ACTIONSCRIPT® 3.0 leren gebruiken

Juridische kennisgeving

Laatst bijgewerkt 13/5/2011

Inhoud

Hoofdstuk 1: Inleiding tot ActionScript 3.0

Informatie over ActionScript

ActionScript is de programmeertaal voor de runtimeomgevingen van Adobe® Flash® Player en Adobe® AIR™. Het maakt interactiviteit, gegevensverwerking en nog veel meer mogelijk in Flash-, Flex- en AIR-inhoud en -toepassingen.

ActionScript wordt uitgevoerd met de ActionScript Virtual Machine, een onderdeel van Flash Player en AIR. ActionScript-code wordt normaal gezien door een compiler getransformeerd naar een bytecodeformaat. (*Bytecode* is een soort programmeringstaal die geschreven en begrepen wordt door computers.) Voorbeelden van compilers zijn degene die ingebouwd zijn in Adobe® Flash® Professional en in Adobe® Flash® Builder™ en beschikbaar is in Adobe® Flex™ SDK. De bytecode is ingesloten in SWF-bestanden, die worden uitgevoerd door Flash Player en AIR.

ActionScript 3.0 biedt een krachtig programmeermodel dat ontwikkelaars met een basiskennis van objectgeoriënteerd programmeren vertrouwd zal lijken. Enkele belangrijke functies van ActionScript 3.0 die een verbetering zijn ten opzichte van oudere versies van ActionScript zijn de volgende:

- Een nieuwe ActionScript virtuele machine, genaamd AVM2, die een nieuwe verzameling bytecode-instructies gebruikt en aanzienlijke prestatieverbeteringen biedt
- Een modernere compilercodebasis die een betere optimalisering uitvoert dan oudere versies van de compiler
- Een uitgebreide en verbeterde programmeerinterface voor toepassingen (API), met besturing van objecten op een laag niveau en een echt objectgeoriënteerd model
- Een XML-API gebaseerd op de specificatie ECMAScript for XML (E4X) (ECMA-357, 2nd Edition). E4X is een taaluitbreiding voor ECMAScript die XML als een native gegevenstype van de taal toevoegt.
- Een gebeurtenismodel, dat gebaseerd is op de Document Object Model (DOM) Level 3 Events Specification.

Voordelen van ActionScript 3.0

ActionScript 3.0 biedt meer dan de scriptmogelijkheden van eerdere versies van ActionScript. Het is ontworpen om het maken van zeer complexe toepassingen met grote gegevenssets en objectgeoriënteerde, herbruikbare code mogelijk te maken. ActionScript 3.0 is niet vereist voor inhoud die in Adobe Flash Player wordt uitgevoerd. Het opent wel de mogelijkheid voor prestatieverbeteringen die alleen beschikbaar zijn met de AVM2 (ActionScript 3.0 Virtual Machine). ActionScript 3.0-code kan tot tien keer sneller zijn dan oudere ActionScript-code.

De vorige versie van ActionScript Virtual Machine, AVM1, voert ActionScript 1.0- en ActionScript 2.0-code uit. Flash Player 9 en 10 ondersteunen AVM1 voor achterwaartse compatibiliteit.

Nieuw in ActionScript 3.0

Hoewel ActionScript 3.0 veel klassen en functies bevat die vertrouwd zijn voor programmeurs van ActionScript 1.0 en 2.0, verschilt ActionScript 3.0 conceptueel en op het gebied van de architectuur van de eerdere versies van ActionScript. De verbeteringen in ActionScript 3.0 bestaan uit nieuwe functies van de kerntaal en een verbeterde API die een betere besturing van objecten op laag niveau biedt.

Functies van de kerntaal

De kerntaal definieert de basisbouwstenen van de programmeertaal, zoals instructies, expressies, voorwaarden, lussen en typen. ActionScript 3.0 bevat veel functies die het ontwikkelingsproces versnellen.

Uitvoeringsuitzonderingen

ActionScript 3.0 meldt meer fouten dan eerdere versies van ActionScript. Uitvoeringsuitzonderingen worden gebruikt voor algemene fouten, wat de foutopsporing verbetert. Hierdoor kunt u toepassingen ontwikkelen die fouten op een krachtige manier afhandelen. Uitvoeringsfouten kunnen stacktraces met aantekeningen van bronbestand en regelnummerinformatie leveren, waarmee u snel fouten kunt vinden.

Uitvoeringstypen

In ActionScript 3.0 wordt type-informatie bij uitvoering opgeslagen. Deze informatie wordt gebruikt voor het controleren van het uitvoeringstype en het verbeteren van de typeveiligheid van het systeem. Informatie over gegevenstypen wordt ook gebruikt om variabelen in native machinerepresentaties te vertegenwoordigen, wat de prestaties verbetert en het geheugengebruik vermindert. In ActionScript 2.0 dienen typeannotaties vooral als hulp voor ontwikkelaars en worden alle waarden tijdens de uitvoering dynamisch ingevoerd.

Verzegelde klassen

ActionScript 3.0 bevat het concept van verzegelde klassen. Een verzegelde klasse bezit alleen de vaste verzameling eigenschappen en methoden die bij compilatie zijn gedefinieerd; u kunt geen aanvullende eigenschappen en methoden toevoegen. Omdat u tijdens de uitvoering een klasse niet meer kunt wijzigen, is een strictere controle van de compilatietijd mogelijk, wat robuustere programma's oplevert. Het verbetert tevens het geheugengebruik, omdat een interne hash-tabel niet voor elke objectinstantie is vereist. Dynamische klassen zijn ook mogelijk door het gebruik van het trefwoord `dynamic`. Alle klassen in ActionScript 3.0 zijn standaard verzegeld, maar kunnen als dynamisch worden gedeclareerd met het trefwoord `dynamic`.

Methode closure

In ActionScript 3.0 kan een methode closure automatisch zijn oorspronkelijke objectinstantie onthouden. Deze functie is nuttig voor het afhandelen van gebeurtenissen. In ActionScript 2.0 onthield een methode closure niet uit welke objectinstantie deze werd geëxtraheerd. Dit leidde tot onverwacht gedrag wanneer de methode closure werd aangeroepen.

ECMAScript for XML (E4X)

ActionScript 3.0 implementeert ECMAScript for XML (E4X), dat onlangs als ECMA-357 is gestandaardiseerd. E4X biedt een natuurlijke, vloeiende verzameling taalconstructies voor het bewerken van XML. In tegenstelling tot traditionele API's die XML parseren, werkt XML met E4X als een native gegevenstype van de taal. E4X stroomlijnt de ontwikkeling van toepassingen die XML bewerken door de hoeveelheid benodigde code drastisch te verminderen.

Ga naar www.ecma-international.org als u de E4X-specificatie van ECMA wilt raadplegen.

Reguliere expressies

ActionScript 3.0 omvat native ondersteuning voor reguliere expressies zodat u tekenreeksen snel kunt zoeken en manipuleren. ActionScript 3.0 implementeert ondersteuning voor reguliere expressies, zoals deze worden gedefinieerd in de ECMAScript Language Specification (ECMA-262), 3rd Edition.

Naamruimten

Naamruimten lijken op de traditionele toegangsspecificaties die worden gebruikt om de zichtbaarheid van declaraties te beheren (`public`, `private`, `protected`). Ze werken als aangepaste toegangsspecificaties, die u namen naar keuze kunt geven. Naamruimten zijn voorzien van een Uniform Resource Identifier (URI) om conflicten te voorkomen. Ze worden tevens gebruikt om XML-naamruimten te vertegenwoordigen wanneer u met E4X werkt.

Nieuwe primitieve typen

ActionScript 3.0 bevat drie numerieke typen: getal, int en uint. Het getal vertegenwoordigt een getal met dubbele precisie en drijvend punt. Het type int is een 32-bits geheel getal met teken waardoor ActionScript-code gebruik kan maken van de snelle wiskundige mogelijkheden voor gehele getallen van de processor. Het type int is nuttig voor lustellers en variabelen waarbij gehele getallen worden gebruikt. Het type uint is een 32-bits geheel getal zonder teken dat nuttig is voor RGB-kleurwaarden, tellingen van bytes en meer. ActionScript 2.0 heeft slechts één numeriek type: Getal.

API-functies

De API's van ActionScript 3.0 bevatten veel klassen waarmee u objecten op een laag niveau kunt beheren. De architectuur van de taal is ontworpen om intuïtiever te werk te gaan dan in oudere versies. Hoewel we niet alle klassen in detail kunnen behandelen, zijn er een aantal belangrijke verschillen.

DOM3-gebeurtenismodel

Het gebeurtenismodel DOM3 (Document Object Model niveau 3) biedt een standaardmanier voor het genereren en beheren van gebeurtenisberichten. Dit gebeurtenismodel is ontworpen voor interactie en communicatie tussen objecten binnen toepassingen, het onderhouden van hun status en reageren op wijzigen. Het gebeurtenismodel ActionScript 3.0 is ontworpen volgens de specificatie voor gebeurtenissen in World Wide Web Consortium DOM niveau 3. Dit model biedt een helderder en efficiënter mechanisme dan de gebeurtenissystemen die beschikbaar zijn in vorige versies van ActionScript.

Gebeurtenissen en foutgebeurtenissen bevinden zich in het pakket flash.events. De Flash Professional-componenten en het Flex-framework gebruiken hetzelfde gebeurtenismodel, zodat het gebeurtenismodel voor het hele Flash Platform hetzelfde is.

Weergaveoverzicht-API

De API voor het verkrijgen van toegang tot het weergaveoverzicht, de boomstructuur die alle visuele elementen in de toepassing bevat, bestaat uit klassen voor het werken met visuele primitieven.

De Sprite-klasse is een lichtgewicht bouwsteen, die is ontworpen als een basisklasse voor visuele elementen zoals gebruikersinterfacecomponenten. De klasse Shape vertegenwoordigt onbewerkte vectorvormen. Deze klassen kunnen op natuurlijke wijze worden geïnstantieerd met de operator `new` en kunnen op elk ogenblik weer dynamisch aan de bovenliggende klasse worden toegevoegd.

Dieptebeheer is automatisch. Methoden worden aangeboden voor het opgeven en beheren van de stapelvolgorde van objecten.

Dynamische gegevens en inhoud verwerken

ActionScript 3.0 bevat mechanismen voor het laden en afhandelen van elementen en gegevens in uw toepassing die intuïtief en consistent zijn in de gehele API. De klasse Loader biedt een enkel mechanisme voor het laden van SWF-bestanden en afbeeldingselementen en biedt een manier om toegang te krijgen tot gedetailleerde informatie over geladen inhoud. De klasse URLLoader biedt een afzonderlijk mechanisme voor het laden van tekst en binaire gegevens in toepassingen die met gegevens werken. De klasse Socket biedt een manier om binaire gegevens te lezen en schrijven naar serversockets in elke willekeurige indeling.

Toegang tot gegevens op een laag niveau

Verschillende API's bieden toegang tot gegevens op laag niveau. Voor gegevens die worden gedownload, biedt de klasse URLStream, toegang tot gegevens als onbewerkte binaire gegevens terwijl ze worden gedownload. Via de klasse ByteArray kunt u lezen, schrijven en werken met binaire gegevens optimaliseren. De geluids-API biedt gedetailleerde besturing van geluid via de klassen SoundChannel en SoundMixer. API's die betrekking hebben op beveiliging, bieden informatie over de beveiligingsrechten van een SWF-bestand of van geladen inhoud. Hierdoor kunt u beveiligingsfouten afhandelen.

Werken met tekst

ActionScript 3.0 bevat een pakket flash.text voor alle tekstgerelateerde API's. De klasse TextLineMetrics bevat gedetailleerde afmetingen voor een regel tekst in een tekstveld; deze vervangt de methode `TextFormat.getTextExtent()` van ActionScript 2.0. De klasse TextField bevat methoden op een laag niveau die specifieke informatie kunnen geven over een regel tekst of over een enkel teken in een tekstveld. De methode `getCharBoundaries()` levert bijvoorbeeld een rechthoek op die het omsluitende kader van een teken vertegenwoordigt. De methode `getCharIndexAtPoint()` levert de index van het teken op een bepaald punt op. De methode `getFirstCharInParagraph()` levert de index van het eerste teken in een alinea op. Methoden op regelniveau bestaan uit `getLineLength()`, dat het aantal tekens in een opgegeven tekstregel retourneert en `getLineText()`, dat de tekst van de opgegeven regel retourneert. De klasse Font biedt een manier om ingesloten lettertypen in SWF-bestanden te beheren.

Voor tekstcontrole op een nog lager niveau, bestaat de Flash Text Engine uit de klassen in het flash.text.engine-pakket. Deze klassenset biedt tekstcontrole op laag niveau en is ontworpen voor het maken van tekstframeworks en componenten.

Hoofdstuk 2: Aan de slag met ActionScript

Programmeerbeginselen

Aangezien ActionScript een programmeertaal is, is het voor het leren van ActionScript nuttig wanneer u eerst kennis opdoet van een aantal algemene programmeerbeginselen.

De werking van een computerprogramma

Allereerst is het goed om te begrijpen wat een computerprogramma is en wat het doet. Er zijn twee hoofdkenmerken van een computerprogramma:

- Een programma is een serie instructies of stappen die de computer moet uitvoeren.
- Elke stap zorgt er uiteindelijk voor dat bepaalde informatie of gegevens worden bewerkt.

In algemene zin is een computerprogramma niets meer dan een lijst met stapsgewijze opdrachten die u geeft aan de computer, die deze vervolgens één voor één uitvoert. Elke individuele opdracht wordt een *instructie* genoemd. In ActionScript wordt elke instructie geschreven met een puntkomma aan het einde.

Een bepaalde opdracht in een programma doet in wezen niets anders dan het manipuleren van een deel van de gegevens die zijn opgeslagen in het computergeheugen. Een eenvoudig voorbeeld van een instructie is twee getallen op te tellen en het resultaat op te slaan het geheugen. Als complexer voorbeeld stelt u zich voor dat een rechthoek op het scherm is getekend en u een programma wilt schrijven om deze naar een andere locatie op het scherm te verplaatsen. De computer houdt bepaalde informatie over de rechthoek bij, zoals de x- en y-coördinaten van de locatie, de breedte en lengte, de kleur enzovoort. Elk van deze gegevens wordt ergens in het computergeheugen opgeslagen. Een programma om de rechthoek naar een andere locatie te verplaatsen bevat bijvoorbeeld stappen als “X-coördinaat wijzigen in 200; y-coördinaat wijzigen in 150." Met andere woorden, zal dit programma dus nieuwe waarden opgeven voor de x- en y-coördinaten. Achter de schermen verandert de computer deze gegevens zodat de cijfers omgezet worden in de afbeelding die op het scherm verschijnt. Op het algemene detailniveau is het voldoende om te weten, dat het proces van "het verplaatsen van een rechthoek op het scherm" inhoudt dat gegevens in het computergeheugen worden gewijzigd.

Variabelen en constanten

Bij programmeren wordt informatie in het computergeheugen gewijzigd. Het is dan ook belangrijk om een manier te hebben om een gedeelte van de informatie in een programma weer te geven. Een *variabele* is een naam die een waarde in het computergeheugen vertegenwoordigt. Wanneer u instructies schrijft om waarden te manipuleren, schrijft u de naam van de variabele in plaats van de waarde zelf. Elke keer dat de computer de naam van de variabele in het programma tegenkomt, wordt de waarde gebruikt die in het geheugen wordt gevonden. Wanneer u bijvoorbeeld twee variabelen `value1` en `value2` hebt die elk een getal bevatten, kunt u de volgende instructie schrijven om beide getallen bij elkaar op te tellen:

```
value1 + value2
```

Wanneer de computer de stappen daadwerkelijk uitvoert, zoekt deze de waarden in beide variabelen en telt deze bij elkaar op.

In ActionScript 3.0 bestaat een variabele feitelijk uit drie gedeelten:

- De naam van de variabele.

- Het type gegevens dat in de variabele kan worden opgeslagen.
- De feitelijke waarde die is opgeslagen in het computergeheugen.

U hebt gezien hoe de computer de naam gebruikt als plaatsaanduiding voor de waarde. Het gegevenstype is ook van belang. Wanneer u in ActionScript een variabele maakt, geeft u het type gegevens op dat hierin wordt opgeslagen. Daarna kunnen de instructies van het programma in de variabele alleen dat type gegevens opslaan. U kunt de waarde aanpassen met de attributen die met dit gegevenstype worden geassocieerd. In ActionScript gebruikt u de instructie `var` om een variabele te maken (ook wel *declareren* genoemd):

```
var value1:Number;
```

Dit voorbeeld zorgt ervoor dat de computer een variabele `value1` maakt, die alleen Number-gegevens kan bevatten. ("Number" is een specifiek gegevenstype dat is gedefinieerd in ActionScript.) U kunt ook direct een waarde in de variabele opslaan:

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

Adobe Flash Professional
In Flash Professional kunt u op een andere manier een variabele declareren. Wanneer u een filmclipsymbool, knopsymbool of tekstveld in het werkgebied plaatst, kunt u dit een instantienaam geven in Eigenschapcontrole. Achter de schermen maakt Flash Professional een variabele met dezelfde naam als de instantienaam. U kunt die naam in uw ActionScript-code gebruiken om dat Werkgebieditem weer te geven. U hebt bijvoorbeeld een filmclipsymbool op het Werkgebied hebt en geeft deze de naam `rocketShip`. Wanneer u de variabele `rocketShip` in uw ActionScript-code gebruikt, wordt deze filmclip bewerkt.

Een *constante* lijkt erg op een variabele. Het is een naam die een waarde vertegenwoordigt in het computergeheugen met een bepaald gegevenstype. Het verschil is dat aan een constante slechts één keer een waarde kan worden toegewezen tijdens het uitvoeren van een ActionScript-toepassing. Wanneer de waarde van een constante is toegewezen, blijft deze gedurende de toepassing hetzelfde. De syntaxis voor het declareren van een constante is bijna dezelfde als voor het declareren van een variabele. het enige verschil is dat u het trefwoord `const` gebruikt in plaats van `var`.

