine andere Weise definiert wurde.

Funktionen waren in ActionScript schon immer extrem wichtig. In ActionScript 1.0 war beispielsweise das Schlüsselwort `class` noch nicht bekannt, daher wurden „Klassen“ durch Konstruktorfunktionen definiert. Obwohl das Schlüsselwort `class` der Programmsprache inzwischen hinzugefügt wurde, ist noch immer ein solides Grundwissen über Funktionen wichtig, wenn Sie alle Vorteile der Sprache nutzen möchten. Dies kann eine schwierige Aufgabe für Programmierer sein, die davon ausgehen, dass sich Funktionen in ActionScript ähnlich wie in anderen Sprachen, z. B. C++ oder Java, verhalten. Obwohl das allgemeine Definieren und Aufrufen von Funktionen keine Herausforderung für einen erfahrenen Programmierer darstellt, erfordern einige erweiterte ActionScript-Funktionen doch eine Erklärung.

Grundfunktionen

Aufrufen von Funktionen

Eine Funktion wird mit ihrem Bezeichner und anschließender Eingabe des Klammernoperators (`()`) aufgerufen. Der Klammernoperator nimmt alle Funktionsparameter auf, die Sie an die Funktion übergeben möchten. Beispielsweise ist `trace()` eine Funktion der obersten Ebene in ActionScript 3.0:

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

Wenn Sie eine Funktion ohne Parameter aufrufen, müssen Sie ein leeres Paar runder Klammern angeben. Zum Erzeugen einer Zufallszahl können Sie beispielsweise die `Math.random()`-Methode verwenden, die ohne Parameter arbeitet:

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

Definieren eigener Funktionen

Funktionen können in ActionScript 3.0 auf zwei Arten definiert werden: mit einer Funktionsanweisung oder einem Funktionsausdruck. Die von Ihnen gewählte Technik hängt davon ab, ob Sie einen statischen oder dynamischen Programmierstil bevorzugen. Wenn Sie das statische Programmieren bzw. den strikten Modus bevorzugen, definieren Sie Ihre Funktionen mit Funktionsanweisungen. Definieren Sie Ihre Funktionen mit Funktionsausdrücken, wenn dies aus bestimmten Gründen erforderlich ist. Funktionsausdrücke werden häufiger bei der dynamischen Programmierung bzw. im Standardmodus eingesetzt.

Funktionsanweisungen

Funktionsanweisungen stellen die bevorzugte Technik bei der Definition von Funktionen im strikten Modus dar. Eine Funktionsanweisung beginnt mit dem Schlüsselwort `function`, gefolgt von:

- dem Funktionsnamen
- den in runde Klammern eingeschlossenen Parametern in einer durch Kommas getrennten Liste
- dem Funktionsrumpf, d. h. dem nach dem Aufrufen der Funktion auszuführenden ActionScript-Code, der in geschweifte Klammern eingeschlossen ist

Im folgenden Beispielcode wird eine Funktion erstellt, die einen Parameter definiert und dann die Funktion mit dem String „ `hello`“ als Parameterwert aufruft:

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

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

Funktionsausdrücke

Das zweite Verfahren zum Deklarieren einer Funktion verwendet eine Funktionsanweisung mit einem Funktionsausdruck, der manchmal auch als ein Funktionsliteral oder als eine anonyme Funktion bezeichnet wird. Dies ist eine wesentlich ausführlichere Methode, die häufig in früheren Versionen von ActionScript verwendet wurde.

Eine Funktionsanweisung mit einem Funktionsausdruck beginnt mit dem Schlüsselwort `var`, gefolgt von:

- dem Funktionsnamen
- dem Doppelpunktoperator (`:`)
- der `Function`-Klasse, um den Datentyp festzulegen

- dem Zuweisungsoperator (=)
- dem Schlüsselwort `function`
- den in runde Klammern eingeschlossenen Parametern in einer durch Kommas getrennten Liste
- dem Funktionsrumpf, d. h. dem nach dem Aufrufen der Funktion auszuführenden ActionScript-Code, der in geschweifte Klammern eingeschlossen ist

  Im folgenden Beispielcode wird die `traceParameter`-Funktion mit einem Funktionsausdruck deklariert:

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

  Im Gegensatz zu einer Funktionsanweisung geben Sie keinen Funktionsnamen an. Ein anderer wichtiger Unterschied zwischen Funktionsausdrücken und Funktionsanweisungen besteht darin, dass ein Funktionsausdruck eher einen Ausdruck als eine Anweisung darstellt. Dies bedeutet, dass ein Funktionsausdruck im Gegensatz zu einer Funktionsanweisung nicht allein stehen kann. Ein Funktionsausdruck kann nur als Teil einer Anweisung verwendet werden, in der Regel wird hier eine Zuweisungsanweisung verwendet. Das folgende Beispiel zeigt einen Funktionsausdruck, der einem Array-Element zugewiesen ist:

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

Auswählen zwischen Anweisungen und Ausdrücken

Im Allgemeinen verwenden Sie eine Funktionsanweisung, es sei denn, besondere Umstände verlangen nach einem Ausdruck. Funktionsanweisungen sind weniger ausführlich und bieten eine einheitlichere Führung zwischen dem strikten Modus und dem Standardmodus als Funktionsausdrücke.

Funktionsanweisungen sind einfacher zu lesen als Zuweisungsanweisung, die Funktionsausdrücke enthalten. Funktionsanweisungen halten den Code kurz und sind einfacher zu verstehen als Funktionsausdrücke, für die Sie die beiden Schlüsselwörter `var` und `function` verwenden müssen.

Funktionsanweisungen sorgen für eine einheitlichere Führung zwischen den beiden Compiler-Modi, da Sie die Punktsyntax sowohl im strikten als auch im Standardmodus verwenden können, um eine Methode aufzurufen, die mit einer Funktionsanweisung deklariert wurde. Dies gilt nicht unbedingt für Methoden, die mit einem Funktionsausdruck deklariert wurden. Der folgende Code generiert beispielsweise eine Klasse namens „Example“ mit zwei Methoden: `methodExpression()`, die mit einem Funktionsausdruck deklariert ist, und `methodStatement()`, die mit einer Funktionsanweisung deklariert ist. Im strikten Modus können Sie die Punktsyntax nicht verwenden, um die `methodExpression()`-Methode aufzurufen.

```
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
```

Funktionsausdrücke sind wesentlich besser zum Programmieren geeignet, wenn das Hauptaugenmerk auf dem Echtzeit- bzw. dynamischen Verhalten liegt. Wenn Sie den strikten Modus bevorzugen, aber ebenfalls eine Methode aufrufen müssen, die mit einem Funktionsausdruck deklariert wurde, können Sie beide Verfahren verwenden. Zunächst können Sie die Methode mit eckigen Klammern (`[]`) anstelle des Punktoperators (`.`) aufrufen. Der folgende Methodenaufruf wird sowohl im strikten als auch im Standardmodus erfolgreich ausgeführt:

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

Alternativ können Sie die gesamte Klasse als dynamische Klasse deklarieren. Obwohl Ihnen dieses Verfahren ermöglicht, die Methode mit dem Punktoperator aufzurufen, gibt es einen Nachteil: Sie opfern einen Teil der Funktionsmerkmale im strikten Modus für alle Instanzen dieser Klasse. Beispielsweise erzeugt der Compiler keinen Fehler, wenn Sie versuchen, bei einer Instanz einer dynamischen Klasse auf eine undefinierte Eigenschaft zuzugreifen.

Es gibt einige Situationen, in denen Funktionsausdrücke sinnvoll sind. Häufig werden Funktionsausdrücke für Funktionen genutzt, die nur einmal verwendet und dann verworfen werden. Eine weitere weniger übliche Anwendungsmöglichkeit ist das Anfügen einer Funktion an eine Prototypeigenschaft. Weitere Informationen finden Sie unter Prototypobjekt.

Es gibt zwei feine Unterschiede zwischen Funktionsanweisungen und Funktionsausdrücken, die Sie unbedingt bei der Auswahl der zu verwendenden Technik berücksichtigen sollten. Der erste Unterschied ist, dass Funktionsausdrücke hinsichtlich der Speicherverwaltung und -bereinigung (Garbage Collection) nicht unabhängig als Objekte existieren. Anders ausgedrückt, wenn Sie einen Funktionsausdruck einem anderen Objekt zuweisen (beispielsweise einem Array-Element oder einer Objekteigenschaft), erstellen Sie in Ihrem Code lediglich einen Verweis auf diesen Funktionsausdruck. Wenn das Array oder Objekt, an das Ihr Funktionsausdruck angehängt ist, außerhalb des Gültigkeitsbereichs gerät oder aus anderen Gründen nicht mehr zur Verfügung steht, ist kein Zugriff auf den Funktionsausdruck mehr möglich. Wenn das Array oder Objekt gelöscht wurde, wird der vom Funktionsausdruck belegte Speicher für die Garbage Collection verfügbar. Dies bedeutet, dass der Speicherbereich für andere Zwecke neu vergeben werden kann.

Das folgende Beispiel zeigt einen Funktionsausdruck, in dem die Funktion nicht mehr zur Verfügung steht, nachdem die Eigenschaft, welcher der Ausdruck zugewiesen war, gelöscht wurde. Die Test-Klasse ist dynamisch. Dies bedeutet, Sie können eine Eigenschaft namens `functionExp` hinzufügen, die einen Funktionsausdruck aufnimmt. Die `functionExp()`-Funktion kann mit dem Punktoperator aufgerufen werden, aber nachdem die Eigenschaft `functionExp` gelöscht wurde, kann nicht mehr auf die Funktion zugegriffen werden.

```
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
```

Wenn die Funktion andererseits zuerst mit einer Funktionsanweisung definiert wird, existiert sie als eigenständiges Objekt und kann auch weiterhin verwendet werden, nachdem die Eigenschaft gelöscht wurde, an die sie angefügt wurde. Der Operator `delete` kann nur mit Eigenschaften von Objekten verwendet werden. Daher hat auch ein Aufruf zum Löschen der Funktion `stateFunc()` keine Auswirkungen.

```
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
```

Der zweite Unterschied zwischen Funktionsanweisungen und Funktionsausdrücken besteht darin, dass Funktionsanweisungen im gesamten Gültigkeitsbereich existieren, in dem sie definiert wurden. Hierzu zählen auch die Anweisungen, die vor der Funktionsanweisung stehen. Im Gegensatz dazu sind Funktionsausdrücke nur für nachfolgende Anweisungen definiert. Im folgenden Codebeispiel wird die `scopeTest()`-Funktion erfolgreich aufgerufen, bevor sie definiert wird:

```
statementTest(); // statementTest

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

Funktionsausdrücke sind erst verfügbar, nachdem sie definiert wurden. Aus diesem Grund führt der folgende Code zu einem Laufzeitfehler:

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

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

Zurückgeben von Werten aus Funktionen

Wenn Ihre Funktion einen Wert zurückgeben soll, verwenden Sie die `return`-Anweisung gefolgt von dem Ausdruck oder Literalwert, der zurückgegeben werden soll. Der folgende Code gibt beispielsweise einen Ausdruck zurück, der einen Parameter darstellt:

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

Beachten Sie, dass die `return`-Anweisung die Funktion beendet. Das heißt, alle Anweisungen nach einer `return`-Anweisung werden nicht ausgeführt, wie im folgenden Code dargestellt:

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

Im strikten Modus müssen Sie einen Wert des entsprechenden Typs zurückgeben, wenn Sie einen Rückgabetyp festgelegt haben. Mit dem folgenden Code wird im strikten Modus eine Fehlermeldung erzeugt, da er keinen gültigen Wert zurückgibt:

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

Verschachtelte Funktionen

Sie können Funktionen verschachteln; anders ausgedrückt, Sie können Funktionen innerhalb von anderen Funktionen deklarieren. Eine verschachtelte Funktion steht nur innerhalb ihrer übergeordneten Funktion zur Verfügung, es sei denn, dem externen Code wurde ein Verweis auf die Funktion übergeben. Im folgenden Codebeispiel werden zwei verschachtelte Funktionen in der Funktion `getNameAndVersion()` deklariert:

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

Verschachtelte Funktionen werden als Funktionshüllen an einen externen Code übergeben. Dies bedeutet, dass die Funktion alle Definitionen beibehält, die sich beim Definieren der Funktion in ihrem Gültigkeitsbereich befanden. Weitere Informationen finden Sie unter Gültigkeitsbereich von Funktionen.

Funktionsparameter

ActionScript 3.0 umfasst einige Funktionsmerkmale für Funktionsparameter, die weniger erfahrenen ActionScript-Programmierern wahrscheinlich unbekannt sind. Die Idee der Übergabe von Parametern als Wert oder als Verweis dürfte den meisten Programmierern vertraut sein, während das `arguments`-Objekt und der ...-Parameter (Rest) wohl vielen unbekannt sein dürften.

Übergeben von Argumenten als Wert oder als Verweis

Bei vielen Programmiersprachen ist es wichtig, den Unterschied zwischen der Übergabe von Argumenten als Wert (Englisch „pass by value“) oder als Verweis (Englisch „pass by reference“) zu verstehen. Der Unterschied kann sich auf das Codedesign auswirken.

Eine Übergabe als Wert bedeutet, dass der Wert des Arguments in ein lokale Variable kopiert wird, damit er innerhalb der Funktion verwendet werden kann. Eine Übergabe als Verweis bedeutet, dass nur ein Verweis auf das Argument anstelle des tatsächlichen Wertes übergeben wird. Es wird keine Kopie des tatsächlichen Arguments erstellt. Stattdessen wird ein Verweis auf die als Argument übergebene Variable erstellt und einer lokalen Variablen für die Verwendung innerhalb der Funktion zugewiesen. Als Verweis auf eine Variable außerhalb der Funktion ermöglicht die lokale Variable das Ändern des Wertes der Ursprungsvariablen.

In ActionScript 3.0 werden alle Argumente als Verweis übergeben, da alle Werte als Objekte gespeichert sind. Objekte, die zu den Grunddatentypen gehören (Boolean, Number, int, uint und String), haben besondere Operatoren, durch die sich Objekte so verhalten können, als würden sie als Wert übergeben. Im folgenden Beispielcode wird eine Funktion namens `passPrimitives()` erstellt, die zwei Parameter namens `xParam` und `yParam` definiert; beides Parameter des Datentyps „int“. Diese Parameter ähneln lokalen Variablen, die innerhalb des Rumpfs der `passPrimitives()`-Funktion deklariert wurden. Wenn die Funktion mit den Argumenten `xValue` und `yValue`

aufgerufen wird, werden die Parameter `xParam` und `yParam` mit Verweisen auf die int-Objekte initiiert, die von `xValue` und `yValue` dargestellt werden. Da diese Argumente Grundtypen sind, verhalten sie sich so, als wenn sie als Wert übergeben worden wären. Obwohl `xParam` und `yParam` zunächst nur Verweise auf die Objekte `xValue` und `yValue` enthalten, erzeugen alle Änderungen an den Variablen innerhalb des Funktionsrumpfs neue Kopien der Werte im Arbeitsspeicher.

```
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
```

In der `passPrimitives()`-Funktion werden die Werte von `xParam` und `yParam` erhöht. Dies wirkt sich jedoch nicht auf die Werte von `xValue` und `yValue` aus, wie aus der letzten `trace`-Anweisung ersichtlich ist. Dies würde eintreten, wenn die Parameter mit den gleichen Namen wie die Variablen `xValue` und `yValue` benannt wären, da `xValue` und `yValue` innerhalb der Funktion auf neue Speicherstellen im Arbeitsspeicher verweisen würden, die separat von den Variablen mit dem gleichen Namen außerhalb der Funktion existieren würden.

Alle anderen Objekte, also Objekte, die nicht zu den Grundtypen gehören, werden immer als Verweis übergeben, sodass Sie die Werte der Ursprungsvariablen ändern können. Im folgenden Beispielcode wird ein Objekt namens `objVar` mit zwei Eigenschaften erstellt, `x` und `y`. Das Objekt wird als Argument an die Funktion `passByRef()` übergeben. Da das Objekt nicht zu den Grundtypen gehört, wird es nicht nur als Verweis übergeben, sondern bleibt auch ein Verweis. Dies bedeutet, dass sich Änderungen an den Parametern innerhalb der Funktion auf die Objekteigenschaften außerhalb der Funktion auswirken.