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

Een constante is handig voor het definiëren van waarden die op meerdere plekken in een project worden gebruikt en die onder normale omstandigheden niet gewijzigd worden. Door het gebruik van constanten in plaats van letterlijke waarden wordt de code beter leesbaar. Bekijk bijvoorbeeld eens deze twee versies van dezelfde code. De ene code vermenigvuldigt de prijs met `SALES_TAX_RATE`. De andere code vermenigvuldigd de prijs met `0,07`. De versie met de constante `SALES_TAX_RATE` is eenvoudiger om te begrijpen. Stel dat de waarde die door de constante wordt gedefinieerd wel wijzigt. Als u een constante gebruikt om deze waarde in uw hele project te vertegenwoordigen, hoeft u de waarde slechts op één locatie te wijzigen (de declaratie van de constante). Als u harde letterlijk waarden hebt gebruikt, moet u de waarde op meerdere locaties te wijzigen.

Gegevenstypen

In ActionScript zijn er veel gegevenstypen die u kunt gebruiken voor de variabelen die u maakt. Sommige hiervan kunnen worden beschouwd als 'eenvoudige' of 'fundamentele' gegevenstypen:

- Tekenreeks: een tekstwaarde, zoals een naam of de tekst van een hoofdstuk uit een boek.
- Numeriek: ActionScript 3.0 bevat drie specifieke gegevenstypen voor numerieke gegevens:
  - Number: een willekeurige numerieke waarde, met of zonder breuk.

  - int: een geheel getal (dus zonder breuk).
  - uint: een geheel getal 'zonder teken' (en dat dus niet negatief kan zijn).
- Boolean: een waarde die waar of onwaar is, bijvoorbeeld het feit of een schakelaar aanstaat of het feit of twee waarden gelijk zijn.

De eenvoudige gegevenstypen vertegenwoordigen een enkel gegeven, bijvoorbeeld een enkel getal of een enkele tekenreeks. De meeste gegevenstypen die in ActionScript zijn gedefinieerd, zijn complexe gegevenstypen. Ze vertegenwoordigen een aantal waarden in één container. Een variabele van het gegevenstype Date vertegenwoordigt bijvoorbeeld een enkele waarde, namelijk een tijdstip. Deze datumwaarde wordt echter vertegenwoordigd door de verschillende waarden dag, maand, jaar, uren, minuten, seconden enzovoort, die elk uit een individueel getal bestaan. Mensen denken bij een datum aan een enkele waarde en u kunt een datum als een enkele waarde behandelen door een variabele Date te maken. De computer beschouwt dit echter als een groep van verschillende waarden, die samen één datum definiëren.

De meeste ingebouwde gegevenstypen, alsmede gegevenstypen die worden gedefinieerd door programmeurs, zijn complexe gegevenstypen. Enkele complexe gegevenstypen die u mogelijk kent, zijn:

- MovieClip: een filmclipsymbool;
- TextField: een dynamisch veld of invoertekstveld;
- SimpleButton: een knopsymbool;
- Date: informatie over een enkel tijdstip (een datum en tijd).

Twee termen die vaak worden gebruikt als synoniem voor gegevenstype zijn klasse en object. Een *klasse* is de definitie van een gegevenstype. Het is een sjabloon voor alle objecten van het gegevenstype, bijvoorbeeld als "alle variabelen van het gegevenstype Voorbeeld hebben de volgende eigenschappen: A, B en C." Een *object* is echter slechts een werkelijke instantie van een klasse. Een variabele met als gegevenstype FilmClip, kan worden omschreven als een Filmclipobject. De volgende uitspraken zijn verschillende manieren om hetzelfde te zeggen:

- Het gegevenstype van de variabele `myVariable` is Number.
- De variabele `myVariable` is een Number-instantie.
- De variabele `myVariable` is een Number-object.
- De variabele `myVariable` is een instantie van de klasse Number.

Werken met objecten

ActionScript is een zogenaamde objectgeoriënteerde programmeertaal. Objectgeoriënteerd programmeren is gewoon een manier van programmeren. Het is eigenlijk niets anders dan een manier om de code met behulp van objecten in een programma te organiseren.

De term 'computerprogramma' werd eerder gedefinieerd als een reeks stappen of instructies die een computer uitvoert. In dat geval kan een computerprogramma worden voorgesteld als een enkele, lange lijst van instructies. In objectgeoriënteerde programmering worden de programma-instructies verdeeld over verschillende objecten. De code wordt gegroepeerd in gedeeltes van de functionaliteit, zodat gerelateerde functionaliteitstypen of gerelateerde informatietypen in één container worden gegroepeerd.

Adobe Flash Professional

Als u al eerder in Flash Professional met symbolen hebt gewerkt, bent u al gewend aan het werken met objecten. Stel dat u een filmclipsymbool hebt gedefinieerd, bijvoorbeeld een tekening van een rechthoek, en een kopie ervan in het werkgebied hebt geplaatst. Dit filmclipsymbool is ook (letterlijk) een object in ActionScript. Het is een instantie van de klasse MovieClip.

Er zijn verschillende kenmerken van de filmclip die u kunt wijzigen. Wanneer de filmclip is geselecteerd, kunt u in de Eigenschappencontrole waarden zoals het x-coördinaat of de breedte wijzigen. U kunt ook verschillende kleuraanpassingen uitvoeren, zoals het wijzigen van de alfa (transparantie) of het toepassen van een schaduwfilter. Met andere Flash Professional-gereedschappen kunt u meer wijzigingen doorvoeren. U kunt bijvoorbeeld met het gereedschap Vrije transformatie de rechthoek roteren. Al deze manieren om een filmclipsymbool aan te passen in Flash Professional zijn ook beschikbaar in ActionScript. U kunt de filmclip in ActionScript aanpassen door gedeelten informatie te wijzigen samen worden gebracht in één Filmclipobject.

Bij objectgeoriënteerd programmeren in ActionScript zijn er drie typen kenmerken die elke klasse kan bevatten:

- Eigenschappen
- Methoden
- Gebeurtenissen

Deze elementen dienen om de gegevens te beheren die door het programma worden gebruikt en te bepalen welke handelingen worden uitgevoerd en in welke volgorde.

Eigenschappen

Een eigenschap vertegenwoordigt één van de gegevens die zijn samengebracht in een object. Een Songobject heeft bijvoorbeeld de eigenschappen `artist` en `title`. De klasse MovieClip kan eigenschappen als `rotation`, `x`, `width` en `alpha` hebben. Eigenschappen worden als individuele variabelen behandeld. Een andere manier om eigenschappen te omschrijven is als de 'onderliggende' variabelen in een object.

Hieronder volgen enkele voorbeelden van ActionScript-code waarin gebruik wordt gemaakt van eigenschappen. De volgende coderegel verplaatst de MovieClip met de naam `square` naar 100 pixels op de x-coördinaat:

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

De volgende code gebruikt de eigenschap rotation om de MovieClip `square` te roteren zodat de rotatie gelijk wordt aan die van de MovieClip `triangle`:

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

De volgende code wijzigt de horizontale schaal van de MovieClip `square` zodat deze anderhalf keer breder wordt:

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

Let op de gemeenschappelijke structuur: u gebruikt een variabele (`square`, `triangle`) als naam van het object, gevolgd door een punt (`.`) en vervolgens de naam van de eigenschap (`x`, `rotation`, `scaleX`). De punt, die *puntoperator* wordt genoemd, wordt gebruikt om aan te geven dat u een van de onderliggende elementen van een object benadert. De gehele structuur samen, 'variabelenaam-punt-eigenschapnaam', wordt gebruikt als een enkele variabele, dus als een naam voor een enkele waarde in het computergeheugen.

Methoden

Een *methode* is een actie die door een object kan worden uitgevoerd. U hebt bijvoorbeeld een filmclipsymbool gemaakt in Flash Professional met op de tijdslijn meerdere hoofdframes en animatie. Deze filmclip kan worden afgespeeld of gestopt of kan worden verteld om de afspeelkop naar een bepaald frame te verplaatsen.

De volgende coderegel zorgt ervoor dat de MovieClip met de naam `shortFilm` begint met afspelen:

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

De volgende coderegel zorgt ervoor dat de MovieClip met de naam `shortFilm` stopt met afspelen (de afspeelkop blijft op zijn plaats, zoals bij het pauzeren van een video):

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

De volgende coderegel zorgt ervoor dat de afspeelkop van de MovieClip met de naam `shortFilm` wordt verplaatst naar frame 1 en stopt met afspelen (zoals bij het terugspoelen van een video):

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

Methoden worden, net zoals eigenschappen, benaderd door de naam van het object (een variabele) te schrijven, gevolgd door een punt en de naam van de methode met haakjes erachter. Met de haakjes geeft u aan dat u de methode *aanroept*, met andere woorden, het object de instructie geeft een bepaalde handeling uit te voeren. Soms worden waarden (of variabelen) tussen haakjes geplaatst om aanvullende informatie door te geven die nodig is om de handeling uit te voeren. Deze waarden worden de *parameters* van een methode genoemd. De methode `gotoAndStop()` moet bijvoorbeeld weten naar welk frame moet worden gegaan en heeft dus een enkele parameter tussen de haakjes nodig. Andere methoden, zoals `play()` en `stop()` behoeven geen uitleg en hebben dus geen aanvullende informatie nodig. Deze methoden worden echter wel met haakjes geschreven.

In tegenstelling tot eigenschappen (en variabelen), worden methoden niet gebruikt als plaatsaanduidingen voor waarden. Sommige methoden kunnen echter berekeningen uitvoeren en een resultaat retourneren dat kan worden gebruikt als een variabele. De methode `toString()` van de klasse Number zet bijvoorbeeld een numerieke waarde om in de bijbehorende tekstrepresentatie:

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

U kunt de methode `toString()` bijvoorbeeld gebruiken wanneer u de waarde van een variabele Getal wilt weergeven in een tekstveld op het scherm. De eigenschap `text` van de klasse TextField wordt gedefinieerd als een String, zodat deze alleen tekstwaarden kan bevatten. (De tekst `property` vertegenwoordigt de werkelijke tekstinhoud die op het scherm wordt weergegeven.) Deze coderegel zet de numerieke waarde in de variabele `numericData` om in tekst. De waarde wordt vervolgens weergegeven op het venster in het TextField-object `calculatorDisplay`:

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

Gebeurtenissen

Hierboven is een computerprogramma beschreven als een serie instructies die de computer stapsgewijs uitvoert. Sommige eenvoudige computerprogramma's bestaan inderdaad uit niets meer dan dat: een paar stappen die worden uitgevoerd door de computer, waarna het programma stopt. ActionScript-programma's zijn echter ontworpen om continu te worden uitgevoerd, terwijl wordt gewacht op gebruikersinvoer of iets anders. Gebeurtenissen vormen het mechanisme waarmee wordt bepaald welke instructies de computer uitvoert en wanneer.

In wezen is een *gebeurtenis* een voorval dat bekend is bij ActionScript en waarop gereageerd kan worden. Veel gebeurtenissen zijn gerelateerd aan gebruikersinteractie, zoals een gebruiker die op een knop klikt of op een toets op het toetsenbord drukt. Er zijn ook andere gebeurtenistypen. Wanneer u ActionScript bijvoorbeeld gebruikt om een externe afbeelding te laden, bestaat er een gebeurtenis die u kan waarschuwen wanneer de afbeelding is geladen. Wanneer er een ActionScript-programma wordt uitgevoerd, wacht het eigenlijk tot er een aantal dingen gebeuren. Wanneer die dingen gebeuren, wordt de ActionScript-code uitgevoerd, die u voor die gebeurtenissen hebt opgegeven.

Standaardgebeurtenisafhandeling

De techniek voor het opgeven van bepaalde handelingen die moeten worden uitgevoerd als reactie op specifieke gebeurtenissen, wordt *gebeurtenisafhandeling* genoemd. Wanneer u ActionScript-code schrijft om gebeurtenisafhandeling uit te voeren, zijn er drie belangrijke elementen die u moet identificeren:

- De gebeurtenisbron: het object waarop de gebeurtenis betrekking heeft. Voorbeelden zijn de knop waarop wordt geklikt of het object Loader dat de afbeelding laadt. De gebeurtenisbron wordt ook wel het *gebeurtenisdoel* genoemd. Deze wordt zo genoemd, omdat dit het object is waarop de computer de gebeurtenis richt (waar de gebeurtenis echt plaatsvindt).
- De gebeurtenis: het voorval waarop u wilt reageren. Het is belangrijk om deze gebeurtenis te identificeren, aangezien veel objecten meerdere gebeurtenissen activeren.
- De reactie: de stappen die u wilt uitvoeren wanneer de gebeurtenis plaatsvindt.

Wanneer u ActionScript-code wilt schrijven om gebeurtenissen uit te voeren, hebt u deze drie elementen nodig. De code volgt een basisstructuur (elementen in vetgedrukte letters zijn voorbeelden die u invult voor geval):

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

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

In deze code gebeuren twee dingen. Eerst wordt een functie gedefinieerd, waarin de handelingen worden opgegeven die u wilt uitvoeren als reactie op de gebeurtenis. Vervolgens wordt de methode `addEventListener()` van het bronobject opgeroepen. Wanneer u `addEventListener()` oproept, wordt de functie aan een bepaalde gebeurtenis 'toegeschreven'. Wanneer de gebeurtenis plaatsvindt, worden de acties van de functie uitgevoerd. Hieronder worden deze onderdelen nader beschreven.

Met een *functie* kunt u handelingen groeperen onder een enkele naam, die fungeert als een soort snelkoppeling om de handelingen uit te voeren. Een functie is identiek aan een methode, behalve dat deze niet noodzakelijk aan een bepaalde klasse is gerelateerd. (De term “methode” kan worden gedefinieerd als een functie die gerelateerd is aan een bepaalde klasse.) Wanneer u een functie maakt voor het uitvoeren van een gebeurtenis, kiest u een naam voor deze functie (in dit geval `eventResponse`). U kunt ook één parameter opgeven (in dit voorbeeld `eventObject`). Het opgeven van een functieparameter lijkt op het declareren van een variabele. U moet dus ook het gegevenstype van de parameter aangeven. (In dit voorbeeld is het gegevenstype van de parameter `EventType`.)

Aan elke type gebeurtenis waarnaar u kunt luisteren is een ActionScript-klasse gekoppeld. Het gegevenstype dat u voor de functieparameter opgeeft is altijd de gekoppelde klasse van de specifieke gebeurtenis waarop u wilt reageren. Een gebeurtenis `click` (die wordt getriggerd wanneer de gebruiker met de muis op een item klikt) is bijvoorbeeld gekoppeld aan de klasse MouseEvent. Als u een listenerfunctie wilt schrijven voor een gebeurtenis `click`, definieert u de listenerfunctie met een parameter met het gegevenstype MouseEvent. Ten slotte schrijft u tussen de accolades openen en sluiten (`{ ... }`) de instructies die uitgevoerd moeten worden wanneer de gebeurtenis plaatsvindt.

De functie voor het behandelen van de gebeurtenis is nu geschreven. Vervolgens vertelt u het gebeurtenisbronobject (het object waarin de gebeurtenis plaatsvindt, zoals de knop) waarmee u de functie wilt oproepen wanneer de gebeurtenis plaatsvindt. U kunt uw functie bij het gebeurtenisbronobject registreren door de methode `addEventListener()` van dit object op te roepen (alle objecten met gebeurtenissen die ook een methode (`addEventListener()` bevatten). Voor de methode `addEventListener()` zijn twee parameters nodig:

- Als eerste de naam van de specifieke gebeurtenis waarop moet worden gereageerd. Elke gebeurtenis is gekoppeld aan een bepaalde klasse. Elke gebeurtenisklasse heeft een bepaalde waarde, die een unieke naam heeft, opgegeven voor elke gebeurtenis in deze klasse. Gebruik deze waarde voor de eerste parameter.
- Als tweede de naam van de functie die reageert op de gebeurtenis. Een functienaam wordt zonder haakjes geschreven wanneer deze als parameter wordt doorgegeven.

Gebeurtenissen verwerken

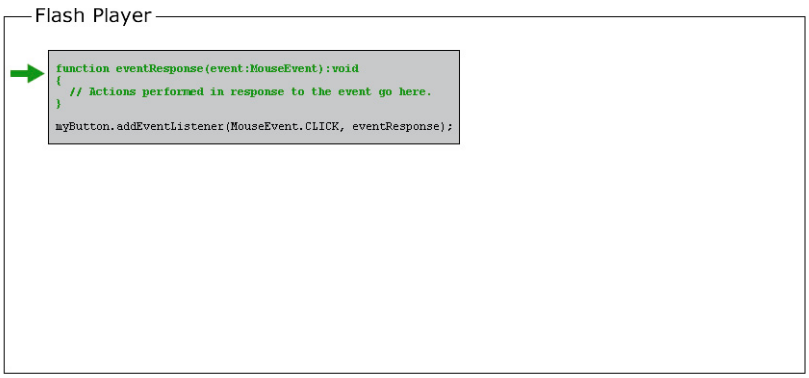
Hieronder volgt een stapsgewijze beschrijving van het proces dat plaatsvindt wanneer u een gebeurtenislistener maakt. In dit voorbeeld wordt een listenerfunctie gemaakt die wordt aangeroepen wanneer op een object `myButton` wordt geklikt.