```
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
```

Der `objParam`-Parameter verweist auf das gleiche Objekt wie die globale Variable `objVar`. Wie Sie anhand der `trace`-Anweisungen im Codebeispiel sehen können, werden Änderungen an den Eigenschaften `x` und `y` des `objParam`-Objekts im `objVar`-Objekt widergespiegelt.

Standard-Parameterwerte

In ActionScript 3.0 können Sie *Standardparameterwerte* für eine Funktion deklarieren. Wenn ein Aufruf einer Funktion mit Standard-Parameterwerten einen Parameter mit Standardwerten weglässt, wird der in der Funktionsdefinition angegebene Wert für diesen Parameter verwendet. Alle Parameter mit Standardwerten müssen am Ende der Parameterliste platziert werden. Bei den als Standardwerte zugewiesenen Werten muss es sich um Konstanten zur Kompilierzeit handeln. Das Vorhandensein eines Standardwerts für einen Parameter macht diesen Parameter zu einem *optionalen Parameter*. Ein Parameter ohne Standardwert wird als *erforderlicher Parameter* betrachtet.

Im folgenden Codebeispiel wird eine Funktion mit drei Parametern erstellt, von denen zwei Standardwerte aufweisen. Wenn die Funktion mit nur einem Parameter aufgerufen wird, werden die Standardwerte für die Parameter verwendet.

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

arguments-Objekt

Wenn Parameter an eine Funktion übergeben werden, können Sie mit dem `arguments`-Objekt auf Informationen über die an Ihre Funktion übergebenen Parameter zugreifen. Einige wichtige Aspekte des `arguments`-Objekts sind:

- Das `arguments`-Objekt ist ein Array, dass alle an die Funktion übergebenen Parameter enthält.
- Die `arguments.length`-Eigenschaft meldet die Anzahl der Parameter, die an die Funktion übergeben werden.
- Die `arguments.callee`-Eigenschaft bietet einen Verweis auf die Funktion selbst. Dies eignet sich für rekursive Aufrufe der Funktionsausdrücke.

  Hinweis: *Das `arguments`-Objekt ist nicht verfügbar, wenn Parameter mit `arguments` bezeichnet wurden oder wenn Sie den Parameter „...(rest)“ verwenden.*

  Wenn im Hauptteil einer Funktion auf das `arguments`-Objekt verwiesen wird, können in ActionScript 3.0 Funktionsaufrufe mehr Parameter enthalten als in der Funktionsdefinition definiert wurden. Dies führt jedoch im strikten Modus zu einem Compiler-Fehler, wenn die Anzahl der Parameter nicht mit der Anzahl der erforderlichen Parameter (und aller ggf. festgelegten optionalen Parameter) übereinstimmt. Sie können den Array-Aspekt des `arguments`-Objekts verwenden, um auf Parameter zuzugreifen, die an die Funktion übergeben wurden. Dabei spielt es keine Rolle, ob dieser Parameter in der Funktionsdefinition definiert wurde. Im folgenden Beispielcode, der nur im Standardmodus kompiliert wird, wird das Array `arguments` zusammen mit der Eigenschaft `arguments.length` verwendet, um alle Parameter zu verfolgen, die an die `traceArgArray()`-Funktion übergeben wurden:

  ```
  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
  ```

  Die `arguments.callee`-Eigenschaft wird häufig in anonymen Funktionen zum Erstellen einer Rekursion verwendet. Außerdem können Sie mit dieser Eigenschaft die Flexibilität Ihres Codes erweitern. Wenn sich der Name einer rekursiven Funktion im Verlauf Ihres Entwicklungszyklus ändert, müssen Sie sich nicht um die Änderung des rekursiven Aufrufs in Ihrem Funktionsrumpf kümmern, wenn Sie `arguments.callee` anstelle des Funktionsnamens verwenden. Die `arguments.callee`-Eigenschaft wird im folgenden Funktionsausdruck verwendet, um eine Rekursion zu ermöglichen:

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

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

Wenn Sie den Parameter „... (rest)“ in Ihrer Funktionsdeklaration verwenden, steht Ihnen das `arguments`-Objekt nicht zur Verfügung. Stattdessen greifen Sie mit den Parameternamen, die Sie für die Parameter deklariert haben, auf die Parameter zu.

Verwenden Sie den String „`arguments`" nicht als Parameternamen, da dadurch das `arguments`-Objekt verborgen wird. Angenommen, die Funktion `traceArgArray()` wird umgeschrieben, um einen `arguments`-Parameter hinzuzufügen, so zeigen die Verweise auf `arguments` im Funktionsrumpf auf den Parameter anstatt auf das `arguments`-Objekt. Der folgende Code erzeugt keine Ausgabe:

```
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
```

In früheren Versionen von ActionScript enthielt das `arguments`-Objekt noch eine Eigenschaft namens `caller`, die einen Verweis auf die Funktion darstellt, welche die aktuelle Funktion aufruft. Die `caller`-Eigenschaft ist in ActionScript 3.0 nicht mehr vorhanden. Wenn Sie einen Verweis auf die aufrufende Funktion benötigen, können Sie die aufrufende Funktion so ändern, dass sie einen weiteren Parameter übergibt, der einen Verweis auf sich selbst darstellt.

... (rest)-Parameter

Mit ActionScript 3.0 wurde eine neue Parameterdeklaration mit der Bezeichnung ... (rest)-Parameter eingeführt. Mit diesem Parameter können Sie einen Array-Parameter angeben, der mit einer beliebigen Anzahl von kommagetrennten Argumenten arbeitet. Der Parameter kann jeden Namen annehmen, der kein reserviertes Wort ist. Diese Parameterdeklaration muss der letzte angegebene Parameter sein. Wenn Sie diesen Parameter verwenden, steht das Objekt `arguments` nicht mehr zur Verfügung. Obwohl Ihnen der Parameter „... (rest)“ die gleichen Funktionsmerkmale wie das `arguments`-Array und die `arguments.length`-Eigenschaft zur Verfügung stellt, verfügt es nicht über die Funktionsmerkmale, der Ihnen von `arguments.callee` bereitgestellt werden. Bevor Sie den ... (rest)-Parameter einsetzen, müssen Sie sicherstellen, dass Sie `arguments.callee` nicht benötigen.

Im folgenden Beispielcode wird die `traceArgArray()`-Funktion mit dem ... (rest)-Parameter anstelle des `arguments`-Objekts neu geschrieben:

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

traceArgArray(1, 2, 3);

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

Der ... (rest)-Parameter kann zusammen mit anderen Parametern verwendet werden, er muss jedoch der letzte angegebene Parameter sein. Im folgenden Beispielcode wird die `traceArgArray()`-Funktion so geändert, dass der erste Parameter `x` den Datentyp „int“ aufweist und der zweite Parameter den ... (rest)-Parameter verwendet: Die Ausgabe überspringt den ersten Wert, weil der Parameter jetzt kein Teil des Arrays mehr ist, das vom ... (rest)-Parameter erstellt wurde.

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

traceArgArray(1, 2, 3);

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

Funktionen als Objekte

Funktionen in ActionScript 3.0 sind Objekte. Beim Erstellen einer Funktion erstellen Sie ein Objekt, das nicht nur als Parameter an eine andere Funktion übergeben werden kann, sondern dem auch Eigenschaften und Methoden angefügt werden können.

Funktionen, die als Argumente an eine weitere Funktion übergeben werden, werden als Verweis und nicht als Wert übergeben. Wenn Sie eine Funktion als ein Argument übergeben, verwenden Sie nur den Bezeichner und nicht den Klammernoperator, der zum Aufrufen der Methode verwendet wird. Im folgenden Beispielcode wird eine Funktion namens `clickListener()` als ein Argument an die `addEventListener()`-Methode übergeben:

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

Obwohl es in ActionScript unerfahrenen Programmierern seltsam erscheinen mag, können Funktionen ebenso wie jedes andere Objekte Eigenschaften und Methoden aufweisen. Tatsächlich verfügt jede Funktion über eine schreibgeschützte Eigenschaft namens `length`, in der die Anzahl der Parameter gespeichert wird, die für die Funktion definiert wurden. Dies unterscheidet sich von der `arguments.length`-Eigenschaft, welche die Anzahl der Argumente meldet, die an die Funktion gesendet werden. Sie erinnern sich, dass in ActionScript die Anzahl der Argumente, die an eine Funktion gesendet werden, die Anzahl der Parameter übersteigen kann, die für diese Funktion definiert wurden. Das folgende Beispiel, das nur im Standardmodus kompiliert wird, da im strikten Modus die Anzahl der übergebenen Argumente und der definierten Parameter exakt übereinstimmen muss, zeigt die Unterschiede zwischen den beiden Eigenschaften:

```
// 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 */
```

Im Standardmodus können Sie eigene Funktionseigenschaften definieren, indem Sie die Definition außerhalb des Funktionsrumpfes durchführen. Funktionseigenschaften dienen als quasi-statische Eigenschaften, mit denen Sie den Status einer mit der Funktion verwandten Variablen speichern können. Angenommen Sie möchten verfolgen, wie oft eine bestimmte Funktion aufgerufen wird. Eine solche Funktion ist sinnvoll, wenn Sie ein Spiel schreiben und verfolgen möchten, wie oft ein Benutzer einen bestimmten Befehl verwendet hat. Natürlich können Sie auch die Eigenschaft einer statischen Klasse für diesen Zweck verwenden. Im folgenden Beispielcode wird eine Funktionseigenschaft außerhalb der Funktionsdeklaration erstellt und die Eigenschaft bei jedem Funktionsaufruf inkrementiert. Dieser Beispielcode wird nur im Standardmodus kompiliert, da im strikten Modus keine dynamischen Eigenschaften zu Funktionen hinzugefügt werden können.

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

someFunction.counter = 0;

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

Gültigkeitsbereich von Funktionen

Der Gültigkeitsbereich einer Funktion legt nicht nur fest, an welcher Stelle im Programm diese Funktion aufgerufen werden kann, sondern auch, welche Definitionen auf die Funktion zugreifen können. Für Funktionsbezeichner gelten die gleichen Gültigkeitsbereichsregeln wie für Variablenbezeichner. Eine Funktion, die im globalen Gültigkeitsbereich deklariert ist, steht im gesamten Code zur Verfügung. Beispielsweise enthält ActionScript 3.0 globale Funktionen wie `isNaN()` und `parseInt()`, die sich an beliebigen Stellen im Code befinden können. Eine verschachtelte Funktion, d. h. eine Funktion, die innerhalb einer anderen Funktion deklariert ist, kann an beliebiger Stelle innerhalb der Funktion verwendet werden, in der sie deklariert wurde.

Gültigkeitsbereichskette

Jedes Mal, wenn eine Funktion die Ausführung beginnt, wird eine Reihe von Objekten und Eigenschaften erstellt. Zunächst wird ein besonderes, als *Aktivierungsobjekt* bezeichnetes Objekt erstellt, das alle im Funktionsrumpf erstellten Parameter und lokalen Variablen oder Funktionen speichert. Ein direkter Zugriff auf das Aktivierungsobjekt ist nicht möglich, da es sich um einen internen Mechanismus handelt. Zweitens wird eine *Gültigkeitsbereichskette* erstellt, die eine sortierte Liste der Objekte enthält, die zur Laufzeit nach Bezeichnerdeklarationen durchsucht werden. Jede ausgeführte Funktion verfügt über eine Gültigkeitsbereichskette, die in einer internen Eigenschaft gespeichert ist. Bei einer verschachtelten Funktion beginnt die Gültigkeitsbereichskette mit ihrem eigenen Aktivierungsobjekt, gefolgt vom Aktivierungsobjekt der übergeordneten Funktion. Die Kette wird in dieser Weise fortgesetzt, bis sie das globale Objekt erreicht. Das globale Objekt wird erstellt, wenn ein ActionScript-Programm beginnt, und enthält alle globalen Variablen und Funktionen.

Funktionshüllen

Eine *Funktionshülle* (Function Closure) ist ein Objekt, das eine Momentaufnahme einer Funktion und ihrer *lexikalischen Umgebung* enthält. Die lexikalische Umgebung einer Funktion enthält alle Variablen, Eigenschaften, Methoden und Objekte in der Gültigkeitsbereichskette einer Funktion sowie deren Werte. Funktionshüllen werden immer dann erstellt, wenn eine Funktion unabhängig von einem Objekt oder einer Klasse ausgeführt wird. Die Tatsache, dass Funktionshüllen den Gültigkeitsbereich beibehalten, in dem sie erstellt wurden, führt zu interessanten Ergebnissen, wenn eine Funktion als Argument oder Rückgabewert an einen anderen Gültigkeitsbereich übergeben wird.

Im folgenden Beispielcode werden zwei Funktionen erstellt: `foo()`, die eine verschachtelte Funktion namens `rectArea()` zurückgibt, mit der die Fläche eines Rechtecks berechnet wird, und `bar()`, die `foo()` aufruft und die zurückgegebene Funktionshülle in einer Variablen namens `myProduct` speichert. Obwohl die Funktion `bar()` ihre eigene lokale Variable `x` (mit einem Wert von 2) definiert, behält Sie beim Aufruf der Funktionshülle `myProduct()` die Variable `x` (mit einem Wert von 40) bei, der in der Funktion `foo() definiert ist.` Die `bar()`-Funktion gibt daher den Wert `160` anstelle von `8 zurück`.

```
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
```

Methoden verhalten sich insofern ähnlich, als dass sie ebenfalls Informationen über die lexikalische Umgebung beibehalten, in der sie erstellt wurden. Diese Eigenschaft macht sich insbesondere dann bemerkbar, wenn eine Methode aus ihrer Instanz extrahiert wird, wodurch eine gebundene Methode entsteht. Der wesentliche Unterschied zwischen einer Funktionshülle und einer gebundenen Methode besteht darin, dass sich der Wert des Schlüsselwortes `this` in einer gebundenen Methode immer auf die Instanz bezieht, an die sie ursprünglich angefügt wurde, während sich der Wert des Schlüsselwortes `this` in einer Funktionshülle ändern kann.

Kapitel 4: Objektorientierte Programmierung mit ActionScript

Einführung in die objektorientierte Programmierung

Bei der objektorientierten Programmierung (OOP) wird der Code eines Programms zur Organisation in Objekten gruppiert. Der Begriff *Objekt* bezieht sich dabei auf ein einzelnes Element, das Informationen (Datenwerte) und Funktionalität enthält. Bei der objektorientierten Organisation eines Programms gruppieren Sie bestimmte Informationseinheiten mit häufigen Funktionen oder Aktionen, die zu diesen Informationen gehören. Beispielsweise können Sie Informationen zu Musik, wie Albumtitel, Songtitel oder Interpret, mit einer bestimmten Funktionalität gruppieren, wie zum Beispiel „Titel zur Wiedergabeliste hinzufügen“ oder „Alle Lieder dieses Interpreten abspielen“. Diese Einheiten werden in einem einzelnen Element, dem Objekt zusammengefasst (z. B. „Album“ oder „MusicTrack“). Die Gruppierung von Werten und Funktionen bietet mehrere Vorteile. Ein wesentlicher Vorteil ist, dass Sie nur eine Variable anstatt von mehreren Variablen verwenden müssen. Außerdem wird zugehörige Funktionalität zusammengehalten. Schließlich bietet die Kombination von Informationen und Funktionalität die Möglichkeit, Programme auf realitätsnahe Weise zu strukturieren.

Klassen

Eine Klasse ist eine abstrakte Darstellung eines Objekts. Eine Klasse speichert Informationen über die Datentypen, die ein Objekt aufnehmen kann, und die Verhalten, die ein Objekt an den Tag legen kann. Der Nutzen einer solchen Abstraktion wird nicht sofort offensichtlich, wenn Sie kleine Skripts schreiben, die nur wenige miteinander interagierende Objekte enthalten. Doch je größer ein Programm, desto mehr Objekte müssen verwaltet werden. In diesem Fall können Sie mithilfe von Klassen die Erstellungsweise und Interaktion von Objekten besser steuern.

Bereits in ActionScript 1.0 konnten ActionScript-Programmierer function-Objekte verwenden, um Konstrukte zu erstellen, die den heutigen Klassen ähnelten. Mit ActionScript 2.0 ist eine formale Unterstützung für Klassen mit Schlüsselwörtern wie `class` und `extends` hinzugekommen. ActionScript 3.0 unterstützt die in ActionScript 2.0 eingeführten Schlüsselwörter, bietet jedoch zudem neue Möglichkeiten. So ermöglicht ActionScript 3.0 beispielsweise eine verbesserte Zugriffskontrolle über die Attribute `protected` und `internal`. Weiterhin verbessern die Schlüsselwörter `final` und `override` die Vererbungssteuerung.

Entwickler, die in Programmiersprachen wie Java, C++ oder C# Klassen erstellt haben, werden sich in ActionScript leicht zurechtfinden. ActionScript verwendet zahlreiche derselben Schlüsselwörter und Attributnamen, wie `class`, `extends` und `public`.

Hinweis: *In der Dokumentation zu Adobe ActionScript bezieht sich der Begriff „Eigenschaft“ auf ein beliebiges Mitglied eines Objekts oder einer Klasse, einschließlich Variablen, Konstanten und Methoden. Darüber hinaus haben hier die Begriffe „Klasse“ und „statisch“ unterschiedliche Bedeutungen, obwohl sie häufig als Synonyme verwendet werden. Beispielsweise bezieht sich der Begriff „Klasseneigenschaften“ auf alle Mitglieder einer Klasse und nicht nur auf statische Mitglieder.*

Klassendefinitionen

Für Klassendefinitionen in ActionScript 3.0 wird eine ähnliche Syntax wie für Klassendefinitionen in ActionScript 2.0 verwendet. Die richtige Syntax einer Klassendefinition umfasst zunächst das Schlüsselwort `class`, gefolgt vom Klassennamen. Dem Klassennamen folgt der in geschweifte Klammern ({}) eingeschlossene Klassenrumpf. Im folgenden Beispielcode wird eine neue Klasse namens „Shape“ erstellt, die eine Variable namens `visible` enthält:

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

Eine wichtige Syntaxänderung betrifft Klassendefinitionen, die sich in einem Paket befinden. In ActionScript 2.0 galt: Wenn sich eine Klasse in einem Paket befindet, muss der Paketname in der Klassendeklaration enthalten sein. n ActionScript 3.0, mit dem die `package`-Anweisung eingeführt wird, muss sich der Paketname nicht mehr in der Klassen-, sondern in der Paketdeklaration befinden. Die folgenden Klassendeklarationen zeigen, wie die BitmapData-Klasse, die Teil des flash.display-Pakets ist, in ActionScript 2.0 und ActionScript 3.0 definiert wird:

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

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

Klassenattribute

Mit ActionScript 3.0 können Sie Klassendefinitionen mit einem der folgenden vier Attribute bearbeiten:

Attribut	Definition
`dynamic`	Ermöglicht zur Laufzeit das Hinzufügen von Eigenschaften zu Instanzen.
`final`	Kann nicht von einer anderen Klasse erweitert werden.
`internal` (Standard)	Sichtbar für Verweise innerhalb des aktuellen Pakets.
`public`	Sichtbar für alle Verweise.

Bei jedem dieser Attribute (außer bei `internal`) wird das Attribut explizit eingeschlossen, um das zugewiesene Verhalten zu erhalten. Wenn Sie beispielsweise beim Definieren einer Klasse das `dynamic`-Attribut weglassen, können Sie einer Klasseninstanz zur Laufzeit keine Eigenschaften hinzufügen. Ein Attribut wird explizit zugewiesen, indem Sie es am Anfang der Klassendefinition platzieren. Dies wird im folgenden Code gezeigt:

```
dynamic class Shape {}
```

Beachten Sie, dass die Liste kein Attribut namens `abstract` enthält, Abstrakte Klassen werden in ActionScript 3.0 nicht unterstützt. Beachten Sie außerdem, dass die Liste keine Attribute namens `private` und `protected` enthält. Diese Attribute sind nur innerhalb einer Klassendefinition von Bedeutung und können nicht auf Klassen selbst angewendet werden. Soll eine Klasse auch außerhalb eines Pakets öffentlich sichtbar sein, platzieren Sie die Klasse in einem Paket und weisen ihr dann das Attribut `internal` zu. Alternativ können Sie die beiden Attribute `internal` und `public` weglassen. In diesem Fall fügt der Compiler das Attribut `internal` automatisch hinzu. Sie können eine Klasse auch so definieren, dass sie nur in der Quelldatei sichtbar ist, in der sie definiert wurde. Platzieren Sie die Klasse unten in der Quelldatei, unter der schließenden geschweiften Klammer der Paketdefinition.

Klassenrumpf

Der Rumpf der Klasse ist in geschweifte Klammern eingeschlossen. Er definiert die Variablen, Konstanten und Methoden der Klasse. Das folgende Beispiel zeigt die Deklaration für die Accessibility-Klasse in ActionScript 3.0:

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

Sie können auch einen Namespace in einem Klassenrumpf definieren. Im folgenden Beispielcode wird gezeigt, wie ein Namespace in einem Klassenrumpf definiert und als Attribut für eine Methode in dieser Klasse verwendet werden kann:

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

Mit ActionScript3.0 können Sie nicht nur Definitionen, sondern auch Anweisungen in einen Klassenrumpf aufnehmen. Anweisungen, die sich innerhalb des Klassenrumpfes, aber außerhalb einer Methodendefinition befinden, werden genau einmal ausgeführt. Diese Ausführung tritt auf, wenn die Klassendefinition zum ersten Mal erkannt wird und das zugehörige Klassenobjekt erstellt wird. Der folgende Beispielcode enthält einen Aufruf der externen Funktion `hello()` und eine `trace`-Anweisung, die eine Bestätigungsmeldung ausgibt, wenn die Klasse definiert ist:

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

In ActionScript 3.0 ist es zulässig, eine statische Eigenschaft und eine Instanzeigenschaft mit dem gleichen Namen im gleichen Klassenrumpf zu definieren. Im folgenden Beispielcode werden eine statische Variable namens `message` und eine Instanzvariable mit dem gleichen Namen deklariert:

```
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
```

Klasseneigenschaftsattribute

In Abhandlungen über das ActionScript-Objektmodell wird der Begriff *Eigenschaft* für alles verwendet, was ein Mitglied einer Klasse sein kann; dazu gehören auch Variable, Konstanten und Methoden. Im Referenzhandbuch zu Adobe ActionScript 3.0 für die Adobe Flash-Plattform wird der Begriff jedoch im engeren Sinne verwendet. In diesem Zusammenhang bezieht sich der Begriff „Eigenschaft“ nur auf Klassenmitglieder, die Variablen sind oder die von einer Get- oder Set-Methode definiert werden. In ActionScript 3.0 gibt es eine Reihe von Attributen, die mit jeder Eigenschaft einer Klasse verwendet werden können. Diese Attribute sind in der folgenden Tabelle aufgeführt.

Attribut	Definition
`internal` (Standard)	Sichtbar für Verweise innerhalb des gleichen Pakets.
`private`	Sichtbar für Verweise in der gleichen Klasse.
`protected`	Sichtbar für Verweise in der gleichen Klasse sowie in abgeleiteten Klassen.
`public`	Sichtbar für alle Verweise.
`static`	Gibt an, dass eine Eigenschaft zur Klasse und nicht zu Instanzen dieser Klasse gehört.
`UserDefinedNamespace`	Benutzerdefinierter Namespace-Name.

Namespace-Attribute zur Zugriffskontrolle

In ActionScript 3.0 stehen vier spezielle Attribute zur Verfügung, mit denen der Zugriff auf Eigenschaften kontrolliert werden kann, die in einer Klasse definiert sind: `public`, `private`, `protected` und `internal`.

Das Attribut `public` macht eine Eigenschaft im gesamten Skript sichtbar. Um beispielsweise eine Methode für Code außerhalb des Pakets verfügbar zu machen, müssen Sie sie mit dem Attribut `public` deklarieren. Dies gilt für jede Eigenschaft, unabhängig davon, ob sie mit dem Schlüsselwort `var`, `const` oder `function` deklariert wurde.

Das Attribut `private` macht eine Eigenschaft nur für aufrufende Objekte innerhalb der Klasse sichtbar, in der die Eigenschaft definiert wurde. Dieses Verhalten unterscheidet sich von dem des `private`-Attributs in ActionScript 2.0, das einer Unterklasse den Zugriff auf eine private Eigenschaft einer übergeordneten Klasse gestattete. Eine andere wesentliche Verhaltensänderung betrifft den Laufzeitzugriff. In ActionScript 2.0 verhinderte das Schlüsselwort `private` den Zugriff während der Kompilierung und wurde zur Laufzeit einfach umgangen. Dies ist in ActionScript 3.0 nicht mehr möglich. Auf Eigenschaften, die als `private` gekennzeichnet sind, kann weder während der Kompilierung noch zur Laufzeit zugegriffen werden.

Der folgende Code erstellt eine einfache Klasse namens „PrivateExample“ mit einer als „private“ deklarierten Variablen und versucht dann, von außerhalb der Klasse aus auf diese Variable zuzugreifen.

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

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

In ActionScript 3.0 führt der Versuch, mit dem Punktoperator auf eine als „private“ deklarierte Eigenschaft zuzugreifen (`myExample.privVar`), im strikten Modus zu einem Kompilierzeitfehler. Andernfalls tritt der Fehler zur Laufzeit auf, als ob Sie den Eigenschaften-Zugriffsoperator (`myExample["privVar"]`) verwenden.

In der folgenden Tabelle sind die Ergebnisse des versuchten Zugriffs auf eine als „private“ deklarierte Eigenschaft aufgeführt, die zu einer versiegelten (nicht dynamischen) Klasse gehört:

	Strikter Modus	**Standardmodus**
Punktoperator (`.`)	Kompilierzeitfehler	Laufzeitfehler
Klammernoperator (`[]`)	Laufzeitfehler	Laufzeitfehler

Bei Klassen, die mit dem `dynamic`-Attribut deklariert sind, führen Versuche, auf eine als „private“ deklarierte Variable zuzugreifen, nicht zu einem Laufzeitfehler. Stattdessen ist die Variable nicht sichtbar, sodass der Wert `undefined` zurückgegeben wird. Wenn Sie den Punktoperator jedoch im strikten Modus verwenden, tritt ein Kompilierzeitfehler auf. Der folgende Beispielcode entspricht dem vorangegangenen, außer dass die PrivateExample-Klasse als dynamische Klasse deklariert ist:

```
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
```

Anstatt einen Fehler zu erzeugen, geben dynamische Klassen im Allgemeinen den Wert `undefined` zurück, wenn Code außerhalb einer Klasse versucht, auf eine als „private“ deklarierte Eigenschaft zuzugreifen. Die folgende Tabelle zeigt, dass nur dann ein Fehler erzeugt wird, wenn im strikten Modus mit dem Punktoperator auf eine als „private“ deklarierte Eigenschaft zugegriffen wird:

	Strikter Modus	**Standardmodus**
Punktoperator (`.`)	Kompilierzeitfehler	`undefined`
Klammernoperator (`[]`)	`undefined`	`undefined`

Das in ActionScript 3.0 neu eingeführte Attribut `protected` macht eine Eigenschaft für aufrufende Objekte innerhalb der eigenen Klasse oder einer Unterklasse sichtbar. Anders ausgedrückt, eine als „protected“ deklarierte Eigenschaft steht innerhalb der eigenen Klasse oder in Klassen zur Verfügung, die sich in der Vererbungshierarchie unterhalb der eigenen Klasse befinden. Dies gilt unabhängig davon, ob sich die Unterklasse im gleichen oder in einem anderen Paket befindet.

Diese Funktionalität ähnelt dem `private`-Attribut in ActionScript 2.0. Das `protected`-Attribut aus ActionScript 3.0 ähnelt auch dem `protected`-Attribut in Java. Der Unterschied besteht darin, dass die Java-Version auch den Zugriff auf aufrufende Objekte im selben Paket ermöglicht. Das `protected`-Attribut eignet sich insbesondere dann, wenn Unterklassen eine Variable oder Methode erfordern, der Code jedoch außerhalb der Vererbungskette nicht sichtbar sein soll.

Das in ActionScript 3.0 neu eingeführte Attribut `internal` macht eine Eigenschaft für aufrufende Objekte innerhalb des eigenen Pakets sichtbar. Dies ist das Standardattribut für Code innerhalb eines Pakets. Es gilt für jede Eigenschaft, die keines der folgenden Attribute aufweist:

- `public`
- `private`
- `protected`
- ein benutzerdefinierter Namespace

Das `internal`-Attribut ähnelt der Standard-Zugriffskontrolle in Java, obwohl es in Java keinen expliziten Namen für diese Zugriffsebene gibt und sie nur durch Weglassen eines anderen Zugriffsmodifizierers erreicht werden kann. Mit dem Attribut `internal` in ActionScript 3.0 können Sie explizit festlegen, dass eine Eigenschaft nur für aufrufende Objekte innerhalb des eigenen Pakets sichtbar ist.

static-Attribut

Das `static`-Attribut, das mit Eigenschaften verwendet werden kann, die mit den Schlüsselwörtern `var`, `const` oder `function` deklariert wurden, ermöglicht Ihnen das Anhängen einer Eigenschaft an die Klasse anstatt an Instanzen der Klasse. Code, der sich außerhalb der Klasse befindet, muss statische Eigenschaften mit dem Klassennamen anstelle des Instanznamens aufrufen.

Statische Eigenschaften werden von Unterklassen nicht übernommen, aber die Eigenschaften sind Teil der Gültigkeitsbereichskette der Unterklasse. Dies bedeutet, dass eine statische Variable oder Methode im Unterklassenrumpf verwendet werden kann, ohne dass auf die Klasse verwiesen wird, in der sie definiert wurde.

Benutzerdefinierte Namespace-Attribute

Als Alternative zu den vordefinierten Attributen der Zugriffskontrolle können Sie einen benutzerdefinierten Namespace erstellen, der als Attribut verwendet werden soll. Es kann nur ein Namespace-Attribut pro Definition verwendet werden, und Sie können ein Namespace-Attribut nicht in Verbindung mit einem der Zugriffskontrollattribute (`public`, `private`, `protected`, `internal`) verwenden.

Variablen

Variablen können mit den Schlüsselwörtern `var` oder `const` deklariert werden. Mit dem Schlüsselwort `var` deklarierte Variablen können ihre Werte während der Ausführung eines Skript mehrmals ändern. Mit dem Schlüsselwort `const` deklarierte Variablen werden als *Konstanten* bezeichnet. Konstanten kann nur einmal ein Wert zugewiesen werden. Der Versuch, einer bereits initialisierten Konstanten einen neuen Wert zuzuweisen, führt zu einer Fehlermeldung.

Statische Variablen

Statische Variablen werden mit einer Kombination aus dem Schlüsselwort `static` und der Anweisung `var` oder der Anweisung `const` deklariert. Statische Variablen, die an eine Klasse und nicht an eine Instanz einer Klasse angehängt sind, eignen sich insbesondere zum Speichern und gemeinsamen Nutzen von Informationen, die für eine gesamte Objektklasse gelten. So können Sie eine statische Variable einsetzen, wenn Sie zählen möchten, wie oft eine Klasse instanziiert wurde, oder wenn der Höchstwert für zulässige Klasseninstanzen gespeichert werden soll.

Im folgenden Beispielcode werden eine `totalCount`-Variable (um die Anzahl der Klasseninstanziierungen zu verfolgen) und eine `MAX_NUM`-Konstante erstellt, in der die Höchstzahl an zulässigen Instanziierungen gespeichert wird. Die Variablen `totalCount` und `MAX_NUM` sind statische Variablen, da sie Werte enthalten, die für die gesamte Klasse und nicht für eine bestimmte Instanz gelten.

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

Code außerhalb der StaticVars-Klasse und einer ihrer Unterklassen kann nur über die Klasse selbst auf die Eigenschaften `totalCount` und `MAX_NUM` verweisen. Der folgende Code arbeitet korrekt:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

Sie können nicht über die Instanz der Klasse auf statische Variablen zugreifen, daher gibt der folgende Code Fehlermeldungen zurück:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Mit den Schlüsselwörtern `static` und `const` deklarierte Variablen müssen zusammen mit der Konstantendeklaration initialisiert werden, wie es die StaticVars-Klasse für `MAX_NUM` durchführt. Sie können `MAX_NUM` innerhalb eines Konstruktors oder einer Instanzmethode keinen Wert zuweisen. Der folgende Code erzeugt eine Fehlermeldung, da es sich nicht um eine gültige Methode zum Initialisieren einer statischen Konstante handelt:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Instanzvariablen

Instanzvariablen enthalten Eigenschaften, die mit den Schlüsselwörtern `var` und `const`, aber ohne das Schlüsselwort `static` deklariert wurden. Instanzvariablen, die nicht an die gesamte Klasse, sondern an Klasseninstanzen angehängt werden, dienen unter anderem zum Speichern von Werten, die speziell für eine Instanz gelten. Beispielsweise verfügt die Array-Klasse über eine Instanzeigenschaft mit dem Namen `length`, die verschiedene Array-Elemente speichert, die eine bestimmte Instanz der Array-Klasse enthält.

Als `var` oder `const` deklarierte Instanzvariablen können in einer Unterklasse nicht überschrieben werden. Mit den get- und set-Methoden können Sie jedoch eine Funktionalität erreichen, die dem Überschreiben von Variablen ähnelt.

Methoden

Methoden sind Funktionen, die einen Teil einer Klassendefinition bilden. Nachdem eine Instanz der Klasse erstellt wurde, ist eine Methode an diese Instanz gebunden. Im Gegensatz zu einer außerhalb einer Klasse deklarierten Funktion kann eine Methode nicht außerhalb der Instanz verwendet werden, an die sie angefügt ist.

Methoden werden mit dem Schlüsselwort `function` definiert. Wie bei jeder Klasseneigenschaft können Sie jedes Klasseneigenschaftsattribut auf Methoden anwenden, einschließlich der Attribute „private“, „protected“, „public“, „internal“, „static“ oder eines benutzerdefinierten Namespace. Sie können eine Funktionsanweisung wie die folgende verwenden:

```
public function sampleFunction():String {}
```

Sie können auch eine Variable verwenden, der ein Funktionsausdruck zugewiesen wird. Dies wird im folgenden Beispiel gezeigt:

```
public var sampleFunction:Function = function () {}
```

In den meisten Fällen verwenden Sie aus den folgenden Gründen eine Funktionsanweisung anstelle eines Funktionsausdrucks:

- Funktionsanweisungen sind kompakter und besser lesbar.
- Funktionsanweisungen ermöglichen Ihnen das Verwenden der Schlüsselwörter `override` und `final`.

- Funktionsanweisungen erstellen eine stärkere Bindung zwischen dem Bezeichner (also dem Namen der Funktion) und dem Code im Methodenrumpf. Der Wert einer Variablen kann mit einer Zuweisungsanweisung geändert werden, somit kann die Verbindung zwischen einer Variablen und ihrem Funktionsausdruck jederzeit aufgehoben werden. Sie können dieses Problem durch Deklarieren der Variablen mit `const` anstelle von `var` umgehen. Allerdings wird von dieser Vorgehensweise abgeraten, da sie den Code schwerer lesbar macht und das Verwenden der Schlüsselwörter `override` und `final` verhindert.

Ein Fall, bei dem Sie einen Funktionsausdruck verwenden, ist wenn Sie eine Funktion an das Prototypobjekt anhängen möchten.

Konstruktormethoden

Konstruktormethoden (manchmal auch als *Konstruktoren* bezeichnet) sind Funktionen, die den gleichen Namen wie die Klassen haben, in denen sie definiert wurden. Ein in eine Konstruktormethode aufgenommener Code wird immer dann ausgeführt, wenn eine Instanz der Klasse mit dem Schlüsselwort `new` erstellt wird. Im folgenden Beispielcode wird eine einfache Klasse namens „Example“ definiert, die eine Eigenschaft mit der Bezeichnung `status` enthält. Der Startwert der Variablen `status` ist in der Konstruktorfunktion festgelegt.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Konstruktormethoden können nur öffentlich sein, aber die Verwendung des Attributs `public` ist optional. Sie können keinen der anderen Zugriffskontrollbezeichner wie `private`, `protected` oder `internal` für einen Konstruktor verwenden. Außerdem ist es nicht möglich, einen benutzerdefinierten Namespace mit einer Konstruktormethode zu verwenden.

Mit der Anweisung `super()` kann ein Konstruktor den Konstruktor der direkt übergeordneten Klasse aufrufen. Wenn der Konstruktor der übergeordneten Klasse nicht explizit aufgerufen wird, fügt der Compiler automatisch einen Aufruf vor der ersten Anweisung in den Konstruktorrumpf ein. Sie können die Methoden der übergeordneten Klasse auch mit dem Präfix `super` als Verweis auf die übergeordnete Klasse aufrufen. Wenn Sie sowohl `super()` als auch `super` im gleichen Konstruktorrumpf verwenden möchten, achten Sie darauf, zuerst `super()` aufzurufen. Andernfalls verhält sich der `super`-Verweis nicht wie erwartet. Außerdem muss der `super()`-Konstruktor vor allen `throw`- oder `return`-Anweisungen aufgerufen werden.

Im folgenden Beispielcode wird gezeigt, was geschieht, wenn Sie versuchen, den Verweis `super` vor dem Aufruf des `super()`-Konstruktors zu verwenden. Eine neue Klasse, „ExampleEx“, erweitert die Example-Klasse. Der ExampleEx-Konstruktor versucht, auf die in der übergeordneten Klasse definierte Statusvariable zuzugreifen, jedoch vor dem Aufruf von `super()`. Die `trace()`-Anweisung im ExampleEx-Konstruktor erzeugt den Wert `null`, da die `status`-Variable erst verfügbar ist, nachdem der `super()`-Konstruktor ausgeführt wurde.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Obwohl die `return`-Anweisung in einem Konstruktor zulässig ist, darf kein Wert zurückgegeben werden. Anders ausgedrückt, `return`-Anweisungen dürfen keine Ausdrücke oder Werte zugewiesen werden. Entsprechend dürfen auch Konstruktormethoden keine Werte zurückgeben. Dies bedeutet, dass kein Rückgabetyp angegeben werden kann.

Wenn Sie keine Konstruktormethode in Ihrer Klasse definieren, erstellt der Compiler automatisch einen leeren Konstruktor für Sie. Wenn Ihre Klasse eine andere Klasse erweitert, nimmt der Compiler einen `super()`-Aufruf in den erstellten Konstruktor auf.

Statische Methoden

Statische Methoden, die auch als *Klassenmethoden* bezeichnet werden, sind Methoden, die mit dem Schlüsselwort `static` deklariert werden. Statische Methoden, die an eine Klasse und nicht an eine Instanz einer Klasse angehängt werden, eignen sich insbesondere zur Kapselung von Funktionsmerkmalen, die sich auf etwas anderes auswirken als auf den Zustand einer bestimmten Instanz. Da statische Methoden an eine gesamte Klasse angehängt werden, kann nur über eine Klasse und nicht über die Instanz der Klasse auf statische Methoden zugegriffen werden.

Statische Methoden eignen sich besonders zur Kapselung von Funktionen, die sich nicht nur auf den Zustand von Klasseninstanzen auswirken. Anders ausgedrückt, eine Methode sollte statisch sein, wenn sie Funktionen bereitstellt, die sich nicht direkt auf den Wert einer Klasseninstanz auswirken. Beispielsweise verfügt die Date-Klasse über eine statische Methode mit der Bezeichnung `parse()`, die einen String in eine Zahl umwandelt. Die Methode ist statisch, da sie keine Auswirkungen auf eine einzelne Instanz der Klasse hat. Stattdessen arbeitet die Methode `parse()` mit einem String, der einen Datumswert darstellt, analysiert den String und gibt eine Zahl in dem Format zurück, das mit der internationalen Darstellung eines Date-Objekts kompatibel ist. Diese Methode ist keine Instanzmethode, da es keinen Sinn ergibt, die Methode auf eine Instanz der Date-Klasse anzuwenden.

Vergleichen Sie die statische Methode `parse()` mit einer der Instanzmethoden der Date-Klasse, wie z. B. `getMonth()`. Die `getMonth()`-Methode ist eine Instanzmethode, da sie direkt den Wert einer Instanz einbezieht, indem sie eine bestimmte Komponente (den Monat) der Date-Instanz abruft.

Da statische Methoden nicht an einzelne Instanzen gebunden sind, können die Schlüsselwörter `this` oder `super` nicht im Rumpf einer statischen Methode verwendet werden. Die Verweise `this` und `super` sind nur innerhalb des Kontextes der Instanzmethode gültig.

Im Gegensatz zu anderen klassenbasierten Programmiersprachen werden statische Methoden in ActionScript 3.0 nicht geerbt.

Instanzmethoden

Instanzmethoden sind Methoden, die ohne das Schlüsselwort `static` deklariert werden. Instanzmethoden, die an Instanzen einer Klasse und nicht an eine gesamte Klasse angehängt werden, eignen sich insbesondere zum Implementieren von Funktionen, die sich auf einzelne Instanzen einer Klasse auswirken. Beispielsweise enthält die Array-Klasse eine Instanzmethode namens `sort()`, die Array-Instanzen direkt einbezieht.

Innerhalb eines Instanzmethodenrumpfs sind sowohl statische als auch Instanzvariablen im Gültigkeitsbereich. Dies bedeutet, dass in der gleichen Klasse definierte Variablen mithilfe eines einfachen Bezeichners referenziert werden können. Beispielsweise erweitert die folgende Klasse, „CustomArray“, die Array-Klasse. Die CustomArray-Klasse definiert eine statische Variable namens `arrayCountTotal`, mit der die Gesamtzahl der Klasseninstanzen verfolgt wird, eine Instanzvariable namens `arrayNumber`, mit der die Reihenfolge verfolgt wird, in der die Instanzen erstellt werden, sowie eine Instanzmethode namens `getPosition()`, mit der die Werte dieser Variablen zurückgegeben werden.

```
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);
    }
}
```

Obwohl Code außerhalb der Klasse über das Klassenobjekt mit `CustomArray.arrayCountTotal` auf die statische Variable `arrayCountTotal` zugreifen muss, kann Code, der sich im Rumpf der `getPosition()`-Methode befindet, direkt auf die statische Variable `arrayCountTotal` verweisen. Dies gilt sogar für statische Variablen in übergeordneten Klassen. Obwohl statische Eigenschaften in ActionScript 3.0 nicht vererbt werden, befinden sich auch die statischen Eigenschaften in übergeordneten Klassen innerhalb des Gültigkeitsbereichs. Beispielsweise verfügt die Array-Klasse über einige statische Variablen, eine davon ist eine Konstante namens `DESCENDING`. Code, der sich in einer Array-Unterklasse befindet, kann mithilfe eines einfachen Bezeichners auf die statische Konstante `DESCENDING` zugreifen:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

Der Wert des Verweises `this` im Rumpf einer Instanzmethode ist ein Verweis auf die Instanz, an die die Methode angefügt ist. Im folgenden Code wird veranschaulicht, dass der Verweis `this` auf die Instanz zeigt, in der die Methode enthalten ist:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Die Vererbung von Instanzmethoden kann mit den Schlüsselwörtern `override` und `final` gesteuert werden. Mit dem Attribut `override` können Sie eine geerbte Methode neu definieren, und mit dem Attribut `final` können Sie verhindern, dass Unterklassen eine Methode überschreiben.

get- und set-Accessormethoden

Mit den get- und set-Accessorfunktionen, die auch als *getter* und *setter* bezeichnet werden, können Sie bei der Programmierung an den Prinzipien des Ausblendens und der Kapselung von internen Daten festhalten, während gleichzeitig eine benutzerfreundliche Programmierschnittstelle für von Ihnen erstellte Klassen bereitgestellt wird. Mit get- und set-Funktionen bleiben Ihre Klasseneigenschaften privat in der Klasse, aber anderen Benutzern Ihrer Klasse wird der Zugriff auf diese Eigenschaften so gestattet, als ob sie auf eine Klassenvariable zugreifen, anstatt eine Klassenmethode aufzurufen.

Der Vorteil dieses Ansatzes liegt darin, dass Sie die traditionellen Accessorfunktionen mit ihren sperrigen Namen wie `getPropertyName()` und `setPropertyName()` vermeiden können. Ein weiterer Vorteil der get- und set-Methoden besteht darin, dass sie zwei öffentliche Funktionen für jede Eigenschaft vermeiden, die Lese- und Schreibzugriff gestatten.

Die folgende Beispielklasse mit der Bezeichnung „GetSet“ enthält ein get- und set-Accessorfunktionspaar namens `publicAccess()`, mit dem auf die private Variable namens `privateProperty` zugegriffen werden kann:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Wenn Sie versuchen, direkt auf die `privateProperty`-Eigenschaft zuzugreifen, tritt ein Fehler auf:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Stattdessen verwendet ein Benutzer der GetSet-Klasse etwas, das eine Eigenschaft namens `publicAccess` zu sein scheint, in Wirklichkeit aber ein get- und set-Accessorfunktionspaar ist, das mit einer privaten Eigenschaft namens `privateProperty` arbeitet. Im folgenden Beispielcode wird die GetSet-Klasse instanziiert und dann der Wert von `privateProperty` mithilfe des öffentlichen Accessors `publicAccess` eingestellt:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Mit get- und set-Funktionen können auch die von einer übergeordneten Klasse übernommenen Eigenschaften überschrieben werden. Dies ist mit normalen Klassenmitgliedervariablen nicht möglich. Mit dem Schlüsselwort `var` deklarierte Klassenmitgliedervariablen können in einer Unterklasse nicht überschrieben werden. Eigenschaften, die mit get- und set-Funktionen erstellt wurden, weisen diese Einschränkung nicht auf. Für get- und set-Funktionen, die von einer übergeordneten Klasse geerbt wurden, können Sie das Attribut `override` verwenden.

Gebundene Methoden

Eine gebundene Methode, die manchmal auch als *Methodenhülle* (Method Closure) bezeichnet wird, ist im Grunde genommen eine Methode, die aus ihrer Instanz extrahiert wurde. Beispiele für gebundene Methoden sind Methoden, die als Argumente an eine Funktion übergeben oder als Werte von einer Funktion zurückgegeben werden. Gebundenen Methoden wurde in ActionScript 3.0 neu eingeführt. Sie ähneln in gewisser Weise einer Funktionshülle, da sie die lexikalische Umgebung auch dann beibehalten, wenn sie aus ihrer Instanz extrahiert wurden. Der wesentliche Unterschied zwischen einer gebundene Methode und einer Funktionshülle besteht jedoch darin, dass der Verweis `this` bei einer gebundenen Methode mit der Instanz verknüpft oder an die Instanz „gebunden" bleibt, welche die Methode implementiert. Anders ausgedrückt, der Verweis `this` in einer gebundenen Methode zeigt immer auf das Ursprungsobjekt, das die Methode implementiert. Bei Funktionshüllen ist der `this`-Verweis generisch, d. h. er zeigt auf jedes Objekt, das der Funktion zum Zeitpunkt des Aufrufs zugewiesen ist.

Das Konzept der gebundenen Methoden ist für das Schlüsselwort `this` extrem wichtig. Zur Erinnerung: Das Schlüsselwort `this` stellt einen Verweis auf eine Methode des übergeordneten Objekts bereit. Die meisten ActionScript-Programmierer gehen davon aus, dass das `this`-Schlüsselwort immer die Klasse bzw. das Objekt mit der Definition einer Methode darstellt. Ohne Methodenbindung ist dies jedoch nicht immer richtig. In früheren Versionen von ActionScript referenzierte der Verweis `this` nicht immer die Instanz, welche die Methode implementierte. Wenn Methoden in ActionScript 2.0 aus einer Instanz extrahiert werden, ist nicht nur der Verweis `this` nicht an die Ursprungsinstanz gebunden, sondern auch die Mitgliedervariablen und -methoden der Instanzklasse stehen nicht zur Verfügung. Dies stellt in ActionScript 3.0 kein Problem dar, da gebundene Methoden automatisch erstellt werden, wenn Sie eine Methode als Parameter übergeben. Gebundene Methoden stellen sicher, dass das Schlüsselwort `this` immer auf das Objekt oder die Klasse verweist, in dem bzw. der eine Methode definiert ist.

Im folgenden Beispielcode wird eine Klasse namens „ThisTest" erstellt, die eine Methode namens `foo()` enthält. Diese wiederum definiert eine gebundene Methode sowie eine Methode mit der Bezeichnung `bar()`, welche die gebundene Methode zurückgibt. Code außerhalb der Klasse erstellt eine Instanz der ThisTest-Klasse, ruft die Methode `bar()` auf und speichert den Rückgabewert in einer Variablen namens `myFunc`.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

Die letzten beiden Codezeilen zeigen, dass der Verweis `this` in der gebundenen Methode `foo()` noch immer auf eine Instanz der ThisTest-Klasse zeigt, obwohl der Verweis `this` in der Zeile direkt davor auf das globale Objekt zeigt. Darüber hinaus hat die in der gebundenen Methode gespeicherte Variable `myFunc` noch immer Zugriff auf die Mitgliedervariablen der ThisTest-Klasse. Wird derselbe Code in ActionScript 2.0 ausgeführt, stimmen beide Aufrufe von `this` überein und die Variable `num` ist `undefined`.

Ein Bereich, in dem sich die Einführung von gebundenen Methoden besonders bemerkbar macht, sind Ereignisprozeduren, da die Methode `addEventListener()` erfordert, dass Sie eine Funktion oder eine Methode als Argument übergeben.

Aufzählungen mit Klassen

Aufzählungen (Englisch „enumerations“) sind benutzerdefinierte Datentypen, die Sie zur Kapselung einer kleinen Wertegruppe erstellen. ActionScript 3.0 unterstützt keine besondere Aufzählungsfunktion, wie dies in C++ mit dem Schlüsselwort `enum` oder in Java mit der Enumeration-Schnittstelle der Fall ist. Sie können Aufzählungen jedoch mit Klassen und statischen Konstanten erstellen. Beispielsweise verwendet die PrintJob-Klasse in ActionScript 3.0 eine Aufzählung mit der Bezeichnung „PrintJobOrientation“, um die Werte `"landscape"` und `"portrait"` zu speichern. Dies wird im folgenden Code gezeigt:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

In der Standardeinstellung wird eine Aufzählungsklasse mit dem Attribut `final` deklariert, da es nicht erforderlich ist, die Klasse zu erweitern. Die Klasse enthält ausschließlich statische Mitglieder, daher müssen Sie keine Instanzen der Klasse erstellen. Stattdessen greifen Sie direkt über das Klassenobjekt auf die Aufzählungswerte zu. Dies wird im folgenden Codeausschnitt gezeigt:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Alle Aufzählungsklassen in ActionScript 3.0 enthalten nur Variablen des Typs „String“, „int“ oder „uint“. Der Vorteil bei der Verwendung von Aufzählungen anstelle von literalen Strings oder Zahlenwerten besteht darin, dass typografische Fehler in Aufzählungen leichter zu finden sind. Wenn Sie den Namen einer Aufzählung falsch eingeben, erzeugt der ActionScript-Compiler einen Fehler. Wenn Sie literale Werte verwenden, wird der Compiler ein falsch geschriebenes Wort nicht bemerken oder eine falsche Zahl verwenden. Im vorangegangenen Beispiel erzeugt der Compiler einen Fehler, wenn der Name der Aufzählungskonstanten falsch ist. Dies wird im folgenden Codeausschnitt gezeigt:

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Der Compiler erzeugt jedoch keinen Fehler, wenn Sie den literalen Wert eines Strings falsch schreiben:

```
    if (pj.orientation == "portrai") // no compiler error
```

Bei einer anderen Technik zur Erstellung von Aufzählungen wird ebenfalls eine separate Klasse mit statischen Eigenschaften für die Aufzählung definiert. Diese Technik unterscheidet sich jedoch insofern von der ersten, als dass jede statische Eigenschaft eine Instanz der Klasse anstelle eines Strings oder einer ganzen Zahl enthält. Im folgenden Beispielcode wird eine Aufzählungsklasse für die Tage der Woche erstellt:

```
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();
}
```

Diese Technik wird von ActionScript 3.0 nicht genutzt, jedoch von vielen Entwicklern verwendet, die von der verbesserten Typüberprüfung profitieren möchten, die diese Technik bietet. Beispielsweise kann eine Methode, die einen Aufzählungswert zurückgibt, den Rückgabewert auf den Datentyp der Aufzählung einschränken. Der folgende Code enthält nicht nur eine Funktion, die einen Tag der Woche zurückgibt, sondern auch einen Funktionsaufruf, der den Datentyp der Aufzählung als Typanmerkung verwendet:

```
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();
```

Sie können die Day-Klasse auch erweitern, sodass sie jedem Wochentag eine ganze Zahl zuweist und eine `toString()`-Methode bereitstellt, die eine Zeichenfolge mit dem Wochentag zurückgibt.

Eingebettete Bestandsklassen

Zum Darstellen von eingebettetem Bestand (Englisch „embedded assets“) verwendet ActionScript 3.0 spezielle Klassen, so genannte *eingebettete Bestandsklassen*. Ein *eingebetteter Bestand* ist ein Bestand, z. B. ein Sound, ein Bild oder eine Schriftart, der während der Kompilierung in einer SWF-Datei enthalten ist. Das Einbetten eines Bestands anstelle des dynamischen Ladens stellt sicher, dass der Bestand zur Laufzeit verfügbar ist. Es bedeutet jedoch auch eine größere SWF-Datei.

Verwenden eingebetteter Bestandsklassen in Flash Professional

Um einen Bestand einzubetten, platzieren Sie diesen zunächst in der Bibliothek einer FLA-Datei. Verwenden Sie anschließend die linkage-Eigenschaft des Bestands, um einen Namen für die eingebettete Bestandsklasse des Bestands anzugeben. Wenn im Klassenpfad keine Klasse mit diesem Namen vorhanden ist, wird automatisch eine entsprechende Klasse erstellt. Sie können dann eine Instanz der eingebetteten Bestandsklasse erstellen und alle Eigenschaften und Methoden verwenden, die für diese Klasse definiert oder von ihr geerbt wurden. Beispielsweise kann der folgende Code verwendet werden, um einen eingebetteten Sound wiederzugeben, der mit einer eingebetteten Bestandsklasse mit dem Namen „PianoMusic“ verknüpft ist:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Alternativ können Sie das `[Embed]`-Metadaten-Tag verwenden, um Bestände in einem Flash Professional-Projekt einzubetten wie nachstehend beschrieben. Wenn Sie das `[Embed]`-Metadaten-Tag in Ihrem Code einsetzen, verwendet Flash Professional den Flex-Compiler anstelle des Flash Professional-Compilers zur Kompilierung des Projekts.

Verwenden von eingebetteten Bestandsklassen mit dem Flex-Compiler

Wenn Sie Ihren Code mit dem Flex-Compiler kompilieren, verwenden Sie das `[Embed]`-Metadaten-Tag, um einen Bestand im ActionScript-Code einzubetten. Platzieren Sie den Bestand im Hauptquellordner oder einem anderen Ordner, der sich im Erstellungspfad Ihres Projekts befindet. Wenn der Flex-Compiler auf ein Embed-Metadaten-Tag trifft, erstellt er die eingebettete Bestandsklasse. Sie können auf die Klasse über eine Variable des Class-Datentyps zugreifen, die Sie unmittelbar nach dem `[Embed]`-Metadaten-Tag deklarieren. Der folgende Beispielcode bettet einen Sound namens „sound1.mp3“ ein und verwendet eine Variable namens `soundCls` zum Speichern eines Verweises auf die eingebettete Bestandsklasse, die diesem Sound zugewiesen ist. Anschließend erstellt der Beispielcode eine Instanz der eingebetteten Bestandsklasse und ruft die `play()`-Methode für diese Instanz auf:

```
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

Um das `[Embed]`-Metadaten-Tag in einem Flash Builder ActionScript-Projekt zu verwenden, importieren Sie die benötigten Klassen aus dem Flex-Framework. Zum Einbetten von Sounds importieren Sie beispielsweise die mx.core.SoundAsset-Klasse. Um die Flex-Architektur zu verwenden, nehmen Sie die Datei „framework.swc“ in den ActionScript-Erstellungspfad auf. Hierdurch vergrößert sich die SWF-Datei.

Adobe Flex

Alternativ können Sie einen Bestand in Flex auch mit der `@Embed()`-Direktive in einer MXML-Tag-Definition einbetten.

Schnittstellen

Eine Schnittstelle (Englisch „interface“) ist eine Sammlung von Methodendeklarationen, über die nicht miteinander verwandte Objekte miteinander kommunizieren können. Beispielsweise definiert ActionScript 3.0 die IEventDispatcher-Schnittstelle, die Methodendeklarationen enthält, die eine Klasse zur Verarbeitung von Ereignisobjekten verwenden kann. Die IEventDispatcher-Schnittstelle ist ein Standardverfahren für Objekte, einander Ereignisobjekte zu übergeben. Der folgende Code zeigt die Definition der IEventDispatcher-Schnittstelle:

```
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;
}
```

Schnittstellen basieren auf dem Unterschied zwischen der Schnittstelle einer Methode und ihrer Implementation. Die Schnittstelle einer Methode umfasst alle Informationen, die zum Aufrufen dieser Methode notwendig sind, einschließlich des Namens der Methode, aller ihrer Parameter sowie ihres Rückgabetyps. Die Implementation einer Methode umfasst nicht nur die Schnittstelleninformationen, sondern auch die ausführbaren Anweisungen, die das Verhalten der Methode ausführen. Eine Schnittstellendefinition enthält nur die Schnittstellen der Methode. Jede Klasse, welche die Schnittstelle implementiert, ist für die Definition der Methodenimplementationen selbst verantwortlich.

In ActionScript 3.0 implementiert die EventDispatcher-Klasse die IEventDispatcher-Schnittstelle, indem sie alle IEventDispatcher-Schnittstellenmethoden definiert und jeder Methode die Methodenrümpfe hinzufügt. Der folgende Code ist ein Auszug der EventDispatcher-Klassendefinition:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

Die IEventDispatcher-Schnittstelle dient als Protokoll, das die EventDispatcher-Instanzen verwenden, um Ereignisobjekte zu verarbeiten und an andere Objekte zu übergeben, die ebenfalls von der IEventDispatcher-Schnittstelle implementiert werden.

Anders ausgedrückt könnte man über eine Schnittstelle auch sagen, dass sie einen Datentyp genauso wie eine Klasse definiert. Entsprechend kann eine Schnittstelle wie eine Klasse als Typanmerkung verwendet werden. Als Datentyp kann eine Schnittstelle auch mit Operatoren wie `is` und `as` verwendet werden, die einen Datentyp benötigen. Im Gegensatz zu einer Klasse kann eine Schnittstelle jedoch nicht instanziiert werden. Dieser Unterschied hat dazu geführt, dass sich viele Programmierer Schnittstellen als abstrakte Datentypen und Klassen als konkrete Datentypen vorstellen.

Definieren einer Schnittstelle

Die Struktur einer Schnittstellendefinition ähnelt der einer Klassendefinition, außer dass eine Schnittstelle nur Methoden ohne Methodenrümpfe enthalten kann. Schnittstellen können get- und set-Methoden, jedoch keine Variablen oder Konstanten enthalten. Zur Definition einer Schnittstelle verwenden Sie das Schlüsselwort `interface`. Beispielsweise handelt es sich bei der Schnittstelle „IExternalizable“ um einen Teil des flash.utils-Pakets in ActionScript 3.0. Die IExternalizable-Schnittstelle definiert ein Protokoll zur Serialisierung eines Objekts. Dies bedeutet die Schnittstelle wandelt ein Objekt in ein Format um, das zur Speicherung auf einem Gerät oder für den Transport über ein Netzwerk geeignet ist.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

Die IExternalizable-Schnittstelle wird mit dem `public`-Zugriffskontrollmodifizierer deklariert. Schnittstellendefinitionen können nur über die Zugriffskontrollbezeichner `public` und `internal` modifiziert werden. Die Methodendeklarationen in einer Schnittstellendefinition können keine Zugriffskontrollbezeichner enthalten.

ActionScript 3.0 folgt der Konvention, den Schnittstellennamen mit einem Großbuchstaben `I` zu beginnen; Sie können jedoch auch jeden anderen zulässigen Bezeichner als Schnittstellennamen verwenden. Schnittstellendefinitionen werden häufig auf der obersten Ebene eines Pakets platziert. Schnittstellendefinitionen können nicht in einer Klassendefinition oder einer anderen Schnittstellendefinition platziert werden.

Schnittstellen können eine oder mehrere andere Schnittstellen erweitern. Beispielsweise erweitert die folgende IExample-Schnittstelle die IExternalizable-Schnittstelle:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Jede Klasse, welche die IExample-Schnittstelle implementiert, muss Implementationen nicht nur für die `extra()`-Methode, sondern auch für die von der IExternalizable-Schnittstelle geerbten Methoden `writeExternal()` und `readExternal()` umfassen.

Implementieren einer Schnittstelle in einer Klasse

Eine Klasse ist das einzige Sprachelemente in ActionScript 3.0, das eine Schnittstelle implementieren kann. Verwenden Sie das Schlüsselwort `implements` in einer Klassendeklaration, um eine oder mehrere Schnittstellen zu implementieren. Im folgenden Codebeispiel werden die beiden Schnittstellen „IAlpha“ und „IBeta“ sowie eine Klasse `Alpha` definiert, die beide Schnittstellen implementiert:

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

In einer Klasse, die eine Schnittstelle implementiert, müssen implementierte Methoden:

- Den Zugriffskontrollbezeichner `public` verwenden.
- Den gleichen Namen wie die Schnittstellenmethode verwenden.
- Die gleiche Anzahl an Parametern aufweisen, jeder mit dem Datentyp, der dem Datentyp der Parameter in der Schnittstellenmethode entspricht.
- Den gleichen Rückgabetyp verwenden.

```
public function foo(param:String):String {}
```

Bei der Benennung der Parameter der von Ihnen implementierten Methoden haben Sie eine gewisse Flexibilität. Obwohl Anzahl der Parameter und Datentyp jedes Parameters in der implementierten Methode denen der Schnittstellenmethode entsprechen müssen, müssen die Parameternamen nicht übereinstimmen. So lautet der Name des Parameters der Methode `Alpha.foo()` aus dem vorangegangenen Beispiel `param`:

In der Schnittstellenmethode `IAlpha.foo()` lautet der Name des Parameters hingegen `str`:

```
function foo(str:String):String;
```

Auch bei den Standard-Parameterwerten haben Sie eine gewisse Flexibilität. Eine Schnittstellendefinition kann Funktionsdeklarationen mit Standard-Parameterwerten enthalten. Eine Methode, die eine solche Funktionsdeklaration implementiert, muss einen Standard-Parameterwert aufweisen, der Mitglied des gleichen Datentyps wie der in der Schnittstellendefinition angegebene Wert ist, der tatsächliche Wert muss jedoch nicht übereinstimmen. Beispielsweise definiert der folgende Code eine Schnittstelle, die eine Methode mit einem Standard-Parameterwert 3 enthält:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

Die folgende Klassendefinition implementiert die IGamma-Schnittstelle, verwendet jedoch einen anderen Standard-Parameterwert:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Der Grund für diese Flexibilität liegt darin, dass die Datentypkompatibilität mit den Regeln zur Implementierung einer Schnittstelle sichergestellt wird und hierfür keine identischen Parameternamen und Standard-Parameterwerte erforderlich sind.

Vererbung

Vererbung ist eine Form der Wiederverwendung von Code, mit der Programmierer neue Klassen entwickeln können, die auf bereits bestehenden Klassen basieren. Die bestehenden Klassen werden häufig als *Basisklassen* oder *übergeordnete Klassen* (Englisch „superclasses“) bezeichnet, während die neuen Klassen häufig als *Unterklassen* (Englisch „subclasses“) bezeichnet werden. Ein wesentlicher Vorteil der Vererbung besteht darin, dass Sie Code aus einer Basisklasse wiederverwenden können, ohne die vorhandene Klasse zu modifizieren. Darüber hinaus erfordert die Vererbung keine Änderungen der Art und Weise, wie andere Klassen mit der Basisklasse interagieren. Anstatt eine sorgfältig getestete oder eventuell sogar eingesetzte vorhandene Klasse zu modifizieren, können Sie eine Klasse mit der Vererbung als integriertes Modul behandeln, das Sie um zusätzliche Eigenschaften oder Methoden erweitern können. Entsprechend verwenden Sie das Schlüsselwort `extends`, um anzugeben, dass eine Klasse von einer anderen Klasse erbt.

Mit der Vererbung können Sie sogar von den Vorteilen des *Polymorphismus* in Ihrem Code profitieren. Unter Polymorphismus versteht man die Möglichkeit, einen einzelnen Methodennamen für eine Methode zu verwenden, die sich je nach Datentyp, für den sie angewendet wird, anders verhält. Ein einfaches Beispiel ist eine Basisklasse mit der Bezeichnung „Shape“ mit zwei Unterklassen namens „Circle“ und „Square“. Die Shape-Klasse definiert eine Methode namens `area()`, welche die Fläche der Form zurückgibt. Ist Polymorphismus implementiert, können Sie die Methode `area()` für die Objekte „Circle“ und „Square“ aufrufen, und es werden die richtigen Berechnungen für Sie durchgeführt. Die Vererbung ermöglicht Polymorphismus, indem sie Unterklassen das Erben und Neudefinieren bzw. *Überschreiben* (Englisch „override“) von Methoden aus der Basisklasse gestattet. Im folgenden Beispielcode wird die Methode `area()` von den Klassen „Circle“ und „Square“ neu definiert:

```
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
```

Da jede Klasse einen Datentyp definiert, sorgt die Vererbung für eine besondere Beziehung zwischen einer Basisklasse und einer Klasse, die sie erweitert. Eine Unterklasse besitzt garantiert alle Eigenschaften ihrer Basisklasse. Dies bedeutet, dass die Instanz einer Unterklasse immer als Ersatz für eine Instanz der Basisklasse verwendet werden kann. Wenn eine Methode z. B. einen Parameter des Typs „Shape“ definiert, ist es zulässig, einen Parameter des Typs „Circle“ zu übergeben, da „Circle“ den Parameter „Shape“ erweitert. Dies wird im folgenden Beispiel gezeigt:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Instanzeigenschaften und Vererbung

Eine Instanzeigenschaft wird von allen Unterklassen geerbt, unabhängig davon, ob sie mit dem Schlüsselwort `function`, `var` oder `const` definiert wurde, solange die Eigenschaft in der Basisklasse nicht mit dem Attribut `private` deklariert wurde. Beispielsweise hat die Event-Klasse in ActionScript 3.0 eine Reihe von Unterklassen, die Eigenschaften erben, die alle Ereignisobjekte gemeinsam haben.

Für einige Ereignistypen enthält die Event-Klasse alle Eigenschaften, die zur Definition des Ereignisses erforderlich sind. Diese Ereignistypen benötigen keine Instanzeigenschaften über die Eigenschaften hinaus, die in der Event-Klasse definiert sind. Beispiele dieser Ereignisse sind `complete`, das nach dem erfolgreichen Laden von Daten eintritt, und `connect`, das nach dem erfolgreichen Herstellen einer Netzwerkverbindung eintritt.

Das folgende Beispiel ist ein Auszug der Event-Klasse, in dem einige der Eigenschaften und Methoden gezeigt werden, die von Unterklassen geerbt werden. Da diese Eigenschaften geerbt sind, kann jede Instanz einer Unterklasse darauf zugreifen.

```
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 {}
    ...
}
```

Andere Ereignistypen erfordern eindeutige Ereignisse, die in der Event-Klasse nicht zur Verfügung stehen. Diese Ereignisse werden mit Unterklassen der Event-Klasse definiert, sodass den bereits in der Event-Klasse definierten Eigenschaften neue hinzugefügt werden können. Ein Beispiel einer solchen Unterklasse ist die MouseEvent-Klasse, die Ereignisse hinzufügt, die nur für Mausbewegungen oder Mausklicks gelten, z. B. die Ereignisse `mouseMove` und `click`. Das folgende Beispiel ist ein Auszug der MouseEvent-Klasse, in dem die Definition der Eigenschaften dargestellt ist, die in der Unterklasse, jedoch nicht in der Basisklasse vorhanden sind:

```
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 {}
    ...
}
```

Zugriffskontrollbezeichner und Vererbung

Eine mit dem Schlüsselwort `public` deklarierte Eigenschaft ist für jeden Code sichtbar. Dies bedeutet, dass das Schlüsselwort `public` im Gegensatz zu den Schlüsselwörtern `private`, `protected` und `internal` keinerlei Einschränkungen bei der Eigenschaftenvererbung einführt.