De code die is geschreven door de programmeur ziet er als volgt uit:

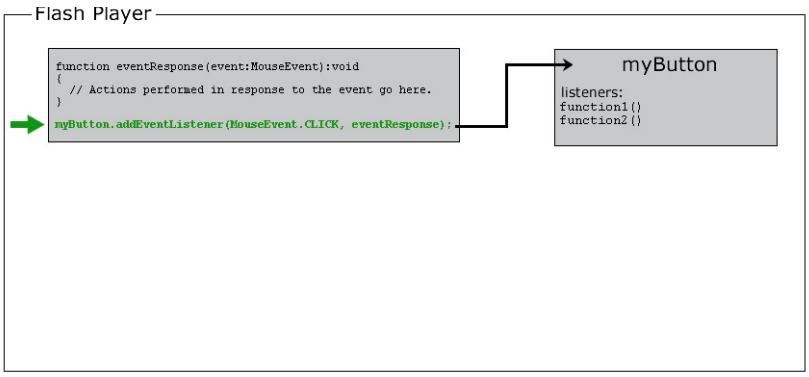
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

Hieronder wordt beschreven hoe deze code werkt wanneer deze wordt uitgevoerd:

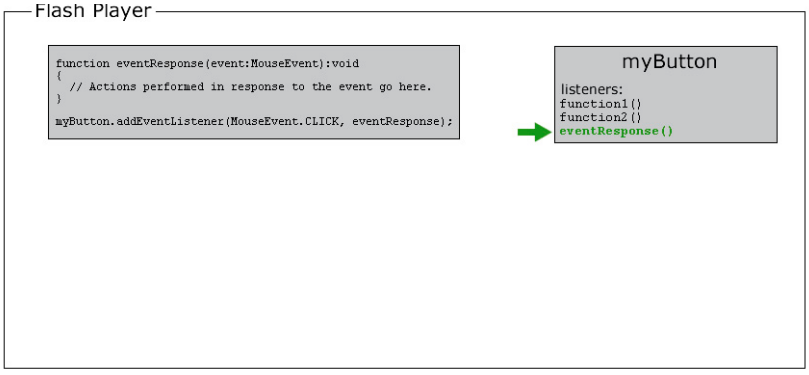
1 Wanneer het SWF-bestand wordt geladen, houdt de computer bij dat deze de functie `eventResponse()` bevat.



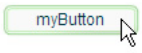
2 Vervolgens wordt de code (met name de coderegels die zich niet in een functie bevinden) uitgevoerd door de computer. In dit geval is dat slechts één regel code: de methode `addEventListener()` van het gebeurtenisbronobject (`myButton`) aanroepen en de functie `eventResponse` doorgeven als parameter.



Intern houdt `myButton` een lijst bij van alle functies die naar elke van deze gebeurtenissen horen. Wanneer de methode `addEventListener()` wordt opgeroepen, slaat `myButton` de functie `eventResponse()` op in de lijst met gebeurtenislisteners.

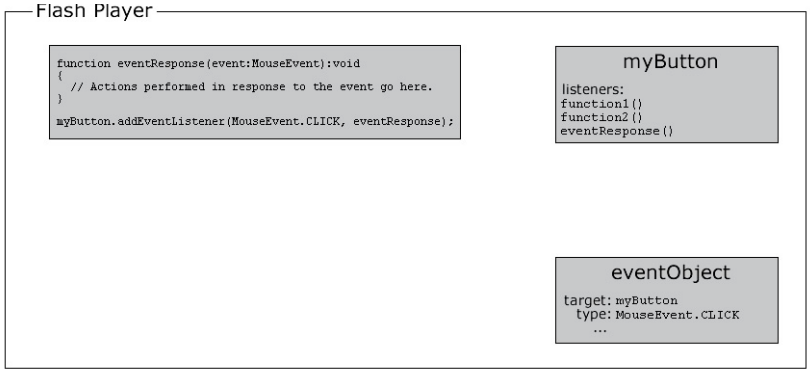


3 Op een gegeven moment klikt de gebruiker op het object `myButton`, waardoor de gebeurtenis `click` wordt geactiveerd (in de code geïdentificeerd als `MouseEvent.CLICK`).

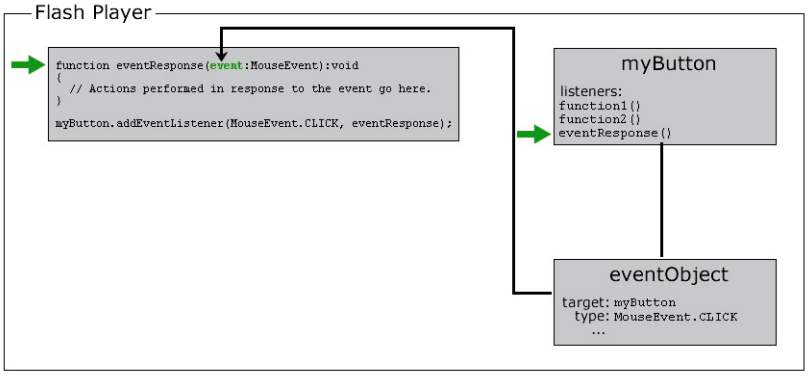


Op dat moment gebeurt het volgende:

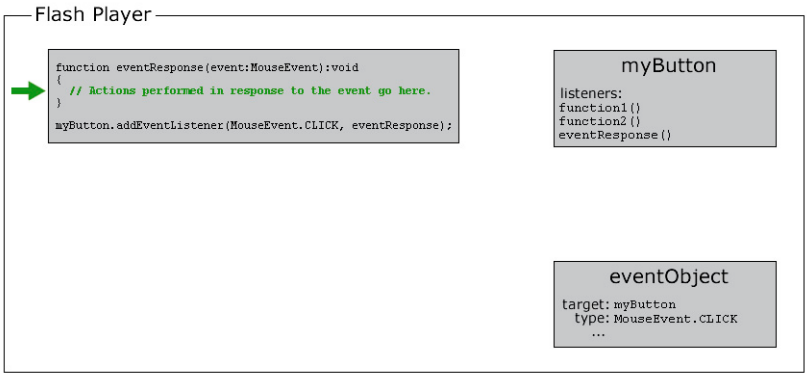
a Er wordt een object gemaakt als instantie van de klasse die gerelateerd is aan de gebeurtenis (in dit voorbeeld MouseEvent). Dit object is vaak een instantie van de klasse Gebeurtenis. Voor muisgebeurtenissen is dit een instantie MouseEvent. Voor andere gebeurtenissen is dit een instantie van de klasse die aan die gebeurtenis gerelateerd is. Het gemaakte object wordt het *gebeurtenisobject* genoemd en bevat bepaalde gegevens over de gebeurtenis die heeft plaatsgevonden: het type gebeurtenis, waar deze heeft plaatsgevonden en andere informatie over de gebeurtenis indien van toepassing.



b Vervolgens controleert de computer de lijst met gebeurtenislisteners die zijn opgeslagen door `myButton`. Deze functies worden één voor één doorlopen, waarbij elke functie wordt opgeroepen en het gebeurtenisobject aan de functie wordt doorgegeven als parameter. Aangezien de functie `eventResponse()` een van de listeners is van `myButton`, roept de computer de functie `eventResponse()` op als onderdeel van dit proces.



c Wanneer de functie `eventResponse()` wordt aangeroepen, wordt de code in deze functie uitgevoerd en worden daarmee de opgegeven handelingen uitgevoerd.



Voorbeelden van gebeurtenisafhandeling

Hier zijn nog een paar concretere voorbeelden van de gebeurtenisafhandelingscode. Deze voorbeelden zijn bedoeld om u een aantal algemene gebeurteniselementen en mogelijk beschikbare variaties voor het schrijven van gebeurtenisafhandelingscode te laten zien:

- Op een knop klikken om de huidige filmclip af te spelen. In het volgende voorbeeld is `playButton` de instantienaam van de knop en is `this` een speciale naam die staat voor 'het huidige object':

```
this.stop();

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

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

- Detecteren dat er in een tekstveld wordt getypt. In dit voorbeeld is `entryText` een invoertekstveld en is `outputText` een dynamisch tekstveld:

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

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

- Op een knop klikken om naar een URL te navigeren. In dit geval is `linkButton` de instantienaam van de knop:

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

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

Objectinstanties maken

Voordat u een object kunt gebruiken in ActionScript, moet dit object uiteraard wel eerst bestaan. Een onderdeel van het maken van een object is het declareren van een variabele. Hiermee wordt echter alleen een lege plaats in het computergeheugen gemaakt. Voordat u een variabele kunt gebruiken of manipuleren, moet u er een werkelijke waarde aan toekennen (dat wil zeggen, een object maken en dit in de variabele opslaan). Het proces om een object te maken wordt ook wel het *instantiëren* van het object genoemd. U maakt met andere woorden een instantie van een bepaalde klasse.

Een eenvoudige manier om een objectinstantie te maken is zonder ActionScript. Plaats in Flash Professional een filmclipsymbool, knopsymbool of tekstveld op het werkgebied en wijs er een instantienaam aan toe. Flash professional declareert automatisch een variabele met die instantienaam, maakt een objectinstantie en slaat dat object in een variabele op. In Flex kunt u een component in MXML maken, door een MXML-tag te coderen of het component op de editor in de modus Flash Builder Design te plaatsen. Wanneer u aan dat component een id toewijst, wordt die id de naam van een ActionScript-variabele met een componentinstantie.

U wilt echter een object misschien niet altijd visueel maken en dit is niet mogelijk voor niet-visuele objecten. Er zijn verschillende andere manieren waarop u alleen met ActionScript objectinstanties kunt maken.

Allereerst kunt u voor verschillende gegevenstypen in ActionScript een instantie maken met behulp van een *letterlijke expressie*. Dit is een waarde die rechtstreeks in de ActionScript-code wordt geschreven. Hieronder volgen enkele voorbeelden:

- Letterlijke numerieke waarde (voer het getal rechtstreeks in):

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

- Letterlijke tekenreekswaarde (plaats dubbele aanhalingtekens rond de tekst):

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

- Letterlijke Booleaanse waarde (gebruik de letterlijke waarden `true` of `false`):

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

- Letterlijke arraywaarde (laat een door komma's gescheiden lijst met waarden teruglopen in vierkante haken):

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

- Letterlijke XML-waarde (voer de XML rechtstreeks in):

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

In ActionScript zijn ook letterlijke expressies gedefinieerd voor de gegevenstypen Array, RegExp, Object en Function.

De meest algemene manier om voor elk gegevenstype een instantie te maken is met behulp van de operator `new` met de klassenaam, zoals hier wordt weergegeven:

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

Het maken van een object met behulp van de operator `new` wordt doorgaans het 'aanroepen van de klassenconstructor' genoemd. Een *constructor* is een speciale methode die wordt aangeroepen tijdens het proces waarin een instantie van een klasse wordt gemaakt. Wanneer u op deze manier een instantie maakt, moet u erop letten dat u na de klassenaam een haakje plaatst. Soms geeft u tussen deze haakjes parameterwaarden op. Dit zijn twee dingen die u ook doet wanneer u een methode oproept.

Zelfs bij gegevenstypen waarvoor u instanties kunt maken met behulp van een literale expressie, kunt u de operator `new` gebruiken om een objectinstantie te maken. De volgende twee coderegels doen bijvoorbeeld precies hetzelfde:

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

Het is van belang dat u bekend met de manier `new`*ClassName*`()` om objecten te maken. Veel ActionScript-gegevenstypen worden niet visueel weergegeven. U kunt ze niet maken door een item in het werkgebied van Flash Professional, de ontwerpmodus of de MXML-editor van Flash Builder te plaatsen. U kunt in ActionScript van een van deze gegevenstypen alleen een instantie maken met de operator `new`.

Adobe Flash Professional
In Flash Professional kan de operator `new` ook worden gebruikt om een instantie te maken van een filmclipsymbool dat in de bibliotheek is gedefinieerd maar niet in het werkgebied is geplaatst.

Meer Help-onderwerpen

Werken met arrays

Reguliere expressies gebruiken

MovieClip-objecten met ActionScript maken

Algemene programmaonderdelen

Er zijn verschillende aanvullende bouwstenen om een ActionScript-programma mee te maken.

Operatoren

Operatoren zijn speciale symbolen (soms woorden) die worden gebruikt om berekeningen uit te voeren. Operatoren worden voornamelijk gebruikt voor wiskundige bewerkingen en ook wanneer waarden met elkaar worden vergeleken. Een operator wordt gebruikt met één of meer waarden en leidt tot een enkel resultaat. Bijvoorbeeld:

- De operator voor optellen (`+`) telt twee waarden bij elkaar op, met als resultaat een enkel getal:

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

- De operator voor vermenigvuldigen (`*`) vermenigvuldigt een waarde met een andere waarde, met als resultaat een enkel getal:

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

- De gelijkheidsoperator (`==`) vergelijkt twee waarden om vast te stellen of deze gelijk zijn, met als resultaat een enkele Booleaanse waarde (waar of onwaar):

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

Zoals u hierboven ziet, worden de gelijkheidsoperator en de andere 'vergelijkingsoperatoren' vaak gebruikt met de instructie `if` om vast te stellen of bepaalde instructies worden uitgevoerd of niet.

Opmerkingen

Wanneer u ActionScript schrijft, wilt u graag voor uzelf notities maken. U wilt bijvoorbeeld soms uitleggen hoe bepaalde coderegels werken of waarom u een bepaalde keuze hebt gemaakt. *Commentaar* is een gereedschap waarmee u tekst kunt schrijven die de computer negeert in uw code. ActionScript kent twee soorten commentaar:

- Commentaar op één regel: Commentaar van een enkele regel geeft u op door ergens op een regel twee slashes te plaatsen. De computer negeert alles vanaf de slashes tot het einde van die regel:

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

- Commentaar van meerdere regels: Commentaar van meerdere regels bevat een beginmarkering voor commentaar (`/*`), vervolgens de inhoud van het commentaar en daarna een eindmarkering voor commentaar (`*/`). De computer negeert alles tussen de begin- en eindmarkeringen, onafhankelijk van het aantal regels van de opmerking.

  ```
  /*
  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.
  */
  ```

Een andere algemene toepassing van opmerkingen is om tijdelijk een of meer coderegels 'uit te schakelen'. U kunt bijvoorbeeld opmerkingen gebruiken wanneer u een andere manier uitprobeert om iets te doen. U kunt deze ook gebruiken om te bepalen waarom een bepaalde ActionScript-code niet werkt zoals u had verwacht.

Verloopbeheer

Vaak wilt u in een programma bepaalde handelingen herhalen, alleen specifieke handelingen uitvoeren en andere niet, alternatieve handelingen uitvoeren op basis van bepaalde voorwaarden enzovoort. Met *verloopbeheer* kunt u bepalen welke handelingen worden uitgevoerd. Er zijn verschillende typen elementen beschikbaar in ActionScript om het verloop te beheren:

- Functies: functies lijken op sneltoetsen. U kunt hiermee een serie handelingen groeperen onder een enkele naam. U kunt een functie ook gebruiken om berekeningen uit te voeren. Functies zijn met name van belang voor het afhandelen van gebeurtenissen, maar worden ook gebruikt als algemeen gereedschap om een serie instructies te groeperen.
- Lussen: Met lusstructuren kunt u een set instructies aanwijzen die een ingesteld aantal keer worden uitgevoerd of worden uitgevoerd totdat aan een bepaalde voorwaarde wordt voldaan. Vaak wordt een lus gebruikt om een aantal verwante items te manipuleren met behulp van een variabele, waarvan de waarde telkens wanneer de lus wordt doorlopen, wordt gewijzigd.
- Voorwaardelijke instructies: voorwaardelijke instructies bieden een manier om bepaalde instructies toe te wijzen die alleen in bepaalde situaties moeten worden uitgevoerd. Ze worden ook gebruikt om alternatieve sets instructies voor verschillende voorwaarden te bieden. De meest gebruikte voorwaardelijke instructie is de instructie `if` . De instructie `if` controleert de waarde of expressie die tussen de haakjes staat. Als de waarde `true` is, worden de coderegels tussen accolades uitgevoerd. Anders worden deze genegeerd. Bijvoorbeeld:

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

Bij de instructie `if` hoort de instructie `else`. Hiermee kunt u alternatieve instructies opgeven die de computer uitvoert wanneer de voorwaarde niet `true` is:

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

Voorbeeld: een portfolio met animatie (Flash Professional)

Dit voorbeeld is ontworpen om u voor het eerst te laten zien hoe u gedeelten van ActionScript kunt samenvoegen tot een volledige toepassing. Het animatieportfoliostuk is een voorbeeld van hoe u aan een bestaande lineaire animatie kleine interactieve elementen toe kunt voegen. U kunt bijvoorbeeld in een online portfolio een animatie toevoegen die u voor een klant hebt gemaakt. Het interactieve gedrag dat we aan de animatie toevoegen bestaat uit twee knoppen waarop de kijker kan klikken: een die de animatie start en een met een koppeling naar een andere URL (bijvoorbeeld het portfoliomenu of de introductiepagina van de maker).

Het proces voor het maken van dit stuk bestaat uit de volgende hoofdonderdelen:

1. Het FLA-bestand voorbereiden om ActionScript en interactieve elementen toe te voegen.
2. De knoppen maken en toevoegen.
3. De ActionScript-code schrijven.
4. De toepassing testen.

Voorbereiden op toevoegen van interactiviteit

Voordat u interactieve elementen aan de animatie kunt toevoegen, is het nuttig om het FLA-bestand voor te bereiden door een aantal plaatsen toe te voegen voor de nieuwe inhoud. Deze taak omvat het maken van werkelijke ruimte in het werkgebied waar de knoppen zich bevinden. Het omvat ook het maken van 'ruimte' in het FLA-bestand, zodat verschillende items apart worden gehouden.

U kunt het FLA-bestand als volgt voorbereiden op het toevoegen van interactieve elementen:

1. Maak een FLA-bestand met een eenvoudige animatie zoals een enkele bewegings-tween of vorm-tween. Als u al een FLA-bestand hebt met de animatie die u in het project weergeeft, opent u dat bestand en slaat dit op met een nieuwe naam.
2. Bepaal waar op het scherm u de twee knoppen wilt laten verschijnen. Een knop om de animatie te starten en een knop om te koppelen aan de auteurportfolio of introductiepagina. Zo nodig kunt u in het werkgebied wat ruimte voor de nieuwe inhoud wissen of toevoegen. Als de animatie er nog geen heeft, kunt u op het eerste frame een opstartscherm maken. In dit geval wilt u de animatie waarschijnlijk verplaatsen, zodat deze op frame 2 of later start.

3 Voeg een nieuwe laag toe boven de andere lagen in de tijdlijn en noem deze laag **knoppen**. In deze laag voegt u knoppen toe.

4 Voeg een nieuwe laag toe boven de laag knoppen en noem deze laag **handelingen**. In deze laag voegt u ActionScript-code aan de toepassing toe.

Knoppen maken en toevoegen

Vervolgens maakt en plaatst u de knoppen die de kern van de interactieve toepassing vormen.

U kunt als volgt knoppen maken en toevoegen aan het FLA-bestand:

1 Maak met behulp van de tekengereedschappen het uiterlijk van de eerste knop (de 'afspeelknop') in de laag knoppen. U kunt bijvoorbeeld een horizontale ovaal maken met tekst erop.

2 Selecteer met het gereedschap Selecteren alle grafische delen van de enkele knop.

3 Selecteer Wijzigen > Omzetten in symbool in het hoofdmenu.

4 Selecteer in het dialoogvenster Knop als symbooltype, geef het symbool een naam en klik op OK.

5 Geef de knop, terwijl deze is geselecteerd, in Eigenschapcontrole de instantienaam **playButton**.

6 Herhaal de stappen 1 tot en met 5 om de knop te maken waarmee de gebruiker naar de introductiepagina van de maker navigeert. Noem deze knop **homeButton**.

De code schrijven

De ActionScript-code voor deze toepassing kan naar functionaliteit worden opgedeeld in drie onderdelen, hoewel alle code op dezelfde plaats worden ingevoerd. De code heeft drie functies:

- De afspeelkop stoppen zodra het SWF-bestand wordt geladen (wanneer de afspeelkop frame 1 ingaat).
- Luisteren naar een gebeurtenis zodat het SWF-bestand wordt afgespeeld wanneer de gebruiker op de afspeelknop klikt.
- Luisteren naar een gebeurtenis zodat de browser naar de juiste URL wordt gestuurd wanneer de gebruiker op de knop voor de introductiepagina klikt.

U kunt als volgt de code schrijven die de afspeelkop stopt wanneer deze frame 1 ingaat:

1 Selecteer het hoofdframe in frame 1 van de laag handelingen.

2 Selecteer Venster > Handelingen in het hoofdmenu om het deelvenster Handelingen te openen.

3 Voer de volgende code in het Script-veld in:

```
stop();
```

U kunt als volgt de code schrijven die de animatie start wanneer op de afspeelknop wordt geklikt:

1 Voeg twee lege regels toe aan het einde van de code die u in de vorige stappen hebt ingevoerd.

2 Voer onderaan het script de volgende code in:

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

Deze code definieert een functie `startMovie()`. Wanneer `startMovie()` wordt aangeroepen, begintde hoofdtijdlijn met afspelen.

3 Voer op een nieuwe regel de volgende code in, na de code die u in de vorige stap hebt ingevoerd:

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

Deze coderegel registreert de functie `startMovie()` als een listener voor de gebeurtenis `click` van `playButton`. Met andere woorden, wanneer op de knop `playButton` wordt geklikt, wordt de functie `startMovie()` aangeroepen.

U kunt als volgt de code schrijven die de browser naar een URL stuurt wanneer op de knop voor de introductiepagina wordt geklikt.

1 Voeg twee lege regels toe aan het einde van de code die u in de vorige stappen hebt ingevoerd.

2 Voer onderaan het script de volgende code in:

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

Deze code definieert een functie `gotoAuthorPage()`. Deze functie maakt eerst een URLRequest-instantie met de URL http://example.com/. Hij geeft die URL vervolgens door aan de functie `navigateToURL()`, waardoor de browser van de gebruiker die URL opent.

3 Voer op een nieuwe regel de volgende code in, na de code die u in de vorige stap hebt ingevoerd:

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

Deze coderegel registreert de functie `gotoAuthorPage()` als een listener voor de gebeurtenis `click` van `homeButton`. Met andere woorden, wanneer op de knop `homeButton` wordt geklikt, wordt de functie `gotoAuthorPage()` aangeroepen.

De toepassing testen

De toepassing is nu volledig functioneel. U kunt een test uitvoeren om te zien of dit inderdaad het geval is.

U kunt als volgt de toepassing testen:

1 Selecteer Besturing > Film testen in het hoofdmenu. Het SWF-bestand wordt gemaakt en geopend in een Flash Professional-venster.

2 Klik op beide knoppen om te controleren of deze naar verwachting functioneren.

3 Wanneer de knoppen niet werken, kunt u het volgende controleren:

- Hebben de knoppen beide verschillende instantienamen?
- Gebruiken de aanroepen van de methode `addEventListener()` dezelfde namen als de instantienamen van de knoppen?
- Worden de juiste gebeurtenisnamen gebruikt in de aanroepen van de methode `addEventListener()`?
- Is de juiste parameter opgegeven voor elk van de functies? (Beide methodes moeten een enkele parameter hebben met het gegevenstype MouseEvent.)

Al deze fouten en de meeste andere mogelijke fouten leveren een foutbericht op. het foutbericht kan verschijnen wanneer u de opdracht Film testen kiest of wanneer u op de knop klikt terwijl u het project test. Kijk in het deelvenster Compilatiefouten voor compilatiefouten (die verschijnen wanneer u voor het eerst Film testen kiest). In het deelvenster Uitvoer vind u uitvoerfouten die optreden wanneer de inhoud wordt afgespeeld, zoals wanneer u op een knop klikt.

Toepassingen ontwikkelen met ActionScript

Wanneer u een toepassing wilt ontwikkelen met ActionScript, hebt u meer nodig dan alleen kennis van de syntaxis en de namen van de klassen die u gebruikt. De meeste Flash Platform-documentatie gaat over twee onderwerpen (syntaxis en het gebruik van ActionScript-klassen). Om een ActionScript-toepassing te bouwen wilt u waarschijnlijk ook meer informatie over de volgende onderwerpen:

- Welke programma's kunnen worden gebruikt voor het schrijven met ActionScript?
- Hoe wordt ActionScript-code georganiseerd?
- Hoe kan ActionScript-code in een toepassing worden toegevoegd?
- Welke stappen moet u volgen bij het ontwikkelen van een ActionScript-toepassing?

Opties voor het ordenen van code

U kunt ActionScript 3.0-code gebruiken voor het aansturen van eenvoudige grafische animaties tot complexe systemen voor client-servertransactieverwerking. Afhankelijk van het type toepassing dat u ontwikkelt, kunt u kiezen voor één of meer van de volgende mogelijkheden om ActionScript in uw project op te nemen.

Code opslaan in frames in een Flash Professional-tijdlijn

In Flash Professional kunt u ActionScript-code toevoegen aan een willekeurig frame in een tijdlijn. Deze code wordt uitgevoerd tijdens het afspelen van de film, wanneer de afspeelkop het betreffende frame ingaat.

Het plaatsen van ActionScript-code in frames is een eenvoudige manier om gedragingen toe te voegen aan een toepassing die is ontwikkeld met Flash Professional. U kunt code toevoegen aan een willekeurig frame in de hoofdtijdlijn van een filmclipsymbool. Deze flexibiliteit heeft echter een nadeel. Wanneer u grotere toepassingen ontwikkelt, verliest u gemakkelijk het overzicht over welke frames welke scripts bevatten. Na verloop van tijd kan de complexe structuur het onderhoud van de toepassing bemoeilijken.

Veel ontwikkelaars vereenvoudigen de organisatie van ActionScript-code in Flash Professional door code alleen te plaatsen in het eerste frame van een tijdlijn of in een specifieke laag in het Flash-document. Wanneer u uw code onderverdeeld, wordt het eenvoudiger om de code in FLA-bestanden van Flash te vinden en te onderhouden. Dezelfde code kan echter niet worden gebruikt in een ander Flash Professional-project zonder de code eerst in een nieuw bestand te kopiëren.

Om het gemakkelijker te maken om uw ActionScript-code later in andere Flash Professional-projecten te gebruiken, slaat u uw code op in externe ActionScript-bestanden (tekstbestanden met de .as-extensie).

Code in Flex MXML-bestanden insluiten

In een Flex-ontwikkelingsomgeving zoals Flash Builder, kunt u ActionScript-code toevoegen in een `<fx:Script>`-tag in een MXML-bestand. Deze techniek kan grote projecten complexer maken en het moeilijker maken om dezelfde code in een ander Flex-project te gebruiken; Om het gemakkelijker te maken om uw ActionScript-code in toekomstige projecten te gebruiken, slaat u uw code op in externe ActionScript-bestanden.

Opmerking: *U kunt een bronparameter opgeven voor een `<fx:Script>`-tag. Als u een bronparameter gebruikt, kunt u ActionScript-code uit een extern bestand importeren alsof deze rechtstreeks in de `<fx:Script>`-tag is ingevoerd. Het bronbestand dat u gebruikt kan geen eigen klasse definiëren, waardoor hergebruik beperkt is.*

Code opslaan in ActionScript-bestanden

Als een project veel ActionScript-code bevat, kunt u de code het best ordenen in afzonderlijke ActionScript-bronbestanden (tekstbestanden met de extensie .as). Een ActionScript-bestand kan op twee manieren worden gestructureerd, afhankelijk van het gebruiksdoel in de toepassing.

- Ongestructureerde ActionScript-code: regels ActionScript-code, met instructies of functiedefinities, die zijn geschreven alsof deze rechtstreeks zijn ingevoerd in een tijdlijnscript of MXML-bestand.

  ActionScript die op deze manier is geschreven, kunt u benaderen met de instructie `include` in ActionScript of met de tag `<fx:Script>` in Flex MXML. De ActionScript-instructie `include` vertelt de compiler om de inhoud van een extern ActionScript-bestand op een bepaalde locatie en binnen een bepaald bereik in een script toe te voegen. Het resultaat is hetzelfde als wanneer de code daar rechtstreeks werd ingevoerd. Wanneer u in MXML-taal een `<fx:Script>`-tag gebruikt met een bronattribuut, geeft dit een extern ActionScript aan dat door de compiler op dat punt in de toepassing wordt geladen. Met de volgende tag wordt bijvoorbeeld het externe ActionScript-bestand Box.as geladen:

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

- ActionScript-klassendefinitie: Een definitie van een ActionScript-klasse, inclusief de methode- en eigenschapdefinities.

  Wanneer u een klasse definieert kunt u de ActionScript-code in de klasse openen, door een instantie van de klasse te maken en zijn eigenschappen, methoden en gebeurtenissen te gebruiken. Gebruik van uw eigen klassen is identiek aan het gebruik van ingebouwde ActionScript-klassen en vereist twee delen:

  - Geef de volledige naam van de klasse op met behulp van de instructie `import`, zodat de compiler van ActionScript de klasse kan vinden. Gebruik bijvoorbeeld om de Filmclip-klasse in ActionScript te gebruiken, importeert u de klasse met de volledige naam, inclusief pakket en klasse:

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

    U kunt ook het pakket importeren dat de klasse MovieClip bevat. Dit staat gelijk aan het schrijven van afzonderlijke instructies `import` voor elke klasse in het pakket:

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

    De bovenste klassen zijn de enige uitzondering op de regel, dat een klasse geïmporteerd moet zijn om die klasse in uw code te gebruiken. Die klassen zijn niet gedefinieerd in een pakket.

  - Code schrijven die specifiek de klassenaam gebruikt. Declareer bijvoorbeeld een variabele met die klasse als gegevenstype en maak een instantie van de klasse om in de variabele op te slaan. Door een klassennaam in ActionScript-code te gebruiken, geeft u aan de compiler door dat de definitie van die klasse moet worden geladen. Bij een externe klasse Box, maakt deze instructie een instantie van de Box-klasse:

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

    De eerste keer dat de compiler de referentie naar de Box-klasse tegenkomt, zoekt deze in de beschikbare broncode naar de locatie van de Box-klassedefinitie.

Het juiste gereedschap kiezen

U kunt een van verschillende gereedschappen (of meerdere gereedschappen tegelijk) gebruiken voor het schrijven en bewerken van uw ActionScript-code.

Flash Builder

Adobe Flash Builder is het belangrijkste gereedschap voor het maken van projecten met het Flex-framework of projecten die vooral uit ActionScript-code bestaan. Flash Builder bevat ook een volledige ActionScript-editor en mogelijkheden voor visuele lay-out en MXML-bewerking. U kunt hiermee alleen-ActionScript en Flex-projecten maken. Flex biedt verschillende voordelen, zoals een uitgebreide verzameling van vooraf ontwikkelde besturingselementen voor gebruikersinterfaces, flexibele, dynamische besturingselementen voor lay-out en ingebouwde mechanismen voor het werken met externe gegevensbronnen en het koppelen van externe gegevens aan gebruikersinterface-elementen. Omdat de extra vereiste code deze functies echter biedt, kunnen projecten met Flex een grotere SWF-bestand hebben dan niet-Flex projecten.

Gebruik Flash Builder als u complete gegevensgeoriënteerde internettoepassingen met Flex maakt. Gebruik het als u ActionScript-code of MXML-code wilt bewerken en uw toepassing visueel wilt ontwerpen met slechts één gereedschap.

Veel Flash professional-gebruikers die ActionScript-georiënteerde projecten maken, gebruiken Flash Professional om visuele attributen te maken en Flash Builder als een editor voor ActionScript-code.

Flash Professional

Naast mogelijkheden voor het maken van afbeeldingen en animatie, heeft Flash Professional ook gereedschappen voor werken met ActionScript-code. De code kan worden gekoppeld aan elementen in een FLA-bestand of in externe alleen-ActionScript-bestanden. Flash Professional is perfect voor projecten die veel animatie of video bevatten. Het is erg nuttig wanneer u zelf de meeste afbeeldingen wilt maken. Als u Flash Professional gebruikt voor het ontwikkelen van uw ActionScript-project, kunt u visuele middelen maken en code schrijven met dezelfde toepassing. Flash Professional bevat ook vooraf gebouwde gebruikersinterfacecomponenten. U kunt die componenten ook gebruiken om een kleinere SWF-bestandsgrootte te verkrijgen en gebruik visuele gereedschappen om ze voor uw project te skinnen.

Flash Professional bevat twee gereedschappen om ActionScript-code te schrijven:

- Deelvenster Handelingen: met dit deelvenster, dat beschikbaar is wanneer u in een FLA-bestand werkt, kunt u ActionScript-code schrijven die is gekoppeld aan frames in een tijdlijn.
- Script-venster: het Script-venster is een speciale teksteditor voor het werken met ActionScript-codebestanden (.as).

ActionScript-editor van derden

Aangezien ActionScript-bestanden (.as) worden opgeslagen als eenvoudige tekstbestanden, kunt u elk programma waarmee onbewerkte tekst kan worden bewerkt, gebruiken om ActionScript-bestanden te schrijven. In aanvulling op de ActionScript-producten van Adobe zijn er door derden verschillende teksteditors ontwikkeld met specifieke ActionScript-mogelijkheden. U kunt met elke willekeurige teksteditor een MXML-bestand of ActionScript-klassen schrijven. U kunt dan van deze bestanden met behulp van Flex SDK een toepassing maken. Het project kan Flex gebruiken of is een alleen-ActionScript-toepassing. Sommige ontwikkelaars gebruiken ook Flash Builder of een ActionScript-editor van derden voor het schrijven van ActionScript-klassen, in combinatie met Flash Professional voor het maken van de grafische inhoud.

Redenen om te kiezen voor een ActionScript-editor van derden zijn:

- U wilt liever ActionScript-code in een apart programma schrijven en visuele elementen ontwerpen in Flash Professional.
- U gebruikt een toepassing om te programmeren in een andere taal dan ActionScript (zoals het maken van HTML-pagina's of het ontwikkelen van toepassingen in een andere programmeertaal) en u wilt dezelfde toepassing ook gebruiken voor uw ActionScript-code.

- U wilt projecten met alleen ActionScript of Flex-projecten maken met behulp van de Flex SDK zonder Flash Professional of Flex Builder.

Enkele bekende code-editors met specifieke ActionScript-ondersteuning zijn de volgende:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (met ActionScript en Flex-collecties)

Het ontwikkelingsproces voor ActionScript

Of uw ActionScript-project nu groot of klein is, als u werkt met een proces om uw toepassing te ontwerpen en ontwikkelen werkt u veel efficiënter en effectiever. De volgende stappen beschrijven een basisproces voor het ontwikkelen van een toepassing met ActionScript 3.0:

1 Ontwerp uw toepassing.

U moet op een of andere manier uw toepassing beschrijven voordat u deze gaat ontwikkelen.

2 Stel uw ActionScript 3.0-code samen.

U kunt ActionScript-code maken met Flash Professional, Flex Builder, Dreamweaver of een teksteditor.

3 Maak een Flash- of Flex-project om uw code uit te voeren.

In Flash Professional, maakt u een FLA-bestand, stelt u de publicatie-instellingen in, voegt u gebruikersinterfacecomponenten in de toepassing in en verwijst u naar de ActionScript-code. definieer de toepassing in Flex, voeg gebruikersinterfacecomponenten toe met MXML en verwijs naar de ActionScript-code.

4 Publiceer en test uw ActionScript-toepassing.

Wanneer u uw toepassing test, voert u uw toepassing uit vanaf uw ontwikkelingsomgeving en controleert u of deze alles uitvoert zoals u had bedoeld.

Het is niet noodzakelijk dat u de stappen in deze volgorde uitvoert of dat u een stap volledig voltooit voordat u aan een andere stap begint. U kunt bijvoorbeeld een scherm van uw toepassing ontwerpen (stap 1) en vervolgens de afbeeldingen, knoppen enzovoort maken (stap 3), voordat u ActionScript-code schrijft (stap 2) en test (stap 4). Of u ontwerpt een onderdeel ervan en voegt vervolgens één knop of interface-element tegelijk toe, waarvoor u ActionScript schrijft en tijdens het ontwikkelen tests uitvoert. Het is goed om deze vier fasen van het ontwikkelingsproces te onthouden. In de ontwikkelingspraktijk is het effectiever om zoals nodig tussen de fasen te wisselen.

Uw eigen klassen maken

Het proces van het maken van klassen om in uw projecten te gebruiken, kan u afschrikken. Het lastigste gedeelte van het maken van een klasse is echter het ontwerpen van de klasse en het identificeren van de methoden, eigenschappen en gebeurtenissen die deze bevat.

Strategieën voor het ontwerpen van een klasse

Objectgeoriënteerd ontwerpen is een complex onderwerp. Er zijn mensen die hun hele leven wijden aan de academische studie en professionele toepassing van deze discipline. Toch volgen hier een paar ideeën die u op weg kunnen helpen.

1 Denk goed na over de rol die instanties van een bepaalde klasse vervullen in de toepassing. In het algemeen hebben objecten één van de volgende rollen:

- Waardeobject: deze objecten dienen vooral als gegevenscontainers. Deze bevatten waarschijnlijk meerdere eigenschappen en minder methoden (of soms geen methoden). Ze zijn over het algemeen gecodeerde weergaven van duidelijk gedefinieerde items. Een muziekspelertoepassing kan bijvoorbeeld een Song-klasse bevatten die één echt nummer bevat en een Playlist-klasse die een conceptuele groep liedjes bevat.
- Weergaveobject: dit zijn objecten die op het scherm worden weergegeven. Voorbeelden zijn gebruikersinterface-elementen zoals een vervolgkeuzelijst of statusuitlezing, grafische elementen zoals een karakter in een videogame enzovoort.
- Toepassingsstructuur: deze objecten spelen verschillende ondersteunende rollen in de logica of verwerking die door toepassingen worden uitgevoerd. U kunt bijvoorbeeld een object maken dat bepaalde berekeningen uitvoert in een biologiesimulatie. U kunt een object maken dat verantwoordelijk is voor het synchroniseren van waarden tussen een kiesbediening en volume-informatie in een muziekspelertoepassing. Een ander object kan de regels in een videospel beheren. Of u kunt een klasse maken voor het laden van een opgeslagen afbeelding in een tekentoepassing.

2 Bepaal de specifieke functionaliteit die de klasse nodig heeft. De verschillende typen functionaliteit worden vaak de methoden van een klasse.

3 Wanneer de klasse moet dienen als een waardeobject, bepaalt u de gegevens die de instanties moeten bevatten. Deze gegevens zijn goede kandidaten voor eigenschappen.

4 Aangezien u de klassen specifiek voor uw project ontwerpt, is het belangrijkste uitgangspunt dat u de functionaliteit biedt die voor uw toepassing nodig is. Probeer deze vragen zelf te beantwoorden:

- Welke gegevens worden in uw toepassing opgeslagen, bijgehouden en gemanipuleerd? Het antwoord op deze vraag helpt u bij het identificeren welke waardeobjecten en eigenschappen u nodig hebt.
- Welke actiesets voert de toepassing uit? Wat gebeurt er bijvoorbeeld wanneer de toepassing eerst wordt opgeladen, wanneer er op een bepaalde knop wordt geklikt, wanneer een film ophoudt met spelen, enz.? Dit zijn goede kandidaten voor methoden. Dit kunnen ook eigenschappen zijn als de 'acties' bepaalde individuele waarden bevatten.
- Welke informatie is vereist voor het uitvoeren van een bepaalde actie? Deze gegevens worden de parameters van de methode.
- Van welke zaken die in een klasse veranderen tijdens het uitvoeren van de toepassing, moeten andere onderdelen van uw toepassing kennisnemen? Dit zijn goede kandidaten voor gebeurtenissen.

5 Is er een bestaand object dat vergelijkbaar is met het object dat u nodig hebt behalve dat hierbij een functionaliteit ontbreekt die u wilt toevoegen? Overweeg in dat geval om een subklasse te maken. (Een *subklasse* is een klasse die bouwt op de functionaliteit van een bestaande klassen in plaats van het definiëren van de volledige eigen functionaliteit.) Om bijvoorbeeld een klasse te maken dat op het scherm een visueel object is, gebruikt u het gedrag van een bestaand weergaveobject als basis voor uw klasse. In dat geval zou het weergaveobject (zoals MovieClip of Sprite) de *basisklasse* zijn en zou uw klasse deze klasse uitbreiden.

De code voor een klasse schrijven

Wanneer u eenmaal een ontwerp voor uw klasse hebt of ten minste een idee van de informatie die erin wordt opgeslagen en welke acties het uitvoert, is de werkelijke syntaxis van het schrijven van een klasse eenvoudig.

U moet minimaal de volgende stappen uitvoeren om uw eigen ActionScript-klasse te maken:

1 Open een nieuw tekstdocument in uw ActionScript-tekstbewerkingsprogramma.

2 Voer een instructie `class` in om de naam van de klasse te definiëren. Om een instructie `class` toe te voegen, voert u de woorden `public class` en de klassenaam toe. Voeg accolades openen en sluiten toe rond de inhoud van de klasse (de methode- en eigenschapdefinities). Bijvoorbeeld:

```
public class MyClass
{
}
```

Het woord `public` geeft aan dat de klasse kan worden benaderd vanuit elke andere code. Zie Naamruimteattributen voor toegangsbeheer voor andere alternatieven.

3 Typ een instructie `package` om de pakketnaam aan te geven die uw klasse bevat. De syntaxis is het woord `package`, gevolgd door de volledige pakketnaam, gevolgd door de accolades openen en sluiten rond het instructieblok `class`). Wijzig de code in de vorige stap bijvoorbeeld in het volgende:

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

4 Definieer elke eigenschap in de klasse met de instructie `var` binnen de hoofdtekst van de klasse. De syntaxis is dezelfde als u gebruikt om een willekeurige variabele te declareren (met toevoeging van de optie `public`). Wanneer bijvoorbeeld de volgende regels worden toegevoegd tussen beide accolades van de klassendefinitie, worden eigenschappen `textProperty`, `numericProperty` en `dateProperty` gemaakt:

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

5 Definieer elke methode in de klasse met dezelfde syntaxis die wordt gebruikt om een functie te definiëren. Bijvoorbeeld:

- Voor het maken van een methode `myMethod()`, voert u het volgende in:

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

- Voor het maken van een constructor (de speciale methode die wordt aangeroepen tijdens het proces waarin een instantie van een klasse wordt gemaakt), maakt u een methode met precies dezelfde naam als de naam van de klasse:

```
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);
}
```

Als u geen constructormethode in uw klasse invoegt, maakt de compiler automatisch een lege constructor in uw klasse. (Met andere woorden, een constructor zonder parameters en instructies.)

Er zijn weinig andere klasse-elementen die u kunt definiëren. Deze elementen zijn complexer.

- *Accessors* zijn een speciale kruising tussen een methode en een eigenschap. Wanneer u de code schrijft om de klasse te definiëren, schrijft u de accessor als een methode zodat u meerdere handelingen kunt uitvoeren (in plaats van alleen een waarde lezen of toekennen, wat alles is dat u kunt doen wanneer u een eigenschap definieert). Wanneer u echter een instantie van uw klasse maakt, behandelt u de accessor als een eigenschap en gebruikt u alleen de naam om de waarde te lezen of toe te kennen.
- Gebeurtenissen in ActionScript worden niet gedefinieerd met een specifieke syntaxis. In plaats daarvan definieert u gebeurtenissen in uw klasse met behulp van de functionaliteit van de EventDispatcher-klasse.

Meer Help-onderwerpen

Gebeurtenissen afhandelen

Voorbeeld: een basistoepassing maken

ActionScript 3.0 kan worden gebruikt binnen een aantal toepassingsontwikkelingsomgevingen, zoals Flash Professional en Flash Builder-gereedschappen of een andere teksteditor.

In dit voorbeeld worden de stappen doorlopen om een eenvoudige ActionScript 3.0-toepassing te maken en uit te breiden met het Flash Professional-programma of Flex Builder. De toepassing die hier wordt ontwikkeld, toont een eenvoudig patroon voor het gebruik van externe ActionScript 3.0-klassenbestanden in Flash Professional- en Flex-toepassingen.

Een ActionScript-toepassing ontwerpen

Deze ActionScript-voorbeeldtoepassing is een standaard 'Hello World'-toepassing, dus het ontwerp is eenvoudig:

- De toepassing heet HelloWorld.
- De toepassing geeft een enkel tekstveld weer met de woorden ‘Hello World!’.
- De toepassing gebruikt één objectgeoriënteerde klasse die Greeter heet. Dit ontwerp staat toe dat de klasse binnen een Flash Professional of Flex-project wordt gebruikt.
- In dit voorbeeld maakt u eerst een basisversie van de toepassing. Vervolgens voegt u een functionaliteit toe, waarin de gebruiker een gebruikernaam invoert en de toepassing de naam controleert in een lijst van onbekende gebruikers.

Aan de hand van deze beknopte beschrijving kunt u de toepassing zelf gaan ontwikkelen.

Het project HelloWorld en de klasse Greeter maken

In de ontwerpbeschrijving voor de toepassing HelloWorld staat de vereiste dat de code eenvoudig opnieuw kan worden gebruikt. Om dit te bereiken, gebruikt de toepassing één objectgeoriënteerde klasse die Greeter heet. U gebruikt deze klasse binnen een toepassing die u maakt in Flash Builder of Flash Professional.

Zo maakt u het HelloWorld-project en de klasse Greeter in Flex:

1 Selecteer in Flash Builder Bestand > Nieuw> Flex-project

2 Voer HelloWorld in als projectnaam. Controleer of het toepassingstype is ingesteld op 'Web (in Adobe Flash Player),' en klik op Voltooien.

Flash Builder maakt uw project en geeft het weer in Package Explorer. Standaard bevat het project al een bestand met de naam HelloWorld.mxml en dat bestand is geopend in het deelvenster Editor.

3 Maak nu een aangepast ActionScript-klassenbestand door in Flash Builder Bestand > Nieuw > ActionScript-klasse·te selecteren.

4 Ga in het dialoogvenster Nieuwe ActionScript-klasse naar het veld Naam en typ **Greeter** als naam voor de klasse en klik op Voltooien.

Er wordt een nieuw ActionScript-bewerkingsvenster weergegeven.

Ga nu verder met het toevoegen van code aan de klasse Greeter.

Een Greeter-klasse maken in Flash Professional:

1 In Flash Professional, selecteert u Bestand > Nieuw.

2 Selecteer ActionScript-bestand in het dialoogvenster Nieuw document en klik op OK.

Er wordt een nieuw ActionScript-bewerkingsvenster weergegeven.

3 Selecteer Bestand > Opslaan. Selecteer een map waarin de toepassing wordt opgeslagen, geef het ActionScript-bestand de naam **Greeter.as** en klik vervolgens op OK.

Ga nu verder naar code toevoegen aan de klasse Greeter.

Code toevoegen aan de klasse Greeter

De klasse Greeter definieert een object `Greeter`, dat u kunt gebruiken in de toepassing HelloWorld.

U kunt als volgt code toevoegen aan de klasse Greeter:

1 Typ de volgende code in het nieuwe bestand (het kan zijn dat sommige code al automatisch is toegevoegd):

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

De klasse Greeter bevat één methode `sayHello()` , die een tekenreeks retourneert met als tekst 'Hello World!'.

2 Selecteer Bestand > Opslaan om dit ActionScript-bestand op te slaan.

De klasse Greeter kan nu in een toepassing worden gebruikt.

Een toepassing maken die gebruikmaakt van uw ActionScript-code

De klasse Greeter die u hebt gemaakt, definieert een op zichzelf staande verzameling softwarefuncties, maar vertegenwoordigt niet een volledige toepassing. Om de klasse te gebruiken, maakt u een Flash Professional-document of Flex-project.

De code heeft een instantie van de Greeter-klasse nodig. Hieronder wordt beschreven hoe u de klasse Greeter voor uw toepassing gebruikt.

U kunt als volgt met Flex Professional een ActionScript-toepassing maken:

1. Selecteer Bestand > Nieuw.
2. Selecteer in het dialoogvenster Nieuw document het type Flash-bestand (ActionScript 3.0) en klik op OK.

   Er wordt een nieuw documentvenster weergegeven.
3. Selecteer Bestand > Opslaan. Selecteer dezelfde map die het klassenbestand Greeter.as bevat, geef het Flash-document de naam **HelloWorld.fla** en klik op OK.
4. In het gereedschapspalet van Flash Professional, selecteert u het gereedschap Tekst. Sleep het over het werkgebied om een nieuw tekstveld te definiëren van ongeveer 300 pixels breed en 100 pixels hoog.
5. Zorg ervoor dat in het deelvenster Eigenschappen het tekstveld in het werkgebied is geselecteerd, stel het teksttype in op Dynamische tekst en typ **mainText** als instantienaam van het tekstveld.
6. Klik op het eerste frame in de hoofdtijdlijn. Open het deelvenster Acties door Venster > Acties te selecteren.
7. Typ het volgende script in het deelvenster Handelingen:

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

8. Sla het bestand op.

Ga verder naar Uw ActionScript-toepassing publiceren en testen.

U kunt als volgt met Flash Builder een ActionScript-toepassing maken:

1. Open het bestand HelloWorld.mxml en voeg code toe die overeenkomt met wat hieronder wordt weergegeven:

```
<?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>
```

Dit Flex-project bevat vier MXML-tags:

- Een tag `<s:Application>`, waarin de container Application wordt gedefinieerd
- Een tag `<s:layout>`, waarin de lay-outstijl wordt (verticale lay-out) voor de tag Application wordt gedefinieerd
- Een tag `<fx:Script>` met daarin ActionScript-code
- Een tag `<s:TextArea>`, waarin een veld wordt gedefinieerd voor het weergeven van berichten van de gebruiker

De code in de tag `<fx:Script>` definieert een methode `initApp()` die wordt aangeroepen wanneer de toepassing wordt geladen. De methode `initApp()` stelt de tekstwaarde van het tekstgebied `mainTxt` in op de tekenreeks 'Hello World!' die wordt geretourneerd door de methode `sayHello()` van de aangepaste klasse Greeter, die u net hebt gemaakt.

2 Selecteer Bestand > Opslaan om de toepassing op te slaan.

Ga verder naar Uw ActionScript-toepassing publiceren en testen.

Uw ActionScript-toepassing publiceren en testen

Softwareontwikkeling is een herhalend proces. U schrijft wat code, probeert dit te compileren en bewerkt de code totdat deze goed wordt gecompileerd. U voert de gecompileerde toepassing uit en test of het voldoet aan het bedoelde ontwerp. Als dit niet het geval is, bewerkt u de code opnieuw tot dit wel zo is. De Flash Professional- en Flash Builder-ontwikkelomgevingen bieden diverse mogelijkheden voor het publiceren en testen van uw toepassingen en het opsporen van fouten.

Hieronder worden de basisstappen beschreven voor het testen van de toepassing HelloWorld in elke omgeving.

Zo publiceert en test u een ActionScript-toepassing met Flash Professional:

1. Publiceer de toepassing en controleer of er compilatiefouten optreden. Selecteer Besturing > Film testen in Flash professional om de ActionScript-code te compileren en de toepassing HelloWorld uit te voeren.
2. Als er in het venster Uitvoer fouten of waarschuwingen worden weergegeven tijdens het testen van uw toepassing, repareert u deze fouten in de bestanden HelloWorld.fla of HelloWorld.as. Test de toepassing opnieuw.
3. Wanneer geen compilatiefouten optreden, wordt in een Flash Player-venster de toepassing HelloWorld weergegeven.

U hebt een eenvoudige maar volledige objectgeoriënteerde toepassing met ActionScript 3.0 gemaakt. Ga verder naar De toepassing HelloWorld uitbreiden.

Zo publiceert en test u een ActionScript-toepassing met Flash Builder:

1. Selecteer Uitvoeren > HelloWorld uitvoeren.
2. De toepassing HelloWorld wordt gestart.
   - Als er in het venster Uitvoer fouten of waarschuwingen worden weergegeven tijdens het testen van uw toepassing, repareert u deze fouten in de bestanden HelloWorld.mxml of Greeter.as. Test de toepassing opnieuw.
   - Wanneer geen compilatiefouten optreden, wordt een browservenster geopend met daarin de toepassing HelloWorld. Als het goed is, wordt de tekst 'Hello World!' weergegeven.

   U hebt een eenvoudige maar volledige objectgeoriënteerde toepassing met ActionScript 3.0 gemaakt. Ga verder naar De toepassing HelloWorld uitbreiden.

De toepassing HelloWorld uitbreiden

Teneinde de toepassing iets interessanter te maken, wordt deze nu zo aangepast dat wordt gevraagd naar de gebruikersnaam en deze wordt gecontroleerd aan de hand van een vooraf gedefinieerde lijst met namen.

Eerst wordt de klasse Greeter bijgewerkt zodat hieraan nieuwe functionaliteit kan worden toegevoegd. Vervolgens wordt de toepassing bijgewerkt zodat deze de nieuwe functionaliteit kan gebruiken.

U kunt als volgt het bestand Greeter.as bijwerken:

1. Open het bestand Greeter.as.
2. Wijzig de inhoud van het bestand in het volgende (nieuwe en gewijzigde regels worden vet weergegeven):

```
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;
            }
        }
    }
}
```

De klasse Greeter bevat nu de volgende nieuwe functies:

- De array `validNames` bevat een lijst met geldige gebruikersnamen. Wanneer de klasse Greeter wordt geladen, wordt de array geïnitialiseerd als een lijst met drie namen.

- De methode `sayHello()` gebruikt nu bepaalde voorwaarden om een gebruikersnaam te accepteren en de begroeting te wijzigen. Wanneer `userName` een lege tekenreeks is (`""`), wordt de eigenschap `greeting` zo ingesteld dat de gebruiker wordt gevraagd een naam op te geven. Wanneer de gebruikernaam geldig is, wordt de begroeting `'Hello, userName'`. Wanneer niet aan een van bovenstaande voorwaarden wordt voldaan, wordt de variabele `greeting` ingesteld op `'Sorry, userName, you are not on the list.'`
- De methode `validName()` retourneert `true` wanneer `inputName` wordt gevonden in de array `validNames` en `false` wanneer deze niet wordt gevonden. De instructie `validNames.indexOf(inputName)` vergelijkt elke tekenreeks in de array `validNames` met de tekenreeks `inputName`. De `Array.indexOf()`-methode geeft de indexpositie van de eerste instantie weer van een object in een array. Het geeft de waarde -1 weer als het object niet wordt gevonden in de array.

Vervolgens bewerkt u het toepassingsbestand waarin wordt verwezen naar deze ActionScript-klasse.

De toepassing aanpassen met Flash Professional:

1. Open het bestand HelloWorld.fla.
2. Wijzig het script in frame 1 als volgt, zodat een lege tekenreeks (`""`) wordt doorgegeven aan de methode `sayHello()` van de klasse Greeter:

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

3. Selecteer het gereedschap Tekst in het gereedschapspalet. Maak twee nieuwe tekstvelden in het werkgebied. Plaats deze naast elkaar direct onder het bestaande tekstveld `mainText`.
4. In het eerste nieuwe tekstveld, dit is het label, vult u de tekst **Gebruikernaam:** in.
5. Selecteer het andere nieuwe tekstveld en selecteer Input Text in Eigenschapcontrole als het type tekstveld. Selecteer Enkele regel als regeltype. Typ **textIn** als instantienaam.
6. Klik op het eerste frame in de hoofdtijdlijn.
7. Voeg in het deelvenster Handelingen de volgende regels toe aan het einde van het bestaande script:

```
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);
    }
}
```

De nieuwe code voegt de volgende functionaliteit toe:

- De eerste twee regels definiëren eenvoudig randen voor twee tekstvelden.
- Een invoertekstveld zoals het veld `textIn` heeft een set gebeurtenissen die het kan verzenden. Met de methode `addEventListener()` kunt u een functie definiëren die wordt uitgevoerd wanneer een bepaald type gebeurtenis plaatsvindt. In dit geval is die gebeurtenis het drukken op een toets op het toetsenbord.

- De aangepaste functie `keyPressed()` controleert of de ingedrukte toets de Enter-toets is. Als dat het geval is, wordt de methode `sayHello()` van het object `myGreeter` aangeroepen, waarbij de tekst uit het tekstveld `textIn` als parameter wordt doorgegeven. Deze methode retourneert een tekenreeks (de begroeting), op basis van de doorgegeven waarde. De geretourneerde tekenreeks wordt vervolgens toegewezen aan de eigenschap `text` van het tekstveld `mainText`.

Het volledige script voor frame 1 ziet er als volgt uit:

```
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 Sla het bestand op.

9 Selecteer Besturing > Film testen om de toepassing uit te voeren.

Wanneer u de toepassing uitvoert, wordt u nu gevraagd om een gebruikernaam in te voeren. Als deze geldig is (Sammy, Frank of Dean), geeft de toepassing het bevestigingsbericht 'hello' weer.

De toepassing aanpassen met Flash Builder:

1 Open het bestand HelloWorld.mxml.

2 Pas vervolgens de `<mx:TextArea>`-tag aan om aan de gebruiker aan te geven dat dit alleen voor weergave is bedoeld. Wijzig de achtergrondkleur in lichtgrijs en stel het attribuut `editable` in op `false`:

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

3 Voeg nu de volgende regels toe achter de afsluitingstag `<s:TextArea>`. Met deze regels maakt u een TextInput-component waarin de gebruiker een gebruikernaam kan invoeren:

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

Het attribuut `enter` bepaalt wat er gebeurt wanneer de gebruiker op de toets Enter drukt in het veld `userNameTxt`. In dit voorbeeld, stuurt de code de tekst in het veld naar de methode `Greeter.sayHello()`. De greeting in het veld `mainTxt` wordt gewijzigd.

Het HelloWorld.mxml-bestand ziet er als volgt uit:

```
<?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 Sla het gewijzigde bestand HelloWorld.mxml op. Selecteer Uitvoeren > HelloWorld uitvoeren om de toepassing uit te voeren.

Wanneer u de toepassing uitvoert, wordt u nu gevraagd om een gebruikernaam in te voeren. Als deze geldig is (Sammy, Frank of Dean), geeft de toepassing het bevestigingsbericht '`Hello, ` *userName*' weer.

Hoofdstuk 3: Taal en syntaxis van ActionScript

ActionScript 3.0 bestaat uit zowel de kerntaal van ActionScript als de API (Application Programming Interface) van Adobe Flash Platform. De kerntaal is dat ActionScript-gedeelte dat de syntaxis beschrijft van de taal en de gegevenstypen op het hoogste niveau. ActionScript 3.0 biedt programmatietoegang tot de uitvoerbestanden van Adobe Flash Platform: Adobe Flash Player en Adobe AIR.

Taal - overzicht

Objecten vormen de kern en fundamentele bouwstenen van de ActionScript 3.0-taal. Elke variabele die u declareert, elke functie die u schrijft en elke instantie van een klasse die u maakt, is een object. U kunt een ActionScript 3.0-programma zien als een groep objecten voor het uitvoeren van taken, reageren op gebeurtenissen en communiceren met anderen.

Programmeurs die bekend zijn met objectgeoriënteerd programmeren (OOP) in Java of C++, kunnen objecten beschouwen als modules met twee soorten leden: gegevens opgeslagen in lidvariabelen of eigenschappen, en gedrag dat via methoden toegankelijk is. ActionScript 3.0 definieert objecten op soortgelijke, maar enigszins verschillende manier. In ActionScript 3.0 zijn objecten niets anders dan verzamelingen van eigenschappen. Deze eigenschappen zijn containers die niet alleen gegevens, maar ook functies of andere objecten kunnen bevatten. Als een functie op deze wijze aan een object is gekoppeld, is er sprake van een methode.

Hoewel de definitie in ActionScript 3.0 voor programmeurs met ervaring in Java of C++ wat vreemd kan overkomen, lijkt het definiëren van objecttypen met ActionScript 3.0-klassen heel sterk op het definiëren van klassen in Java of C++. Het onderscheid tussen de twee definities van object is van belang bij de bespreking van het ActionScript-objectmodel en andere geavanceerde onderwerpen, maar in de meeste andere situaties betekent de term *eigenschappen* klassenlidvariabelen in tegenstelling tot methoden. In de Naslaggids voor ActionScript 3.0 voor het Adobe Flash-platform wordt bijvoorbeeld de term *eigenschappen* gebruikt voor variabelen of voor de eigenschappen getter/setter. De term *methoden* betekent in dit geval functies die onderdeel zijn van een klasse.

Een subtiel verschil tussen klassen in ActionScript en klassen in Java of C++ is dat in ActionScript klassen niet alleen maar abstracte entiteiten zijn. ActionScript-klassen worden vertegenwoordigd door *klassenobjecten* die de eigenschappen en methoden van de klasse opslaan. Dit maakt technieken mogelijk die vreemd lijken voor programmeurs in Java en C++, zoals het opnemen van instructies of uitvoerbare code op het hoofdniveau van een klasse of pakket.

Nog een verschil tussen klassen in ActionScript en Java of C++ is dat elke ActionScript-klasse een zogeheten *prototypeobject* heeft. In eerdere versies van ActionScript vormden prototypeobjecten (aan elkaar gekoppeld in *prototypeketens*) samen de basis van de gehele klassenovererving shiërarchie. In ActionScript 3.0 spelen prototypeobjecten echter maar een kleine rol in het systeem van overerving. Het prototypeobject kan nog wel steeds nuttig zijn als alternatief voor statische eigenschappen en methoden als u een eigenschap en de waarde ervan wilt delen voor alle instanties van een klasse.

Voorheen konden gevorderde ActionScript-programmeurs de prototypeketen rechtstreeks manipuleren met speciale ingebouwde taalelementen. Nu de taal een krachtigere, op klassen gebaseerde programmeerinterface biedt, maken vele van deze speciale taalelementen, zoals `__proto__` en `__resolve`, geen deel meer uit van de taal. Bovendien zijn de optimalisaties van het interne overervingsmechanisme aanzienlijk verbeterd en is directe toegang tot het overervingsmechanisme daardoor uitgesloten.

Objecten en klassen

In ActionScript 3.0 wordt elk object gedefinieerd door een klasse. Een klasse is een soort sjabloon of ontwerp voor een type object. Klassendefinities kunnen variabelen en constanten bevatten, die weer gegevenswaarden bevatten, plus methoden, ofwel functies waarin gedrag is ondergebracht dat tot de klasse behoort. De opgeslagen waarden in eigenschappen kunnen *primitieve waarden* of andere objecten zijn. Primitieve waarden zijn getallen, tekenreeksen of Booleaanse waarden.

ActionScript bevat diverse ingebouwde klassen die deel uitmaken van de kerntaal. Sommige van deze ingebouwde klassen, zoals Number, Boolean en String, vertegenwoordigen de primitieve waarden die beschikbaar zijn in ActionScript. Andere klassen, zoals Array, Math en XML, definiëren complexere objecten.

Alle klassen, zowel ingebouwd als door de gebruiker gedefinieerd, zijn afgeleid van de klasse Object. Voor programmeurs die al ervaring hebben met ActionScript, is het van belang op te merken dat het gegevenstype Object niet meer het standaardgegevenstype is, ook al zijn alle klassen er nog wel van afgeleid. In ActionScript 2.0 kwamen de volgende twee regels code overeen, omdat bij het ontbreken van een typeannotatie een variabele het type Object kreeg toegewezen:

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

In ActionScript 3.0 wordt echter het concept van variabelen zonder type geïntroduceerd, die op de volgende manieren kunnen worden toegewezen:

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

Een variabele zonder type is niet hetzelfde als een variabele van het type Object. Het wezenlijke verschil is dat variabelen zonder type de speciale waarde `undefined` kunnen bevatten en een variabele van het type Object die waarde niet kan bevatten.

U kunt uw eigen klassen definiëren met behulp van het trefwoord `class`. U kunt klasseneigenschappen op drie manieren declareren: constanten kunt u definiëren met het trefwoord `const`, variabelen met het trefwoord `var` en eigenschappen getter and setter met de attributen `get` en `set` in een methodedeclaratie. U kunt methoden declareren met het trefwoord `function`.

U kunt een instantie van een klasse maken met de operator `new`. In het volgende voorbeeld wordt een instantie van de klasse Date gemaakt met de naam `myBirthday`.

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

Pakketten en naamruimten

Pakketten en naamruimten zijn verwante concepten. Met pakketten kunt u klassendefinities zo bundelen dat het delen van code wordt vergemakkelijkt en naamgevingsconflicten worden beperkt. Met naamruimten kunt u de zichtbaarheid van id's, zoals de naam van eigenschappen en methoden, beheren. U kunt naamruimten toepassen op code die zich binnen of buiten een pakket bevindt. Met pakketten kunt u uw klassenbestanden ordenen en met naamruimten de zichtbaarheid van afzonderlijke eigenschappen en methoden beheren.

Pakketten

Pakketten in ActionScript 3.0 worden geïmplementeerd met naamruimten, maar ze zijn geen synoniem van elkaar. Wanneer u een pakket declareert, maakt u impliciet een speciaal type naamruimte die gegarandeerd bekend is bij compilatie. Expliciet gemaakte naamruimten zijn niet noodzakelijkerwijs bekend bij compilatie.

In het volgende voorbeeld wordt de aanwijzing `package` gebruikt om een eenvoudig pakket met één klasse te maken:

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

De naam van de klasse in dit voorbeeld is SampleCode. De klasse bevindt zich binnen het pakket samples, zodat de compiler de klassennaam bij compilatie automatisch kwalificeert in de volledig gekwalificeerde naam: samples.SampleCode. De compiler kwalificeert ook de namen van aanwezige eigenschappen of methoden, zodat `sampleGreeting` en `sampleFunction()` resulteren in respectievelijk `samples.SampleCode.sampleGreeting` en `samples.SampleCode.sampleFunction()`.

Veel ontwikkelaars, vooral diegenen met programmeerervaring in Java, kiezen er mogelijk voor alleen klassen op het hoofdniveau van een pakket te plaatsen. ActionScript 3.0 biedt echter niet alleen ondersteuning voor klassen op het hoofdniveau van een pakket, maar ook voor variabelen, functies en zelfs instructies. Gevorderde gebruikers kunnen zo bijvoorbeeld een naamruimte op het hoofdniveau van een pakket definiëren, zodat deze beschikbaar is voor alle klassen in dat pakket. Er zijn echter maar twee toegangsspecificaties, `public` en `internal`, toegestaan op het hoofdniveau van een pakket. In Java kunt u geneste klassen als private declareren. ActionScript 3.0 biedt echter geen ondersteuning voor geneste klassen en ook niet voor klassen van het type private.

In vele andere opzichten zijn pakketten in ActionScript 3.0 te vergelijken met pakketten in de programmeertaal Java. Zoals u in het vorige voorbeeld ziet, worden verwijzingen naar volledig gekwalificeerde pakketten uitgedrukt met behulp van de puntoperator (.), net als in Java. Met pakketten kunt u code ordenen in een intuïtieve hiërarchische structuur voor gebruik door andere programmeurs. Dit vergemakkelijkt het delen van code doordat u zelf pakketten kunt maken om met anderen te delen en pakketten van anderen in uw code kunt gebruiken.

Door pakketten te gebruiken weet u ook zeker dat de id-namen die u gebruikt, uniek zijn en geen conflicten opleveren met andere id-namen. Volgens sommigen is dat zelfs het belangrijkste voordeel van pakketten. Twee programmeurs die bijvoorbeeld hun code met elkaar willen delen, hebben een klasse gemaakt met de naam SampleCode. Zonder pakketten zou dit resulteren in een naamconflict en moet een van de klassen een andere naam krijgen. Met pakketten kan het naamconflict gemakkelijk worden voorkomen door een van de klassen, of liever nog beide, in pakketten met unieke namen te plaatsen.

U kunt ook ingesloten punten in de pakketnaam opnemen om geneste pakketten te maken. U kunt pakketten zo hiërarchisch ordenen. Een goed voorbeeld hiervan is het pakket flash.display dat bij ActionScript 3.0 wordt geleverd. Het pakket flash.display bevindt zich in het pakket flash.

ActionScript 3.0 is voornamelijk georganiseerd binnen het pakket flash. Het pakket flash.display bevat bijvoorbeeld de weergaveoverzicht-API en het pakket flash.events bevat het nieuwe gebeurtenismodel.

Pakketten maken

ActionScript 3.0 biedt aanzienlijke flexibiliteit voor de manier waarop u pakketten, klassen en bronbestanden ordent. In eerdere versies van ActionScript was maar één klasse per bronbestand mogelijk en moest de naam van het bronbestand overeenkomen met de naam van de klasse. In ActionScript 3.0 kunt u meerdere klassen in één bronbestand opnemen, maar slechts één klasse in elk bestand kan beschikbaar worden gemaakt voor code die buiten dat bestand valt. Met andere woorden, maar één klasse in elk bestand kan worden gedeclareerd in een pakketdeclaratie. U moet andere klassen buiten de pakketdefinitie declareren, hetgeen die klassen onzichtbaar maakt voor code buiten dat bronbestand. De naam van de gedeclareerde klasse binnen de pakketdefinitie moet overeenkomen met de naam van het bronbestand.

ActionScript 3.0 biedt eveneens meer flexibiliteit voor de manier waarop u pakketten declareert. In eerdere versies van ActionScript waren pakketten niet meer dan mappen waarin u bronbestanden kon plaatsen. U declareerde geen pakketten met de instructie `package`, maar nam de naam van het pakket op als onderdeel van de volledig gekwalificeerde klassennaam in de klassendeclaratie. Hoewel pakketten nog steeds als mappen worden gebruikt in ActionScript 3.0, kunnen ze meer dan alleen klassen bevatten. In ActionScript 3.0 gebruikt u de instructie `package` om een pakket te declareren. Dit betekent dat u ook variabelen, functies en naamruimten op het hoofdniveau van een pakket kunt declareren. U kunt zelfs uitvoerbare instructies opnemen op het hoofdniveau van een pakket. Als u variabelen, functies of naamruimten declareert op het hoofdniveau van een pakket, zijn de enige beschikbare attributen op dat niveau `public` en `internal`. Ook kan maar voor één declaratie op pakketniveau per bestand het attribuut `public` worden gebruikt, of de declaratie nu een klasse, variabele, functie of naamruimte betreft.

Pakketten zijn nuttig voor het ordenen van code en het vermijden van naamconflicten. U moet het concept van pakketten niet verwarren met het niet-gerelateerde concept van klassenovererving. Twee klassen in hetzelfde pakket hebben een gemeenschappelijke naamruimte, maar hoeven in geen enkel ander opzicht aan elkaar te zijn gerelateerd. Zo hoeft een genest pakket ook geen semantische relatie te hebben met het bovenliggende pakket.

Pakketten importeren

Als u een klasse binnen een pakket wilt gebruiken, moet u het pakket of de specifieke klasse importeren. In ActionScript 2.0 was het importeren van klassen optioneel.

Bekijk bijvoorbeeld het voorbeeld van de klasse SampleCode hierboven. Als de klasse zich in een pakket met de naam samples bevindt, moet u een van de volgende instructies import gebruiken voordat u de klasse SampleCode gebruikt:

```
import samples.*;
```

of

```
import samples.SampleCode;
```

In het algemeen geldt dat instructies `import` zo specifiek mogelijk moeten zijn. Als u alleen de klasse SampleCode uit het pakket samples wilt gebruiken, importeert u alleen de klasse SampleCode en niet het gehele pakket waartoe de klasse behoort. Het importeren van hele pakketten kan resulteren in onverwachte naamconflicten.

U moet tevens de broncode plaatsen waarmee het pakket of de klasse in het *klassenpad* wordt gedefinieerd. Het klassenpad is een door de gebruiker gedefinieerde lijst met lokale mappaden op basis waarvan de compiler zoekt naar geïmporteerde pakketten en klassen. Het klassenpad wordt ook wel *bouwpad* of *bronpad* genoemd.

Nadat u de klasse of het pakket correct hebt geïmporteerd, kunt u de volledig gekwalificeerde naam van de klasse gebruiken (samples.SampleCode) of alleen de klassennaam zelf (SampleCode).

Volledig gekwalificeerde namen zijn nuttig wanneer klassen, methoden of eigenschappen met een identieke naam resulteren in dubbelzinnige code. Ze kunnen echter moeilijk te beheren zijn als u ze voor alle id's gebruikt. Het gebruik van de volledig gekwalificeerde naam leidt bijvoorbeeld tot nogal uitgebreide code wanneer u een instantie van de klasse SampleCode instantieert:

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

Naarmate het aantal niveaus van geneste pakketten toeneemt, neemt de leesbaarheid van de code af. In situaties waarin u zeker weet dat dubbelzinnige id's geen probleem vormen, kunt u de code leesbaarder maken door eenvoudige id's te gebruiken. Het gebruik van alleen de klasse-id levert bijvoorbeeld veel minder uitgebreide code op wanneer u een nieuwe instantie van de klasse SampleCode instantieert:

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

Als u probeert id-namen te gebruiken zonder eerst het desbetreffende pakket of de klasse te importeren, kan de compiler de klassendefinities niet vinden. Aan de andere kant geldt dat wanneer u een pakket of klasse importeert en probeert een naam te definiëren die conflicteert met een geïmporteerde naam, een fout wordt gegenereerd.

Bij het maken van een pakket is de standaardtoegangsspecificatie voor alle leden van dat pakket `internal`. Dit betekent dat pakketleden standaard alleen zichtbaar zijn voor andere leden van het pakket. Als u wilt dat een klasse beschikbaar is voor code buiten het pakket, moet u de klasse als `public` declareren. Het volgende pakket bevat bijvoorbeeld twee klassen, SampleCode en CodeFormatter:

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

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

De klasse SampleCode is zichtbaar buiten het pakket, omdat deze klasse als `public` is gedeclareerd. De klasse CodeFormatter is echter alleen zichtbaar binnen het pakket samples zelf. Als u probeert toegang te krijgen tot de klasse CodeFormatter buiten het pakket samples, wordt een fout gegenereerd, zoals in het volgende voorbeeld wordt getoond:

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

Als u wilt dat beide klassen beschikbaar zijn buiten het pakket, moet u beide klassen als `public` declareren. U kunt het attribuut `public` niet toepassen op de pakketdeclaratie.

Volledig gekwalificeerde namen zijn nuttig voor het oplossen van naamconflicten die bij het gebruik van pakketten kunnen optreden. Dit scenario kan zich voordoen als u twee pakketten importeert die klassen met dezelfde id definiëren. Het volgende pakket bevat bijvoorbeeld ook een klasse met de naam SampleCode:

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

Als u beide klassen als volgt importeert, is er sprake van een naamconflict bij het gebruiken van de klasse SampleCode:

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

Voor de compiler is niet duidelijk welke klasse SampleCode moet worden gebruikt. U kunt dit conflict oplossen door de volledig gekwalificeerde naam van elke klasse te gebruiken, als volgt:

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

Opmerking: *Programmeurs met ervaring in C++ verwarren de instructie* `import` *vaak met* `#include`*. De aanwijzing* `#include` *is nodig in C++, omdat compilers voor C++ één bestand tegelijk verwerken. Andere bestanden met klassendefinities worden buiten beschouwing gelaten, tenzij een headerbestand expliciet wordt opgenomen. ActionScript 3.0 heeft een aanwijzing* `include`*, maar die is niet bedoeld om klassen en pakketten te importeren. Voor het importeren van klassen of pakketten in ActionScript 3.0 moet u de instructie* `import` *gebruiken en het bronbestand met het pakket in het klassenpad plaatsen.*

Naamruimten

Met naamruimten kunt u de zichtbaarheid beheren van de eigenschappen en methoden die u maakt. U kunt de toegangsbeheerspecificaties `public`, `private`, `protected` en `internal` zien als ingebouwde naamruimten. Als deze vooraf gedefinieerde toegangsbeheerspecificaties niet voldoen aan uw behoeften, kunt u zelf naamruimten maken.

Als u ervaring hebt met XML-naamruimten, zal het meeste u bekend voorkomen. De syntaxis en details voor de implementatie van ActionScript verschillen echter iets van die van XML. Als u nog nooit met naamruimten hebt gewerkt, zult u merken dat het concept zelf eenvoudig is, maar dat bij de implementatie specifieke terminologie wordt gebruikt die u moet leren kennen.

Als u wilt begrijpen hoe naamruimten werken, is het nuttig om te weten dat de naam van een eigenschap of methode altijd twee gedeelten bevat: een id en een naamruimte. De id is wat gewoonlijk als naam wordt beschouwd. De id's in de volgende klassendefinitie zijn bijvoorbeeld `sampleGreeting` en `sampleFunction()`:

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

Wanneer definities niet door een naamruimte-attribuut worden voorafgegaan, worden de namen ervan gekwalificeerd door de standaardnaamruimte `internal`. Dit betekent dat ze alleen zichtbaar zijn voor aanroepende elementen in hetzelfde pakket. Als de compiler is ingesteld op de strikte modus, wordt een waarschuwing weergegeven dat de naamruimte `internal` van toepassing is op alle id's zonder naamruimte-attribuut. Als u een id overal beschikbaar wilt maken, moet u de id-naam specifiek vooraf laten gaan door het attribuut `public`. In de voorgaande voorbeeldcode hebben zowel `sampleGreeting` als `sampleFunction()` de naamruimtewaarde `internal`.

Er zijn drie basisstappen voor het gebruiken van naamruimten. Als eerste stap definieert u de naamruimte met behulp van het trefwoord `namespace`. Met de volgende code wordt bijvoorbeeld de naamruimte `version1` gedefinieerd:

```
namespace version1;
```

Als tweede stap past u de naamruimte toe door deze te gebruiken in plaats van een toegangsbeheerspecificatie in een eigenschap- of methodedeclaratie. In het volgende voorbeeld wordt een functie met de naam `myFunction()` in de naamruimte `version1` geplaatst:

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

Als derde stap, nadat u de naamruimte hebt toegepast, kunt u ernaar verwijzen met de aanwijzing `use` of door de naam van een id met een naamruimte te kwalificeren. In het volgende voorbeeld wordt naar een functie met de naam `myFunction()` verwezen met behulp van de aanwijzing `use`:

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

U kunt ook een gekwalificeerde naam gebruiken om te verwijzen naar de functie `myFunction()`, zoals in het volgende voorbeeld wordt getoond:

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

Naamruimten definiëren

Naamruimten bevatten één waarde, de URI (Uniform Resource Identifier), ook wel de *naamruimtenaam* genoemd. Met een URI weet u zeker dat de naamruimtedefinitie uniek is.

U maakt een naamruimte door op een van de twee mogelijke manieren een naamruimtedefinitie te declareren. U kunt een naamruimte definiëren met een expliciete URI, zoals bij een XML-naamruimte, of u kunt de URI weglaten. Het volgende voorbeeld laat zien hoe u een naamruimte kunt definiëren met een URI:

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

De URI dient als unieke id-tekenreeks voor de naamruimte. Als u de URI weglaat, zoals in het volgende voorbeeld, wordt door de compiler een unieke interne id-tekenreeks gemaakt in plaats van de URI. U hebt geen toegang tot deze interne id-tekenreeks.

```
namespace flash_proxy;
```

Nadat u een naamruimte hebt gedefinieerd, met of zonder URI, kunt u die naamruimte niet opnieuw definiëren in hetzelfde bereik. Een poging om een naamruimte te definiëren die al in hetzelfde bereik is gedefinieerd, resulteert in een compilatiefout.

Als een naamruimte is gedefinieerd binnen een pakket of een klasse, is de naamruimte mogelijk niet zichtbaar voor code buiten dat pakket of de klasse, tenzij de juiste toegangsbeheerspecificatie wordt gebruikt. Met de volgende code wordt bijvoorbeeld de naamruimte `flash_proxy` getoond, gedefinieerd binnen het pakket flash.utils. In het voorbeeld betekent het ontbreken van een toegangsbeheerspecificatie dat de naamruimte `flash_proxy` alleen zichtbaar is voor code binnen het pakket flash.utils en niet voor andere code buiten het pakket:

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

Met de volgende code wordt het attribuut `public` gebruikt om de naamruimte `flash_proxy` zichtbaar te maken voor code buiten het pakket:

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

Naamruimten toepassen

Een naamruimte toepassen betekent een definitie plaatsen in een naamruimte. Definities die u in naamruimten kunt plaatsen zijn functies, variabelen en constanten (u kunt geen klasse in een aangepaste naamruimte plaatsen).

Neem bijvoorbeeld een functie die is gedeclareerd met de toegangsbeheernaamruimte `public`. Door het gebruik van het attribuut `public` in een functiedefinitie wordt de functie in de naamruimte public geplaatst en is zo beschikbaar voor alle code. Nadat u een naamruimte hebt gedefinieerd, kunt u de gedefinieerde naamruimte net zo gebruiken als het attribuut `public`. De definitie is dan beschikbaar voor code die naar de aangepaste naamruimte kan verwijzen. Als u bijvoorbeeld een naamruimte `example1` definieert, kunt u een methode met de naam `myFunction()` toevoegen door `example1` als attribuut te gebruiken, zoals in het volgende voorbeeld wordt getoond:

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

Door de methode `myFunction()` te declareren met de naamruimte `example1` als attribuut, behoort de methode tot de naamruimte `example1`.

Let op het volgende wanneer u naamruimten toepast:

- U kunt slechts één naamruimte toepassen op elke declaratie.
- U kunt een naamruimte-attribuut nooit op meer dan één definitie tegelijk toepassen. Met andere woorden, als u een naamruimte wilt toepassen op tien verschillende functies, moet u de naamruimte als attribuut toevoegen aan elk van de tien functiedefinities.
- Als u een naamruimte toepast, kunt u geen toegangsbeheerspecificatie opgeven, omdat naamruimten en toegangsbeheerspecificaties elkaar uitsluiten. Met andere woorden, u kunt een functie of eigenschap niet declareren als `public`, `private`, `protected` of `internal` nadat u de naamruimte hebt toegepast.

Verwijzen naar naamruimten

U hoeft niet expliciet naar een naamruimte te verwijzen wanneer u een methode of eigenschap gebruikt die wordt gedeclareerd met een van de toegangsbeheernaamruimten, zoals `public`, `private`, `protected` en `internal`. Dat komt doordat de toegang tot deze speciale naamruimten wordt bepaald door de context. Definities die bijvoorbeeld in de naamruimte `private` worden geplaatst, zijn automatisch beschikbaar voor code binnen dezelfde klasse. Voor naamruimten die u zelf definieert, bestaat een dergelijke contextgevoeligheid echter niet. Als u een methode of eigenschap wilt gebruiken die u in een aangepaste naamruimte hebt geplaatst, moet u naar de naamruimte verwijzen.

U kunt naar naamruimten verwijzen met de aanwijzing `use namespace` of u kunt de naam met de naamruimte kwalificeren met behulp van het naamkwalificatieteken (`::`). Door naar een naamruimte te verwijzen met de aanwijzing `use namespace` wordt de naamruimte 'geopend'. Zo kan de naamruimte worden toegepast op alle id's die niet zijn gekwalificeerd. Als u bijvoorbeeld de naamruimte `example1` hebt gedefinieerd, hebt u toegang tot namen in die naamruimte door `use namespace example1` te gebruiken:

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

U kunt meerdere naamruimten tegelijk openen. Nadat u een naamruimte hebt geopend met `use namespace`, blijft deze open voor het blok met code waarin de naamruimte is geopend. U kunt een naamruimte niet expliciet sluiten.

Als u meer dan één naamruimte open hebt, neemt de kans op naamconflicten echter toe. Als u een naamruimte liever niet wilt openen, kunt u de aanwijzing `use namespace` vermijden door de naam van de methode of eigenschap met de naamruimte en het naamkwalificatieteken te kwalificeren. Met de volgende code wordt bijvoorbeeld getoond hoe u de naam `myFunction()` met de naamruimte `example1` kunt kwalificeren:

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

Naamruimten gebruiken

U vindt een praktijkvoorbeeld van een naamruimte die wordt gebruikt om naamconflicten te voorkomen in de klasse flash.utils.Proxy die deel uitmaakt van ActionScript 3.0. Met de klasse Proxy, die de vervanging is van de eigenschap `Object.__resolve` eigenschap in ActionScript 2.0, waarmee u verwijzingen naar niet-gedefinieerde eigenschappen of methoden onderscheppen voordat een fout optreedt. Alle methoden van de klasse Proxy bevinden zich in de naamruimte `flash_proxy` om naamconflicten te voorkomen.

Voor een beter begrip van hoe de naamruimte `flash_proxy` wordt gebruikt, moet u weten hoe u de klasse Proxy gebruikt. De functionaliteit van de klasse Proxy is alleen beschikbaar voor klassen die ervan overerven. Met andere woorden, als u de methoden van de klasse Proxy voor een ander object wilt gebruiken, moet de klassendefinitie van het object de klasse Proxy uitbreiden. Als u bijvoorbeeld pogingen om een ongedefinieerde methode aan te roepen wilt onderscheppen, breidt u de klasse Proxy uit en overschrijft u vervolgens de methode `callProperty()` van de klasse Proxy.

Zoals eerder is besproken, bestaat het implementeren van naamruimten doorgaans uit drie stappen: definiëren, toepassen en ernaar verwijzen. Omdat u de methoden van de klasse Proxy echter nooit expliciet aanroept, wordt de naamruimte `flash_proxy` alleen gedefinieerd en toegepast, maar wordt er niet naar verwezen. ActionScript 3.0 definieert de naamruimte `flash_proxy` en past deze toe in de klasse Proxy. De code hoeft de naamruimte `flash_proxy` alleen toe te passen op klassen die de klasse Proxy uitbreiden.

De naamruimte `flash_proxy` wordt ongeveer als volgt gedefinieerd in het pakket flash.utils:

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

De naamruimte wordt toegepast op de methoden van de klasse Proxy zoals in het volgende fragment van de klasse Proxy wordt getoond:

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

Zoals de volgende code laat zien, moet u eerst de klasse Proxy en de naamruimte `flash_proxy` importeren. Vervolgens moet u de klasse zo declareren dat deze de klasse Proxy uitbreidt (u moet ook het attribuut `dynamic` toevoegen als u in de strikte modus compileert). Wanneer u de methode `callProperty()` overschrijft, moet u de naamruimte `flash_proxy` gebruiken.