Eine mit dem Schlüsselwort `private` deklarierte Eigenschaft ist nur in der Klasse sichtbar, in der sie definiert wurde. Dies bedeutet, dass sie nicht von Unterklassen geerbt werden kann. Dieses Verhalten weicht von dem in früheren ActionScript-Versionen ab, in denen sich das Schlüsselwort `private` mehr wie das ActionScript 3.0-Schlüsselwort `protected` verhielt.

Das Schlüsselwort `protected` kennzeichnet, dass eine Eigenschaft nicht nur für die Klasse sichtbar ist, in der sie definiert wurde, sondern für alle Unterklassen. Im Gegensatz zum Schlüsselwort `protected` in der Programmiersprache Java macht das Schlüsselwort `protected` in ActionScript 3.0 eine Eigenschaft nicht für alle Klassen im gleichen Paket sichtbar. In ActionScript 3.0 können nur Unterklassen auf eine Eigenschaft zugreifen, die mit dem Schlüsselwort `protected` deklariert wurde. Darüber hinaus ist eine geschützte Eigenschaft für eine Unterklasse sichtbar, unabhängig davon, ob sich die Unterklasse im gleichen Paket wie die Basisklasse oder in einem anderen Paket befindet.

Um die Sichtbarkeit einer Eigenschaft auf das Paket zu beschränken, in dem sie definiert wurde, verwenden Sie entweder das Schlüsselwort `internal` oder wenden gar keinen Zugriffskontrollbezeichner an. Der Zugriffskontrollbezeichner `internal` ist der Standard-Zugriffskontrollbezeichner. Er wird automatisch angewendet, wenn kein Zugriffskontrollbezeichner angegeben wurde. Eine als `internal` deklarierte Eigenschaft wird nur von einer Unterklasse geerbt, die sich im gleichen Paket befindet.

Im folgenden Beispielcode können Sie sehen, wie sich die Zugriffskontrollbezeichner auf die Vererbung über Paketgrenzen hinaus auswirken. Im folgenden Beispielcode werden eine Hauptanwendungsklasse namens „AccessControl“ sowie zwei weitere Klassen „Base“ und „Extender“ definiert. Die Base-Klasse befindet sich in einem Paket namens „foo“ und die Extender-Klasse (bei der es sich um eine Unterklasse der Base-Klasse handelt) in einem Paket namens „bar“. Die AccessControl-Klasse importiert nur die Extender-Klasse und erstellt dann eine Instanz der Extender-Klasse, die versucht, auf eine Variable namens `str` zuzugreifen, die in der Base-Klasse definiert ist. Die Variable `str` ist als `public` deklariert, sodass der Code wie im folgenden Codeauszug kompiliert und ausgeführt wird:

```
// 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
        }
    }
}
```

Um festzustellen, wie sich andere Zugriffskontrollbezeichner auf die Kompilierung und Ausführung des vorangegangenen Beispiels auswirken, ändern Sie den Zugriffskontrollbezeichner der Variablen `str` in `private`, `protected` oder `internal`, nachdem Sie die folgende Zeile der `AccessControl`-Klasse gelöscht oder auskommentiert haben:

```
trace(myExt.str);// error if str is not public
```

Überschreiben von Variablen nicht gestattet

Mit den Schlüsselwörtern `var` oder `const` deklarierte Eigenschaften sind geerbt, können jedoch nicht überschrieben werden. Das Überschreiben einer Eigenschaft bedeutet, dass die Eigenschaft in einer Unterklasse neu definiert wird. Der einzige Eigenschaftstyp, der überschrieben werden kann, sind get- und set-Accessorfunktionen (also Eigenschaften, die mit dem `function`-Schlüsselwort deklariert werden). Eine Instanzvariable kann nicht überschrieben werden. Sie können jedoch durch Erstellen von get- und set-Methoden für die Instanzvariable und Überschreiben der Methoden eine ähnliche Funktion erreichen.

Überschreiben von Methoden

Überschreiben einer Methode bedeutet, das Verhalten einer geerbten Methode neu zu definieren. Statische Methoden werden nicht geerbt und können nicht überschrieben werden. Andererseits werden die Instanzmethoden von Unterklassen geerbt und können überschrieben werden, solange die beiden folgenden Kriterien erfüllt sind:

- Die Instanzmethode wurde in der Basisklasse nicht mit dem Schlüsselwort `final` deklariert. Wurde das Schlüsselwort `final` mit einer Instanzmethode verwendet, möchte der Programmierer verhindern, dass die Methode von Unterklassen überschrieben wird.
- Die Instanzmethode wurde in der Basisklasse nicht mit dem Zugriffskontrollbezeichner `private` deklariert. Wurde eine Methode in der Basisklasse als `private` deklariert, muss bei der Definition einer Methode gleichen Namens in der Unterklasse das `override`-Schlüsselwort nicht verwendet werden, da die Basisklassenmethode für die Unterklasse nicht sichtbar ist.

  Um eine Instanzmethode zu überschreiben, die diese Kriterien erfüllt, muss die Methodendefinition in der Unterklasse das Schlüsselwort `override` verwenden und der Methode in der übergeordneten Klasse hinsichtlich Folgendem entsprechen:

- Die überschreibende Methode muss über die gleiche Zugriffskontrollebene wie die Basisklassenmethode verfügen. Als „internal“ deklarierte Methoden haben die gleiche Zugriffskontrollebene wie Methoden, die über keinen Zugriffskontrollbezeichner verfügen.
- Die überschreibende Methode muss über die gleiche Anzahl an Parametern wie die Basisklassenmethode verfügen.
- Die Parameter der überschreibenden Methode müssen die gleichen Datentypanmerkungen wie die Parameter der Basisklassenmethode aufweisen.
- Die überschreibende Methode muss über den gleichen Rückgabetyp wie die Basisklassenmethode verfügen.

Die Namen der Parameter in der überschreibenden Methode müssen jedoch nicht mit den Namen der Parameter in der Basisklasse übereinstimmen, solange die Anzahl der Parameter und der Datentyp jedes Parameters übereinstimmen.

super-Anweisung

Häufig möchten Programmierer beim Überschreiben einer Methode das Verhalten der überschriebenen Methode der übergeordneten Klasse nur ändern und nicht vollständig ersetzen. Dies erfordert einen Mechanismus, mit dem es einer Methode in einer Unterklasse möglich ist, die Version gleichen Namens in der übergeordneten Klasse aufzurufen. Einen solchen Mechanismus bietet die Anweisung `super`, da sie einen Verweis auf die unmittelbar übergeordnete Klasse enthält. Im folgenden Beispielcode wird eine Klasse namens „Base“ definiert, die eine Methode mit der Bezeichnung `thanks()` sowie eine Unterklasse der Base-Klasse namens „Extender“ enthält, welche die `thanks()`-Methode überschreibt. Die `Extender.thanks()`-Methode verwendet die `super`-Anweisung zum Aufrufen von `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Überschreiben von get- und set-Methoden

Im Gegensatz zu Variablen, die in einer übergeordneten Klasse definiert wurden, können get- und set-Methoden überschrieben werden. Im folgenden Beispielcode wird eine get-Methode namens `currentLabel` überschrieben, die in der MovieClip-Klasse von ActionScript 3.0 definiert ist.:

```
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;
        }
    }
}
```

Die Ausgabe der `trace()`-Anweisung im OverrideExample-Klassenkonstruktor lautet `Override: null`; dies zeigt, dass dieses Beispiel in der Lage ist, die geerbte Eigenschaft `currentLabel` zu überschreiben.

Nicht geerbte statische Eigenschaften

Statische Eigenschaften werden von Unterklassen nicht geerbt. Dies bedeutet, dass über die Instanz einer Unterklasse nicht auf statische Eigenschaften zugegriffen werden kann. Der Zugriff auf eine statische Eigenschaft ist nur über das Klassenobjekt möglich, in dem sie definiert wurde. Im folgenden Beispielcode werden eine Basisklasse namens „Base“ sowie eine Unterklasse mit der Bezeichnung „Extender“ definiert, welche die Base-Klasse erweitert. In der Base-Klasse ist eine statische Variable namens `test` definiert. Der Code im folgenden Auszug kann nicht im strikten Modus kompiliert werden und erzeugt im Standardmodus einen Laufzeitfehler.

```
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 { }
```

Die einzige Möglichkeit, auf die statische Variable `test` zuzugreifen, ist über das Klassenobjekt. Dies wird im folgenden Code gezeigt:

```
Base.test;
```

Es ist jedoch auch zulässig, eine Instanzeigenschaft mit dem gleichen Namen wie eine statische Eigenschaft zu definieren. Eine solche Instanzeigenschaft kann in der gleichen Klasse wie die statische Eigenschaft oder in einer Unterklasse definiert werden. So kann die Base-Klasse aus dem vorangegangenen Beispiel eine Instanzeigenschaft namens `test` aufweisen. Der folgende Beispielcode wird kompiliert und ausgeführt, da die Instanzeigenschaft von der Extender-Klasse geerbt wird. Der Code ließe sich auch dann kompilieren und ausführen, wenn die Definition der Instanzvariablen „test“ in die Extender-Klasse verschoben wird, jedoch nicht, wenn sie dorthin kopiert wird.

```
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 {}
```

Statische Eigenschaften und die Gültigkeitsbereichskette

Obwohl statische Eigenschaften nicht geerbt werden, befinden sie sich in der Gültigkeitsbereichskette der Klasse, in der sie definiert wurden, sowie in allen Unterklassen dieser Klasse. Anders ausgedrückt, statische Eigenschaften befinden sich *im Gültigkeitsbereich* der Klasse, in der sie definiert wurden, sowie in allen Unterklassen. Dies bedeutet, dass auf eine statische Eigenschaft direkt aus dem Klassenrumpf zugegriffen werden kann, in dem sie definiert wird, sowie aus allen Unterklassen dieser Klasse.

Im folgenden Beispielcode werden die im vorangegangenen Beispiel definierten Klassen modifiziert, um zu zeigen, dass sich die in der Base-Klasse definierte statische Variable `test` auch im Gültigkeitsbereich der Extender-Klasse befindet. Anders ausgedrückt, die Extender-Klasse kann auf die Variable `test` zugreifen, ohne ihr den Namen der Klasse voranzustellen, in der `test` definiert ist.

```
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
    }

}
```

Wenn eine Instanzeigenschaft mit dem gleichen Namen wie eine statische Eigenschaft in der gleichen oder einer übergeordneten Klasse definiert ist, so hat die Instanzeigenschaft eine höhere Rangstufe in der Gültigkeitsbereichskette. Man kann sagen, die Instanzeigenschaft *verbirgt* die statische Eigenschaft, daher wird der Wert der Instanzeigenschaft anstelle des Werts der statischen Eigenschaft verwendet. Im folgenden Code ist dargestellt, dass bei einer in der Extender-Klasse definierten Instanzvariablen namens `test` die `trace()`-Anweisung anstelle des Wertes der statischen Variablen den Wert der Instanzvariablen verwendet:

```
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
    }

}
```

Erweiterte Themen

Geschichte der OOP-Unterstützung durch ActionScript

Da ActionScript 3.0 auf früheren Versionen von ActionScript aufbaut, sind Kenntnisse, wie sich das ActionScript-Objektmodell entwickelt hat, durchaus von Nutzen. ActionScript begann als einfacher Mechanismus zur Skripterstellung für frühe Versionen von Flash Professional. Später begannen Programmierer, mit ActionScript zunehmend komplexere Anwendungen zu erstellen. Um den Anforderungen dieser Programmierer zu entsprechen, wurden jeder neuen Version weitere Sprachfunktionen hinzugefügt, welche die Erstellung komplexer Anwendungen vereinfachten.

ActionScript 1.0

ActionScript 1.0 ist die Version der Sprache, die in Flash Player 6 und früheren Versionen verwendet wurde. Bereits in diesem frühen Entwicklungsstadium basierte das ActionScript-Objektmodell auf dem Konzept eines Objekts als grundlegendem Datentyp. Ein ActionScript-Objekt ist ein zusammengesetzter Datentyp mit einer Reihe von *Eigenschaften*. Im Sinne eines Objektmodells umfasst der Begriff *Eigenschaften* alle Elemente, die an ein Objekt angefügt werden können, beispielsweise Variable, Funktionen oder Methoden.

Obwohl diese erste Generation von ActionScript das Definieren von Klassen mit dem Schlüsselwort `class` noch nicht unterstützte, konnten Sie eine Klasse mit einem speziellen Objekttyp namens „Prototypobjekt“ definieren. Anstatt das Schlüsselwort `class` zum Erstellen einer abstrakten Klassendefinition zu verwenden, die Sie in konkrete Objekte instanziieren (wie in klassenbasierten Sprachen wie Java und C++), verwenden Sie in prototypbasierten Sprachen wie ActionScript 1.0 ein existierendes Objekt als Modell (oder Prototyp) für andere Objekte. Während Objekte in einer klassenbasierten Sprache auf eine Klasse verweisen können, die als Vorlage dient, verweisen Objekte in einer prototypbasierten Sprache stattdessen auf ein anderes Objekt (ihren Prototyp), der als Vorlage dient.

Zum Erstellen einer Klasse in ActionScript 1.0 definieren Sie eine Konstruktorfunktion für diese Klasse. In ActionScript sind Funktionen tatsächliche Objekte, nicht nur abstrakte Definitionen. Die von Ihnen erstellte Konstruktorfunktion dient als prototypisches Objekt für Instanzen dieser Klasse. Im folgenden Codebeispiel werden eine Klasse namens „Shape“ erstellt und eine Eigenschaft mit der Bezeichnung `visible` definiert, die standardmäßig auf `true` gesetzt ist:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Diese Konstruktorfunktion definiert eine Shape-Klasse, die Sie wie folgt mit dem `new`-Operator instanziieren können:

```
myShape = new Shape();
```

Das `Shape()`-Konstruktorfunktionsobjekt kann nicht nur als Prototyp für Instanzen der Shape-Klasse dienen, sondern auch als Prototyp für Unterklassen der Shape-Klasse verwendet werden, d. h. für Klassen, die die Shape-Klasse erweitern.

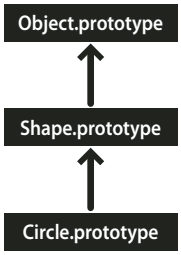
Das Erstellen einer Klasse, bei der es sich um eine Unterklasse der Shape-Klasse handelt, umfasst zwei Schritte. Als Erstes erstellen Sie die Klasse, indem Sie eine Konstruktorfunktion für die Klasse definieren. Dies wird im folgenden Code gezeigt:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Dann verwenden Sie den `new`-Operator, um die Shape-Klasse zu deklarieren, die der Prototyp für die Circle-Klasse ist. In der Standardeinstellung verwendet jede von Ihnen erstellte Klasse die Object-Klasse als Prototyp. Das bedeutet, dass `Circle.prototype` derzeit ein generisches Objekt enthält (eine Instanz der Object-Klasse). Um festzulegen, dass der Circle-Prototyp jetzt „Shape“ anstelle von „Object“ sein soll, verwenden Sie den folgenden Code. Darin wird der Wert von `Circle.prototype` so geändert, dass er ein Shape-Objekt anstelle eines generischen Objekts enthält:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

Die Shape-Klasse und die Circle-Klasse sind jetzt in einer Vererbungsbeziehung miteinander verknüpft, die allgemein als *Prototypkette* bezeichnet wird. Im folgenden Diagramm werden die Beziehungen in einer Prototypkette verdeutlicht:



Die Basisklasse am Ende jeder Prototypkette ist die Object-Klasse. Die Object-Klasse enthält eine statische Eigenschaft namens `Object.prototype`, die auf das Basis-Prototypobjekt für alle in ActionScript 1.0 erstellten Objekte zeigt. Das nächste Objekt in der Beispiel-Prototypkette ist das Shape-Objekt. Der Grund dafür ist, dass die `Shape.prototype`-Eigenschaft nie explizit eingestellt wurde, daher enthält es noch immer ein generisches Objekt (eine Instanz der Object-Klasse). Das abschließende Glied in dieser Kette ist die Circle-Klasse, die mit ihrem Prototyp verknüpft ist (die `Circle.prototype`-Eigenschaft enthält ein Shape-Objekt).

Wenn Sie wie im folgenden Beispielcode eine Instanz der Circle-Klasse erstellen, so erbt die Instanz die Prototypkette der Circle-Klasse:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Das Beispiel enthielt eine Eigenschaft namens `visible` als Mitglied der Shape-Klasse. In diesem Beispiel existiert die `visible`-Eigenschaft nicht als Teil des `myCircle`-Objekts, sondern als Mitglied des Shape-Objekts. Die folgende Codezeile gibt `true` zurück:

```
trace(myCircle.visible); // output: true
```

Die Laufzeit kann sicherstellen, dass das `myCircle`-Objekt die `visible`-Eigenschaft erbt, indem es die Prototypkette aufwärts durchläuft. Bei Ausführung dieses Codes sucht die Laufzeit zunächst in den Eigenschaften des `myCircle`-Objekts nach einer Eigenschaft namens `visible`, findet sie jedoch nicht. Als Nächstes sucht sie im `Circle.prototype`-Objekt, findet jedoch noch immer keine Eigenschaft namens `visible`. Auf dem weiteren Weg durch die Prototypkette findet die Laufzeit schließlich die im `Shape.prototype`-Objekt definierte Eigenschaft `visible` und gibt den Wert dieser Eigenschaft zurück.

Der Einfachheit halber werden viele Details und Feinheiten der Prototypkette weggelassen. Stattdessen sollen ausreichend Informationen zur Verfügung gestellt werden, um ein Verständnis des Objektmodells von ActionScript 3.0 zu ermöglichen.

ActionScript 2.0

Mit ActionScript 2.0 wurden neue Schlüsselwörter wie `class`, `extends`, `public` und `private` eingeführt, mit denen Klassen so definiert werden konnten, wie es Programmierer mit Kenntnissen von klassenbasierten Sprachen wie Java und C++ gewohnt sind. Es ist wichtig zu verstehen, dass sich der zugrunde liegende Vererbungsmechanismus zwischen ActionScript 1.0 und ActionScript 2.0 nicht geändert hat. In ActionScript 2.0 wurde lediglich eine neue Syntax für die Definition von Klassen hinzugefügt. Die Prototypkette ist in beiden Versionen der Programmiersprache identisch.

Mit der in ActionScript 2.0 neu eingeführten Syntax, die in dem folgenden Codeauszug dargestellt wird, können Klassen intuitiver definiert werden, wie viele Programmierer bestätigen:

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

Mit ActionScript 2.0 wurden auch Typanmerkungen für die Typüberprüfung während der Kompilierung eingeführt. Mit diesen Typanmerkungen können Sie deklarieren, dass die Eigenschaft `visible` aus dem vorangegangenen Beispiel nur einen booleschen Wert enthalten darf. Das neue Schlüsselwort `extends` vereinfacht darüber hinaus das Erstellen einer Unterklasse. Im folgenden Beispielcode wird ein Prozess, für den in ActionScript 1.0 zwei Schritte erforderlich waren, mithilfe des Schlüsselworts `extends` in nur einem Schritt abgeschlossen:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

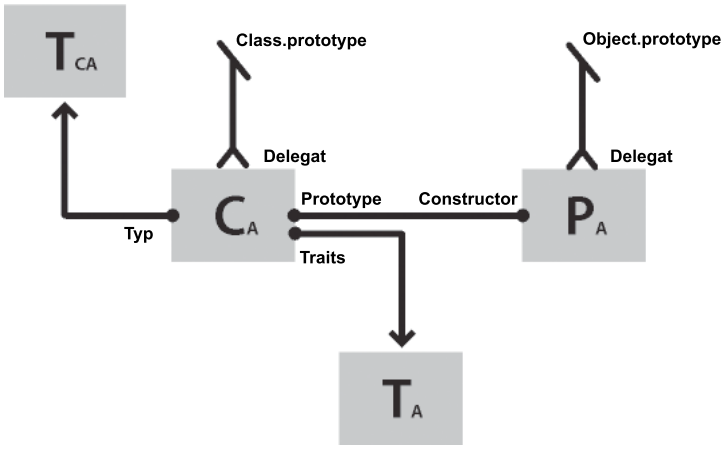
Der Konstruktor wird nun als Teil der Klassendefinition deklariert und die Klasseneigenschaften `id` und `radius` müssen ebenfalls explizit deklariert werden.

Darüber hinaus wurde in ActionScript 2.0 Unterstützung für die Definition von Schnittstellen eingefügt. Somit können Sie objektorientierte Programme mit formal definierten Protokollen für die Kommunikation von Objekten untereinander aufwerten.

ActionScript 3.0-Klassenobjekt

Ein allgemeines Schema bei der objektorientierten Programmierung, das im Wesentlichen mit Java und C++ in Verbindung gebracht wird, verwendet Klassen zur Definition der Objekttypen. Programmiersprachen, die dieses Schema angenommen haben, neigen dazu, Klassen zum Konstruieren von Instanzen des von der Klasse definierten Datentyps zu verwenden. ActionScript verwendet für beides Klassen, der Ursprung als eine prototypbasierte Sprache fügt jedoch eine weitere interessante Eigenschaft hinzu. ActionScript erstellt für jede Klassendefinition ein spezielles Klassenobjekt, dass die gemeinsame Nutzung von Verhalten und Zustand ermöglicht. Für viele ActionScript-Programmierer hat diese Unterscheidung keine praktischen Auswirkungen. ActionScript 3.0 ist so konzipiert, dass Sie moderne objektorientierte ActionScript-Anwendungen erstellen können, ohne dass Sie diese speziellen Klassenobjekte verwenden oder gar verstehen müssen.

Das folgende Diagramm zeigt die Struktur eines Klassenobjekts, das eine einfache Klasse namens „A“ darstellt, die mit der Anweisung `class A {}` definiert wurde:



Jedes Rechteck im Diagramm stellt ein Objekt dar. Jedes Objekt im Diagramm hat einen tiefgestellten Buchstaben A, der kennzeichnet, dass es zur Klasse „A“ gehört. Das Klassenobjekt (CA) enthält Verweise auf verschiedene andere wichtige Objekte. Ein Instanz-Traitsobjekt (TA) speichert die Instanzeigenschaften, die in einer Klassendefinition definiert sind. Ein Klassen-Traitsobjekt (TCA) stellt den internen Typ der Klasse dar und speichert die statischen Eigenschaften, die von der Klasse definiert wurden (das tiefgestellte Zeichen C steht für „Class“ [Klasse]). Das Prototypobjekt (PA) verweist immer auf das Klassenobjekt, an das es ursprünglich über die `constructor`-Eigenschaft angefügt wurde.

Traitsobjekt

Das in ActionScript 3.0 neu eingeführte Traitsobjekt wurde zur Performanceverbesserung implementiert. In früheren Versionen von ActionScript konnte das Nachschlagen eines Namens ein sehr zeitaufwändiger Prozess sein, da Flash Player die Prototypkette aufwärts durchlief. In ActionScript 3.0 wird die Suche eines Namens sehr viel effizienter und weniger zeitaufwändig durchgeführt, da geerbte Eigenschaften aus den übergeordneten Klassen in die Traitsobjekte der Unterklassen kopiert werden.

Ein direkter Zugriff auf das Traitsobjekt über den Programmcode ist nicht möglich, aber das Vorhandensein ist aufgrund der Performanceverbesserungen und besseren Speichernutzung deutlich spürbar. Das Traitsobjekt stellt der AVM2 ausführliche Informationen zu Layout und Inhalt einer Klasse zur Verfügung. Mit diesen Daten ist die AVM2 in der Lage, die Ausführungszeit deutlich zu reduzieren, da sie häufig direkte Maschinenanweisungen erzeugen kann, um direkt ohne zeitaufwändiges Suchen von Namen auf Eigenschaften oder Aufrufmethoden zuzugreifen.

Dank des Traitsobjekts belegt ein Objekt deutlich weniger Speicherplatz als ein ähnliches Objekt in früheren Versionen von ActionScript. Ist eine Klasse versiegelt (also nicht dynamisch deklariert), benötigt eine Klasseninstanz keine Hashtabelle für dynamisch hinzugefügte Eigenschaften und enthält nur wenig mehr als einen Zeiger auf die Traitsobjekte und einige Slots für feste Eigenschaften, die in der Klasse definiert sind. Daher belegt ein Objekt, das in ActionScript 2.0 noch 100 Byte im Arbeitsspeicher belegte, in ActionScript 3.0 nur noch 20 Byte.

Hinweis: *Das Traitsobjekt ist ein internes Implementationsdetail. Es besteht keine Garantie, dass es in zukünftigen Versionen von ActionScript unverändert bleibt oder nicht wieder verschwindet.*

Prototypobjekt

Jedes ActionScript-Klassenobjekt verfügt über eine Eigenschaft namens `prototype`, die einen Verweis auf das Prototypobjekt der Klasse darstellt. Das Prototypobjekt ist ein Vermächtnis aus den Ursprüngen von ActionScript als eine prototypbasierte Sprache. Weitere Informationen finden Sie unter Geschichte der OOP-Unterstützung durch ActionScript.

Die `prototype`-Eigenschaft ist schreibgeschützt, d. h. sie kann nicht geändert werden, um auf andere Objekte zu verweisen. Dies unterscheidet sich von der `prototype`-Eigenschaft einer Klasse in früheren Versionen von ActionScript. Hier konnte der Prototyp neu zugewiesen werden, sodass er auf eine andere Klasse verwies. Auch wenn die Eigenschaft `prototype` schreibgeschützt ist, das Prototypobjekt, auf das sie verweist, ist es nicht. Anders ausgedrückt, einem Prototypobjekt können neue Eigenschaften hinzugefügt werden. Einem Prototypobjekt hinzugefügte Eigenschaften stehen allen Instanzen der Klasse zur Verfügung.

Die Prototypkette, die in früheren Versionen von ActionScript den einzigen Vererbungsmechanismus darstellte, spielt in ActionScript 3.0 nur noch eine sekundäre Rolle. Hier ist der primäre Vererbungsmechanismus die Vererbung fester Eigenschaften, die intern vom Traitsobjekt verarbeitet wird. Eine feste Eigenschaft ist eine Variable oder eine Methode, die als Teil einer Klassendefinition definiert ist. Die Vererbung von festen Eigenschaften wird auch als Klassenvererbung bezeichnet, da sie der einzige Vererbungsmechanismus ist, der mit Schlüsselwörtern wie `class`, `extends` und `override` verbunden ist.

Die Vererbungskette bietet einen alternativen Vererbungsmechanismus, der dynamischer als die Vererbung fester Eigenschaften ist. Sie können dem Prototypobjekt einer Klasse Eigenschaften nicht nur als Teil der Klassendefinition hinzufügen, sondern über die Eigenschaft `prototype` des Klassenobjekts auch zur Laufzeit. Beachten Sie jedoch Folgendes: wenn Sie den Compiler auf den strikten Modus einstellen, können Sie nicht auf die dem Prototypobjekt hinzugefügten Eigenschaften zugreifen, es sei denn, Sie deklarieren eine Klasse mit dem Schlüsselwort `dynamic`.

Ein gutes Beispiel für eine Klasse, bei der mehrere Eigenschaften an das Prototypobjekt angehängt wurden, ist die Object-Klasse. Bei den Methoden `toString()` und `valueOf()` der Objektklasse handelt es sich tatsächlich um Funktionen, die den Eigenschaften des Prototypobjekts der Object-Klasse zugeordnet sind. Der folgende Code zeigt, wie die Deklaration dieser Methoden theoretisch aussehen könnte (die tatsächliche Implementation unterscheidet sich aufgrund der Implementationsdetails ein wenig):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Wie bereits erwähnt, können Sie eine Eigenschaft an das Prototypobjekt einer Klasse außerhalb der Klassendefinition anhängen. Beispielsweise kann die Methode `toString()` auch außerhalb der Object-Klassendefinition definiert werden. Dies wird im folgenden Beispiel gezeigt:

```
Object.prototype.toString = function()
{
    // statements
};
```

Im Gegensatz zur Vererbung von festen Eigenschaften ist für die Prototypvererbung kein Schlüsselwort `override` erforderlich, wenn Sie eine Methode in einer Unterklasse neu definieren möchten. Wenn Sie die Methode `valueOf()` in einer Unterklasse der Object-Klasse neu definieren möchten, haben Sie drei Optionen: Zunächst können Sie eine `valueOf()`-Methode für das Prototypobjekt der Unterklasse in der Klassendefinition definieren. Im folgenden Beispielcode wird zunächst ein Objekt namens „Foo“ erstellt und dann die `valueOf()`-Methode des Prototypobjekts von „Foo“ als Teil der Klassendefinition neu definiert. Da jede Klasse von der Object-Klasse erbt, muss das Schlüsselwort `extends` nicht verwendet werden.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Dann können Sie eine `valueOf()`-Methode für das Prototypobjekt von „Foo“ außerhalb der Klassendefinition definieren. Dies wird im folgenden Beispiel gezeigt:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Als dritte Möglichkeit können Sie eine feste Eigenschaft namens `valueOf()` als Teil der Foo-Klasse definieren. Diese Technik unterscheidet sich insofern von den anderen, als dass sie die Vererbung fester Eigenschaften mit der Prototypvererbung mischt. Jede Unterklasse von „Foo“, die `valueOf()` neu definieren möchte, muss das Schlüsselwort `override` verwenden. Im folgenden Beispielcode wird gezeigt, wie `valueOf()` als eine feste Eigenschaft in „Foo“ definiert wird:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

AS3-Namespace

Die beiden separaten Vererbungsmechanismen, die Vererbung fester Eigenschaften und die Prototypvererbung, stellen hinsichtlich der Eigenschaften und Methoden der Hauptklassen eine interessante Herausforderung an die Kompatibilität. Die Kompatibilität mit der ECMAScript-Sprachspezifikation, auf der ActionScript aufbaut, erfordert die Verwendung der Prototypvererbung. Dies bedeutet, dass die Eigenschaften und Methoden einer Hauptklasse für das Prototypobjekt dieser Klasse definiert werden. Andererseits verlangt die Kompatibilität mit ActionScript 3.0 nach einer Vererbung fester Eigenschaften. Dies bedeutet, dass die Eigenschaften und Methoden einer Hauptklasse mithilfe der Schlüsselwörter `const`, `var` und `function` in der Klassendefinition definiert sind. Darüber hinaus kann die Verwendung der festen Eigenschaften anstelle der Prototypversionen eine deutliche Leistungsverbesserung zur Laufzeit bedeuten.

ActionScript 3.0 löst dieses Problem, indem für die Hauptklassen sowohl die Prototypvererbung als auch die Vererbung fester Eigenschaften verwendet wird. Jede Hauptklasse enthält zwei Sätze mit Eigenschaften und Methoden. Ein Satz wird zur Kompatibilität mit der ECMAScript-Spezifikation am Prototypobjekt definiert, der andere Satz wird zur Kompatibilität mit ActionScript 3.0 mit festen Eigenschaften und dem AS3-Namespace definiert.

Der AS3-Namespace bietet einen bequemen Mechanismus zur Auswahl zwischen den beiden Eigenschaften- und Methodensätzen. Wenn Sie den AS3-Namespace nicht verwenden, erbt eine Instanz der Hauptklasse die Eigenschaften und Methoden, die im Prototypobjekt der Hauptklasse definiert wurden. Wenn Sie den AS3-Namespace verwenden, erbt eine Instanz der Hauptklasse die AS3-Versionen, da die festen Eigenschaften gegenüber den Prototypeigenschaften immer bevorzugt werden. Anders ausgedrückt, wenn eine feste Eigenschaft verfügbar ist, sollte sie stets anstelle einer identisch benannten Prototypeigenschaft verwendet werden.

Sie können selektiv die AS3-Namespace-Version einer Eigenschaft oder Methode verwenden, indem Sie sie mit dem AS3-Namespace qualifizieren. Im folgenden Beispielcode wird die AS3-Version der `Array.pop()`-Methode verwendet:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Alternativ können Sie die Direktive `use namespace` verwenden, um den AS3-Namespace für alle Definitionen innerhalb eines Codeblocks zu öffnen. Im folgenden Beispielcode wird die Direktive `use namespace` verwendet, um den AS3-Namespace für die Methoden `pop()` und `push()` zu öffnen:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

Darüber hinaus bietet ActionScript 3.0 Compileroptionen für jeden Eigenschaftensatz, sodass Sie den AS3-Namespace auf das gesamte Programm anwenden können. Die Compileroption `-as3` stellt den AS3-Namespace, die Compileroption `-es` die Prototypvererbungsoption dar (`es` steht für ECMAScript). Um den AS3-Namespace für das gesamte Programm zu öffnen, setzen Sie die Compileroption `-as3` auf `true` und die Compileroption `-es` auf `false`. Wenn Sie die Prototypversionen verwenden möchten, setzen Sie die Compileroptionen auf die jeweils entgegengesetzten Werte. Die Compiler-Standardeinstellungen für Flash Builder und Flash Professional lauten `-as3 = true` und `-es = false`.

Wenn Sie eine der Hauptklassen erweitern und Methoden überschreiben möchten, müssen Sie verstanden haben, wie der AS3-Namespace Ihr Vorgehen beim Deklarieren einer überschriebenen Methode beeinflussen kann. Wenn Sie den AS3-Namespace verwenden, muss eine Methodenüberschreibung einer Hauptklassenmethode auch den AS3-Namespace zusammen mit dem Attribut `override` verwenden. Wenn Sie den AS3-Namespace nicht verwenden und eine Hauptklassenmethode in einer Unterklasse neu definieren möchten, dürfen Sie den AS3-Namespace oder das Schlüsselwort `override` nicht verwenden.

Beispiel: GeometricShapes

Die Beispielanwendung „GeometricShapes“ verdeutlicht, wie verschiedene objektorientierte Konzepte und Funktionen mit ActionScript 3.0 angewendet werden können:

- Definieren von Klassen
- Erweitern von Klassen
- Polymorphismus und das Schlüsselwort `override`
- Definieren, Erweitern und Implementieren von Schnittstellen

Die Beispielanwendung enthält darüber hinaus eine „Factory-Methode“, mit der gezeigt wird, wie ein Rückgabewert als Instanz einer Schnittstelle deklariert und das zurückgegebene Objekt generisch verwendet wird.

Die Anwendungsdateien für dieses Beispiel finden Sie unter www.adobe.com/go/learn_programmingAS3samples_flash_de. Die Dateien der Anwendung „GeometricShapes“ befinden sich im Ordner „Samples/GeometricShapes“. Die Anwendung umfasst die folgenden Dateien:

Datei	Beschreibung
GeometricShapes.mxml oder GeometricShapes.fla	Die Hauptanwendungsdatei im Flash-Format (FLA) oder Flex-Format (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	Die Methoden, mit denen die grundlegenden Schnittstelle definiert wird, die von allen GeometricShapes-Anwendungsklassen implementiert werden müssen.
com/example/programmingas3/geometricshapes/IPolygon.as	Die Methoden, mit denen eine Schnittstelle definiert wird, die von allen GeometricShapes-Anwendungsklassen mit mehreren Seiten implementiert werden müssen.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Eine geometrische Form, deren gleich lange Seiten symmetrisch um den Mittelpunkt der Form angeordnet sind.
com/example/programmingas3/geometricshapes/Circle.as	Eine geometrische Form, die einen Kreis definiert.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Eine Unterklasse von RegularPolygon, die ein Dreieck definiert, dessen Seiten die gleiche Länge aufweisen.
com/example/programmingas3/geometricshapes/Square.as	Eine Unterklasse von RegularPolygon, die ein Rechteck definiert, dessen vier Seiten die gleiche Länge aufweisen.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Eine Klasse, die eine Factory-Methode zum Erstellen von Formen eines bestimmten Typs und einer bestimmten Größe enthält.

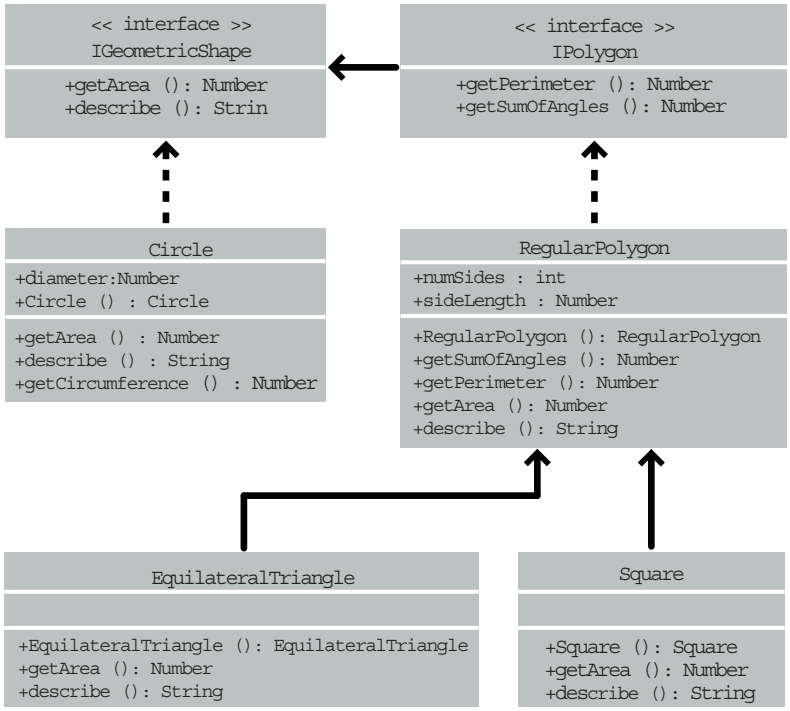
Definieren der GeometricShapes-Klassen

Mit der Anwendung „GeometricShapes“ kann ein Benutzer Typ und Größe einer geometrischen Form angeben. Die Anwendung reagiert dann mit einer Beschreibung der Form, ihrer Fläche und ihres Umfangs.

Die Benutzerschnittstelle der Anwendung ist einfach aufgebaut und enthält nur wenige Steuerelemente zur Auswahl von Typ und Größe der Form sowie zum Anzeigen der Beschreibung. Der interessanteste Teil dieser Anwendung liegt unter der Oberfläche, in der Struktur der Klassen und Schnittstellen selbst.

Die Anwendung arbeitet zwar mit geometrischen Formen, zeigt sie aber nicht grafisch an.

Im folgenden Diagramm sind die Klassen und Schnittstellen, mit denen die geometrischen Formen in diesem Beispiel definiert werden, in der Unified Modeling Language-Notation (UML) dargestellt:



GeometricShapes-Beispielklassen

Definieren des allgemeinen Verhaltens mit Schnittstellen

In dieser GeometricShapes-Anwendung werden drei Arten von Formen verwendet: Kreise, Quadrate und gleichseitige Dreiecke. Die GeometricShapes-Klassenstruktur beginnt mit einer sehr einfachen Schnittstelle, IGeometricShape, in der die Methoden aufgeführt sind, die für alle drei Formen gelten:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

Die Schnittstelle definiert zwei Methoden: die Methode `getArea()`, mit der die Fläche der Form berechnet und zurückgegeben wird, sowie die Methode `describe()`, die eine Textbeschreibung der Eigenschaften der Form zusammensetzt.

Außerdem soll der Umfang jeder Form ermittelt werden. Jedoch wird der Umfang eines Kreises als Kreisumfang bezeichnet und mit einem besonderen Verfahren berechnet. Daher weicht das Verhalten von dem für ein Dreieck oder ein Quadrat ab. Dennoch gibt es ausreichend Gemeinsamkeiten zwischen Dreiecken, Quadraten und anderen Polygonen, dass es sinnvoll ist, eine neue Schnittstellenklasse für sie zu definieren: IPolygon. Die IPolygon-Schnittstelle ist ebenfalls recht einfach:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Diese Schnittstelle definiert zwei Methoden, die alle Polygone gemeinsam haben: die `getPerimeter()`-Methode, die den kombinierten Abstand aller Seiten misst, und die `getSumOfAngles()`-Methode, die alle Innenwinkel summiert.

Die IPolygon-Schnittstelle erweitert die IGeometricShape-Schnittstelle. Dies bedeutet, dass alle Klassen, welche die IPolygon-Schnittstelle implementieren, alle vier Methoden deklarieren müssen – die zwei aus der IGeometricShape-Schnittstelle und die zwei aus der IPolygon-Schnittstelle.

Definieren der Formklassen

Jetzt, nachdem Sie eine Vorstellung der Methoden haben, die für jeden Formtyp gleich sind, können Sie die Formklassen selbst definieren. Hinsichtlich der Anzahl der zu implementierenden Methoden ist die einfachste Form die im Folgenden aufgeführte Circle-Klasse:

```
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;
        }
    }
}
```

Die Circle-Klasse implementiert die IGeometricShape-Schnittstelle, also müssen Sie einen Code für die Methoden `getArea()` und `describe()` bereitstellen. Darüber hinaus definiert sie die `getCircumference()`-Methode, die nur für die Circle-Klasse gilt. Die Circle-Klasse deklariert ebenfalls eine Eigenschaft, `diameter`, die in den anderen Polygonklassen nicht vorhanden sein wird.

Die anderen beiden Formtypen, Quadrate und gleichseitige Dreiecke, haben einige andere Gemeinsamkeiten: Bei beiden Typen sind die Seiten gleich lang und für beide können die gleichen Formeln zum Berechnen des Umfangs und der Summe der Innenwinkel verwendet werden. Tatsächlich gelten diese gemeinsamen Formeln auch für alle anderen regelmäßigen Polygone, die Sie noch definieren werden.

Die RegularPolygon-Klasse ist die übergeordnete Klasse der Square-Klasse und der EquilateralTriangle-Klasse. Mit einer übergeordneten Klasse können Sie allgemeine Methoden an einem Ort definieren, sodass Sie sie nicht in jeder untergeordneten Klasse erneut definieren müssen. Im Folgenden finden Sie den Code für die RegularPolygon-Klasse:

```
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;
        }
    }
}
```

Zuerst deklariert die RegularPolygon-Klasse zwei Eigenschaften, die alle regelmäßigen Polygone gemeinsam haben: die Länge jeder Seite (Eigenschaft `sideLength`) und die Anzahl der Seiten (Eigenschaft `numSides`).

Die RegularPolygon-Klasse implementiert die IPolygon-Schnittstelle und deklariert alle vier IPolygon-Schnittstellenmethoden. Sie implementiert zwei dieser Methoden (`getPerimeter()` und `getSumOfAngles()`) mithilfe gemeinsamer Formeln.

Da die Formel der `getArea()`-Methode je nach Form anders ist, kann die Basisklassenversion der Methode keine gemeinsame Logik enthalten, die von den Methoden der Unterklasse geerbt werden kann. Stattdessen gibt sie einfach eine 0 als Standardwert zurück und kennzeichnet so, dass die Fläche nicht berechnet wurde. Um die Fläche jeder Form korrekt zu berechnen, müssen die Unterklassen der RegularPolygon-Klasse die `getArea()`-Methode selbst übergehen.

Der folgende Code für die EquilateralTriangle-Klasse zeigt, wie die `getArea()`-Methode überschrieben wird:

```
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;
        }
    }
}
```

Das Schlüsselwort `override` gibt an, dass die `EquilateralTriangle.getArea()`-Methode die `getArea()`-Methode der RegularPolygon-Unterklasse absichtlich überschreibt. Wenn die `EquilateralTriangle.getArea()`-Methode aufgerufen wird, berechnet sie die Fläche mithilfe der Formel im vorangegangenen Code, und der Code in der `RegularPolygon.getArea()`-Methode wird nie ausgeführt.

Im Gegensatz dazu definiert die EquilateralTriangle-Klasse keine eigene Version der `getPerimeter()`-Methode. Wenn die `EquilateralTriangle.getPerimeter()`-Methode aufgerufen wird, durchläuft der Aufruf die Vererbungskette aufwärts und führt den Code in der `getPerimeter()`-Methode der übergeordneten RegularPolygon-Klasse aus.

Der `EquilateralTriangle()`-Konstruktor verwendet die `super()`-Anweisung, um den `RegularPolygon()`-Konstruktor der übergeordneten Klasse explizit aufzurufen. Wenn beide Konstruktoren über den gleichen Parametersatz verfügen, können Sie den `EquilateralTriangle()`-Konstruktor weglassen. Stattdessen wird der `RegularPolygon()`-Konstruktor ausgeführt. Der `RegularPolygon()`-Konstruktor benötigt jedoch einen zusätzlichen Parameter, `numSides`. Somit ruft der `EquilateralTriangle()`-Konstruktor `super(len, 3)` auf, der den Eingabeparameter `len` und den Wert 3 übergibt. Auf diese Weise wird angegeben, dass das Dreieck drei Seiten hat.

Die `describe()`-Methode verwendet ebenfalls die `super()`-Anweisung, jedoch auf andere Weise. Sie ruft damit die Version der RegularPolygon-Superklasse der `describe()`-Methode auf. Die `EquilateralTriangle.describe()`-Methode stellt zuerst die String-Variable `desc` auf eine Anweisung zum Typ der Form ein. Dann ruft sie die Ergebnisse der `RegularPolygon.describe()`-Methode ab, indem sie `super.describe()` aufruft, und hängt das Ergebnis an den String `desc` an.

Die Square-Klasse wird hier nicht in allen Einzelheiten besprochen, aber sie ähnelt der EquilateralTriangle-Klasse. Sie stellt einen Konstruktor und ihre eigenen Implementationen der Methoden `getArea()` und `describe()` bereit.

Polymorphismus und die Factory-Methode

Ein Klassensatz, der Schnittstellen und Vererbung einsetzt, kann auf viele interessante Arten verwendet werden. Beispielsweise implementieren alle bisher beschriebenen Formklassen entweder die IGeometricShape-Schnittstelle oder erweitern eine Unterklasse, die diese Schnittstelle implementiert. Wenn Sie also eine Variable als Instanz von IGeometricShape definieren, müssen Sie nicht unbedingt wissen, ob es sich nun um eine Instanz der Circle- oder der Square-Klasse handelt, wenn Sie nur ihre `describe()`-Methode aufrufen möchten.

Der folgende Code zeigt, wie dies funktioniert:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Wenn `myShape.describe()` aufgerufen wird, führt es die Methode `Circle.describe()` aus, denn obwohl die Variable als Instanz der IGeometricShape-Schnittstelle definiert ist, ist „Circle“ die zugrunde liegende Klasse.

Dieses Beispiel veranschaulicht die Funktionsweise von Polymorphismus: Der exakt gleiche Methodenaufruf führt zur Ausführung von unterschiedlichem Code, je nachdem, welcher Klasse das Objekt angehört, dessen Methode aufgerufen wird.

Die Anwendung „GeometricShapes“ wendet diese Art des schnittstellenbasierten Polymorphismus mit einer einfachen Version eines Entwurfsmusters an, das als Factory-Methode bezeichnet wird. Der Begriff *Factory-Methode* bezieht sich auf eine Funktion, die ein Objekt zurückgibt, dessen zugrunde liegender Datentyp oder Inhalt je nach Kontext unterschiedlich sein kann.

Die hier vorgestellte GeometricShapeFactory-Klasse definiert eine Factory-Methode namens `createShape()`:

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

Die Factory-Methode `createShape()` ermöglicht es den Konstruktoren der Form-Unterklasse, die Details der von ihnen erstellten Instanzen zu definieren, während die neuen Objekte als IGeometricShape-Instanzen zurückgegeben werden, sodass sie von der Anwendung allgemeiner verarbeitet werden können.

Die `describeShape()`-Methode aus dem vorangegangenen Beispiel zeigt, wie eine Anwendung die Factory-Methode verwenden kann, um einen generischen Verweis auf ein spezifischeres Objekts zu erhalten. Die Anwendung kann die Beschreibung für das neu erstellte Circle-Objekt wie folgt erhalten:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

Die `describeShape()`-Methode ruft dann die Factory-Methode `createShape()` mit den gleichen Parametern auf und speichert das neue Circle-Objekt in einer statischen Variablen namens `currentShape`, die als ein IGeometricShape-Objekt typisiert wurde. Als Nächstes wird die `describe()`-Methode für das `currentShape`-Objekt aufgerufen. Dieser Aufruf wird automatisch aufgelöst, um die `Circle.describe()`-Methode auszuführen, die eine detaillierte Beschreibung des Kreises zurückgibt.

Ausbauen der Beispielanwendung

Die wahre Leistungsfähigkeit von Schnittstellen und Vererbung wird sichtbar, wenn Sie Ihre Anwendung aufwerten oder ändern.

Angenommen, Sie möchten der Beispielanwendung eine neue Form, ein Fünfeck, hinzufügen. In diesem Fall erstellen Sie eine Pentagon-Klasse, die die RegularPolygon-Klasse erweitert und ihre eigene Version der Methoden `getArea()` und `describe()` definiert. Dann würden Sie dem Kombinationsfeld in der Benutzeroberfläche der Anwendung eine neue Pentagon-Option hinzufügen. Das ist dann aber auch schon alles. Die Pentagon-Klasse würde automatisch die Funktionen der Methoden `getPerimeter()` und `getSumOfAngles()` der RegularPolygon-Klasse erben. Da sie von einer Klasse erbt, die bereit die IGeometricShape-Schnittstelle geerbt hat, kann eine Pentagon-Instanz ebenfalls als eine IGeometricShape-Instanz behandelt werden. Dies bedeutet, dass es zum Hinzufügen eines neuen Formtyps nicht erforderlich ist, die Methodensignatur einer der Methoden in der GeometricShapeFactory-Klasse zu ändern (folglich müssen auch keine Änderungen an Code vorgenommen werden, der die GeometricShapeFactory-Klasse verwendet).

Sie können dem GeometricShapes-Beispiel jetzt zur Übung eine Pentagon-Klasse hinzufügen. Dabei werden Sie feststellen, wie Schnittstellen und Vererbung das Hinzufügen von neuen Funktionen zu einer Anwendung vereinfachen.