```
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);
        }
    }
}
```

Als u een instantie van de klasse MyProxy maakt en een ongedefinieerde methode aanroept, zoals de aangeroepen methode `testing()` in het volgende voorbeeld, onderschept het object Proxy de methodeaanroep en worden de instructies binnen de overschreven methode `callProperty()` uitgevoerd (in dit geval een eenvoudige instructie `trace()`).

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

Er zijn twee voordelen verbonden aan het opnemen van de klasse Proxy binnen de naamruimte `flash_proxy`. Ten eerste vermindert een aparte naamruimte de hoeveelheid rommelige code in de openbare interface van de klassen die de klasse Proxy uitbreiden. (Er zijn meer dan tien methoden in de klasse Proxy die u kunt overschrijven en die u niet rechtstreeks kunt aanroepen. Het plaatsen van al deze methoden in de naamruimte public zou tot verwarring kunnen leiden.) Ten tweede voorkomt het gebruik van de naamruimte `flash_proxy` naamconflicten in het geval dat de subklasse Proxy instantiemethoden bevat met namen die overeenkomen met een van de methoden van de klasse Proxy. U wilt bijvoorbeeld een van uw eigen methoden `callProperty()` noemen. De volgende code is acceptabel, omdat uw versie van de methode `callProperty()` zich in een andere naamruimte bevindt:

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

Naamruimten kunnen ook nuttig zijn wanneer u voor toegang wilt zorgen tot methoden of eigenschappen op een manier die niet mogelijk is met de vier toegangsbeheerspecificaties (`public`, `private`, `internal` en `protected`). U hebt bijvoorbeeld enkele hulpprogrammamethoden verspreid over diverse pakketten. U wilt deze methoden beschikbaar maken voor alle pakketten, maar ze niet als public declareren. U kunt voor dit doel een naamruimte maken en gebruiken als uw eigen speciale toegangsbeheerspecificatie.

In het volgende voorbeeld wordt een door de gebruiker gedefinieerde naamruimte gebruikt om twee functies in verschillende pakketten te groeperen. Door de twee functies in dezelfde naamruimte te groeperen, kunt u ze beide zichtbaar maken voor een klasse of pakket met behulp van een instructie `use namespace`.

In dit voorbeeld worden vier bestanden gebruikt om deze techniek te demonstreren. Al deze bestanden moeten in uw klassenpad zijn opgenomen. Met het eerste bestand, myInternal.as, wordt de naamruimte `myInternal` gedefinieerd. Het bestand bevindt zich in een pakket met de naam example, zodat u het bestand in een map met de naam example moet plaatsen. De naamruimte is gemarkeerd als `public`, zodat deze in andere pakketten kan worden geïmporteerd.

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

Het tweede en derde bestand, Utility.as en Helper.as, definiëren de klassen met de methoden die beschikbaar moeten zijn voor andere pakketten. De klasse Utility bevindt zich in het pakket example.alpha. Dit betekent dat het bestand in een map moet worden geplaatst met de naam alpha die een submap is van de map example. De klasse Helper bevindt zich in het pakket example.beta. Dit betekent dat het bestand in een map moet worden geplaatst met de naam beta die eveneens een submap is van de map example. Beide pakketten, example.alpha en example.beta, moeten de naamruimte importeren voordat deze wordt gebruikt.

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

Het vierde bestand, NamespaceUseCase.as, is de hoofdtoepassingsklasse en moet op hetzelfde niveau zijn als de map example. In Flash Professional zou deze klasse worden gebruikt als de documentklasse voor het FLA-bestand. De klasse NamespaceUseCase importeert bovendien de naamruimte `myInternal` en gebruikt deze om de twee statische methoden op te roepen die zich in de andere pakketten bevinden. In het voorbeeld worden alleen statische methoden gebruikt om de code eenvoudig te houden. U kunt zowel statische methoden als instantiemethoden in de naamruimte `myInternal` plaatsen.

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

Variabelen

Met behulp van variabelen kunt u waarden opslaan die u in uw programma gebruikt. Voor het declareren van een variabele moet u de instructie `var` gebruiken met de variabelenaam. In ActionScript 3.0 is het gebruik van de instructie `var` altijd vereist. Met de volgende regel ActionScript wordt bijvoorbeeld een variabele gedeclareerd met de naam `i`:

```
var i;
```

Als u de instructie `var` weglaat bij het declareren van een variabele, krijgt u een compilatiefout in de strikte modus en een uitvoeringsfout in de standaardmodus. De volgende coderegel resulteert bijvoorbeeld in een fout als de variabele `i` niet eerder is gedefinieerd:

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

Als u een variabele wilt koppelen aan een gegevenstype, moet u dit doen wanneer u de variabele declareert. U kunt een variabele declareren zonder het type op te geven, maar dit zorgt voor een compilerwaarschuwing in de strikte modus. U geeft het type van een variabele op door achter de variabelenaam een dubbele punt (:) te plaatsen, gevolgd door het type van de variabele. Met de volgende coderegel wordt bijvoorbeeld een variabele `i` gedeclareerd van het type int:

```
var i:int;
```

U kunt een waarde aan een variabele toewijzen met behulp van de toewijzingsoperator (=). Met de volgende coderegel wordt bijvoorbeeld een variabele `i` gedeclareerd en wordt er de waarde 20 aan toegewezen:

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

Het kan handig zijn een waarde aan een variabele toe te wijzen wanneer u de variabele declareert, zoals in het volgende voorbeeld:

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

De techniek om een waarde aan een variabele toe te wijzen wanneer deze wordt gedeclareerd, wordt vaak gebruikt bij het toewijzen van primitieve waarden zoals gehele getallen en tekenreeksen, maar ook bij het maken van een array of instantiëren van een instantie van een klasse. In het volgende voorbeeld wordt een array getoond waarvoor een waarde wordt gedeclareerd en toegewezen in één coderegel.

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

U kunt een instantie van een klasse maken met de operator `new`. In het volgende voorbeeld wordt een instantie van een klasse met de naam `CustomClass` gemaakt en wordt een verwijzing naar de nieuwe instantie van de klasse toegewezen aan de variabele met de naam `customItem`:

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

Als u meerdere variabelen wilt declareren, kunt u ze allemaal in één coderegel declareren met behulp van de kommaoperator (`,`) als scheidingsteken tussen de variabelen. Met de volgende code worden bijvoorbeeld drie variabelen gedeclareerd in één coderegel:

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

U kunt ook waarden toewijzen aan elk van de variabelen in dezelfde coderegel. Met de volgende code worden bijvoorbeeld drie variabelen gedeclareerd (`a`, `b` en `c`) en wordt aan elke variabele een waarde toegewezen:

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

Hoewel u de kommaoperator kunt gebruiken om declaraties van variabelen te groeperen in één instructie, komt dit de leesbaarheid van de code niet altijd ten goede.

Variabelenbereik

Het *bereik* van een variabele is het gebied van de code waar toegang tot de variabele mogelijk is door een lexicale verwijzing. Een *algemene* variabele wordt gedefinieerd in alle gebieden van de code, een *lokale* variabele wordt gedefinieerd in een deel van de code. In ActionScript 3.0 krijgen variabelen altijd het bereik toegewezen van de functie of klasse waarin ze worden gedeclareerd. Een algemene variabele wordt gedefinieerd buiten een functie- of klassendefinitie. Met de volgende code wordt bijvoorbeeld een algemene variabele `strGlobal` gemaakt door deze buiten een functie te declareren. Het voorbeeld toont dat een algemene variabele beschikbaar is zowel binnen als buiten de functiedefinitie.

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

U declareert een lokale variabele binnen een functiedefinitie. Het kleinste codegebied waarvoor u een lokale variabele kunt definiëren is een functiedefinitie. Een lokale variabele die binnen een functie wordt gedeclareerd, bestaat alleen in die functie. Wanneer u bijvoorbeeld een variabele declareert met de naam `str2` binnen een functie met de naam `localScope()`, is deze variabele niet beschikbaar buiten de functie.

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

Wanneer de variabelenaam die u gebruikt voor de lokale variabele al is gedeclareerd als een algemene variabele, verbergt (of arceert) de lokale definitie de algemene definitie wanneer de lokale variabele in het bereik valt. De algemene variabele blijft bestaan buiten de functie. Met de volgende code wordt bijvoorbeeld een algemene tekenreeksvariabele gemaakt met de naam `str1` en wordt vervolgens een lokale variabele met dezelfde naam gemaakt binnen de functie `scopeTest()`. De instructie `trace` geeft binnen de functie de lokale waarde van de variabele als uitvoer. Buiten de functie geeft de instructie `trace` de algemene waarde van de variabele als uitvoer.

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

Variabelen in ActionScript hebben geen bereik op blokniveau, wat in C++ en Java wel het geval is. Een codeblok is een groep instructies tussen een accolade openen ({) en een accolade sluiten (}). In sommige programmeertalen, zoals C++ en Java, zijn variabelen die binnen een codeblok worden gedeclareerd, niet beschikbaar buiten dat codeblok. Deze beperking in het bereik wordt bereik op blokniveau genoemd en komt in ActionScript niet voor. Als u een variabele binnen een codeblok declareert, is die variabele niet alleen beschikbaar in dat codeblok maar ook in alle andere delen van de functie waartoe het codeblok behoort. De volgende functie bevat bijvoorbeeld variabelen die worden gedefinieerd in diverse blokbereiken. Alle variabelen zijn beschikbaar in de gehele functie.

```
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"]);
```

Een interessant gevolg van het ontbreken van bereik op blokniveau is dat u een variabele kunt lezen of ernaar schrijven voordat deze wordt gedeclareerd (mits declaratie plaatsvindt voordat de functie eindigt). Dit komt door een techniek die *hoisting* wordt genoemd, wat betekent dat de compiler alle declaraties van variabelen naar het hoofdniveau van de functie verplaatst. De volgende code wordt bijvoorbeeld gecompileerd ook al treedt de eerste functie `trace()` voor de variabele `num` op voordat de variabele `num` wordt gedeclareerd:

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

In de compiler vindt echter geen hoisting plaats voor toewijzingsinstructies. Dit verklaart waarom de eerste `trace()` van `num` resulteert in `NaN` (Not a Number), wat de standaardwaarde is voor variabelen van het gegevenstype Number. Dit betekent dat u waarden kunt toewijzen aan variabelen nog voordat deze worden gedeclareerd, zoals in het volgende voorbeeld wordt getoond:

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

Standaardwaarden

Een *standaardwaarde* is de waarde die een variabele bevat voordat u de waarde ervan instelt. U *initialiseert* een variabele wanneer u voor de eerste keer de waarde ervan instelt. Wanneer u een variabele declareert maar niet de waarde ervan instelt, is de variabele *niet-geïnitialiseerd*. De waarde van een niet-geïnitialiseerde variabele is afhankelijk van het gegevenstype ervan. In de volgende tabel worden de standaardwaarden van variabelen beschreven, geordend op gegevenstype:

Gegevenstype	**Standaardwaarde**
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
Tekenreeks	`null`
uint	0
Niet gedeclareerd (komt overeen met typeannotatie `*`)	`undefined`
Alle andere klassen, inclusief door de gebruiker gedefinieerde klassen	`null`

Voor variabelen van het type Number is de standaardwaarde `NaN` (Not a Number). Dit is een speciale waarde gedefinieerd door de IEEE-754-standaard die een waarde anders dan een getal vertegenwoordigt.

Als u wel een variabele maar geen gegevenstype declareert, is het gegevenstype `*` (variabele zonder type) van toepassing. Als u een variabele zonder type ook niet initialiseert met een waarde, is de standaardwaarde `undefined`.

Voor gegevenstypen anders dan Boolean, Number, int en uint is de standaardwaarde voor een niet-geïnitialiseerde variabele `null`. Dit geldt voor alle klassen die door ActionScript 3.0 worden gedefinieerd plus de aangepaste klassen die u zelf maakt.

De waarde `null` is geen geldige waarde voor variabelen van het type Boolean, Number, int of uint. Wanneer u probeert de waarde `null` toe te wijzen aan een van deze variabelen, wordt de waarde omgezet in de standaardwaarde voor dat gegevenstype. U kunt de waarde `null` toewijzen aan variabelen van het type Object. Wanneer u probeert de waarde `undefined` toe te wijzen aan een variabele van het type Object, wordt de waarde omgezet in `null`.

Voor variabelen van het type Number bestaat een speciale functie op hoofdniveau met de naam `isNaN()` die de Booleaanse waarde `true` retourneert als de variabele geen getal is en `false` als dat wel het geval is.

Gegevenstypen

Een *gegevenstype* definieert een set waarden. Het gegevenstype Boolean is bijvoorbeeld de set met precies twee waarden: `true` en `false`. Naast het gegevenstype Boolean zijn in ActionScript 3.0 diverse gangbare gegevenstypen gedefinieerd, zoals String, Number en Array. U kunt uw eigen gegevenstypen definiëren door met behulp van klassen of interfaces een aangepaste set waarden te definiëren. Alle waarden in ActionScript 3.0, primitieve en complexe, zijn objecten.

Een *primitieve waarde* behoort tot een van de volgende gegevenstypen: Boolean, int, Number, String en uint. Het werken met primitieve waarden is doorgaans sneller dan het werken met complexe waarden, omdat in ActionScript primitieve waarden op een speciale manier worden opgeslagen, waarbij geheugen en snelheid worden geoptimaliseerd.

Opmerking: *Voor lezers met belangstelling voor de technische details: in ActionScript worden primitieve waarden intern opgeslagen als onveranderlijke objecten. Dit betekent dat het doorgeven als verwijzing hetzelfde effect heeft als doorgeven als waarde. Zo wordt het geheugengebruik minder en de snelheid van uitvoering hoger, omdat verwijzingen meestal aanzienlijk kleiner zijn dan de waarden zelf.*

Een *complexe waarde* is een waarde die geen primitieve waarde is. Gegevenstypen die sets complexe waarden definiëren, zijn Array, Date, Error, Function, RegExp, XML en XMLList.

In veel programmeertalen wordt onderscheid gemaakt tussen primitieve waarden en de bijbehorende omvattende objecten. Java heeft bijvoorbeeld een primitieve int en de klasse java.lang.Integer die deze omvat. Primitieven in Java zijn geen objecten, maar de omvattende objecten wel. Dit maakt primitieven handig voor sommige bewerkingen en omvattende objecten beter geschikt voor andere bewerkingen. In ActionScript 3.0 wordt uit praktische overwegingen geen onderscheid gemaakt tussen primitieve waarden en de omvattende objecten. Alle waarden, dus ook primitieve waarden, zijn objecten. Tijdens de uitvoering worden deze primitieve typen als speciale gevallen behandeld die zich als objecten gedragen, maar niet de normale overhead vereisen die gepaard gaat met het maken van objecten. Dit betekent dat de volgende twee regels code gelijkwaardig zijn:

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

Alle genoemde primitieve en complexe gegevenstypen worden gedefinieerd door de kernklassen van ActionScript 3.0. Met behulp van de kernklassen kunt u objecten maken en letterlijke waarden gebruiken in plaats van de operator `new` te gebruiken. U kunt bijvoorbeeld als volgt een array maken met een letterlijke waarde of de klassenconstructor Array:

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

Type controleren

Het type kan worden gecontroleerd bij compilatie of uitvoering. In statisch getypeerde talen, zoals C++ en Java, wordt het type gecontroleerd bij compilatie. In dynamisch getypeerde talen, zoals Smalltalk en Python, wordt het type gecontroleerd bij uitvoering. ActionScript 3.0 is een dynamisch getypeerde taal en controle van het type vindt dus bij uitvoering plaats. Controle van het type bij compilatie wordt echter ook ondersteund in een speciale compilermodus, *strikte modus* genoemd. In de strikte modus vindt controle van het type bij zowel compilatie als uitvoering plaats, in de standaardmodus alleen bij uitvoering.

Dynamisch getypeerde talen bieden een grote mate van flexibiliteit wanneer u code structureert. Daardoor kunnen bij uitvoering echter wel typeringsfouten optreden. Statisch getypeerde talen rapporteren typeringsfouten bij compilatie, maar dan moet de type-informatie bij compilatie al wel bekend zijn.

Type controleren bij compilatie

Type controleren bij compilatie geniet vaak de voorkeur bij grotere projecten. Flexibiliteit in de gegevenstypen is bij grotere projecten immers vaak van minder belang dan het zo vroeg mogelijk afvangen van typefouten. Daarom wordt de ActionScript-compiler in Flash Professional en Flash Builder standaard uitgevoerd in de strikte modus.

Adobe Flash Builder

U kunt de strikte modus in Flash Builder uitschakelen via de compileerinstellingen voor ActionScript in het dialoogvenster Project Properties (Projecteigenschappen).

Voor het controleren van het type bij compilatie moet in de compiler de informatie over gegevenstypen bekend zijn voor de variabelen of expressies in de code. U kunt een gegevenstype expliciet declareren voor een variabele door de operator dubbelepunt (`:`) toe te voegen, gevolgd door het gegevenstype als achtervoegsel van de variabelenaam. U kunt een gegevenstype aan een parameter koppelen door de operator dubbelepunt te gebruiken, gevolgd door het gegevenstype. Met de volgende code wordt bijvoorbeeld informatie over het gegevenstype toegevoegd aan de parameter `xParam` en wordt een variabele `myParam` gedeclareerd met een expliciet gegevenstype:

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

In de strikte modus rapporteert de ActionScript-compiler niet-overeenkomende typen als compilatiefouten. Met de volgende code wordt bijvoorbeeld een functieparameter `xParam` gedeclareerd van het type Object. Verderop wordt geprobeerd waarden van het type String en Number toe te voegen aan die parameter. Dit resulteert in een compilatiefout in de strikte modus.

```
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);
    }
}
```

Zelfs in de strikte modus kunt u controle van het type bij compilatie selectief omzeilen door de rechterkant van een toewijzingsinstructie zonder type te laten. U kunt een variabele of expressie zonder type markeren door een typeannotatie weg te laten of door de speciale asterisk (`*`) te gebruiken als typeannotatie. Als de parameter `xParam` in het vorige voorbeeld geen typeannotatie krijgt, wordt de code als volgt in de strikte modus gecompileerd:

```
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");
```

Type controleren bij uitvoering

Type controleren bij uitvoering vindt altijd plaats in ActionScript 3.0, of u nu compileert in de strikte modus of in de standaardmodus. Neem bijvoorbeeld een situatie waarin de waarde 3 als argument wordt doorgegeven aan een functie die een array verwacht. In de strikte modus wordt een compilatiefout gegenereerd, omdat de waarde 3 niet compatibel is met het gegevenstype Array. Als u de strikte modus uitschakelt en in de standaardmodus werkt, worden bij compilatie de niet-overeenkomende typen niet als fout gemeld. Bij controle van het type bij uitvoering treedt echter wel een uitvoeringsfout op.

In het volgende voorbeeld wordt een functie getoond met de naam `typeTest()` die een argument van het type Array verwacht, maar de waarde 3 krijgt doorgegeven. Dit resulteert in een uitvoeringsfout in de standaardmodus, omdat de waarde 3 geen lid is van het gedeclareerde gegevenstype voor de parameter (Array).

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

Er kunnen zich ook situaties voordoen waarin een uitvoeringsfout optreedt zelfs terwijl u in de strikte modus werkt. Dit is mogelijk wanneer u de strikte modus gebruikt, maar controle van het type bij compilatie omzeilt door een variabele zonder type te gebruiken. Wanneer u een variabele zonder type gebruikt, wordt het controleren van het type niet uitgeschakeld, maar uitgesteld tot uitvoering plaatsvindt. Als bijvoorbeeld de variabele `myNum` in het vorige voorbeeld geen gedeclareerd gegevenstype heeft, worden de niet-overeenkomende typen niet gedetecteerd in de compiler. De code genereert echter een uitvoeringsfout, omdat de uitvoeringswaarde van `myNum`, die op 3 is ingesteld door de toewijzingsinstructie, wordt vergeleken met het type van `xParam`, die is ingesteld op het gegevenstype Array.

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

Controle van het type bij uitvoering biedt tevens meer flexibiliteit voor het gebruik van overerving dan controle bij compilatie. Door de controle van het type uit te stellen tot de uitvoering, kunt u in de standaardmodus naar eigenschappen van een subklasse verwijzen zelfs wanneer er sprake is van *upcasting*. Een upcast treedt op wanneer u een basisklasse gebruikt om het type van een klasseninstantie te declareren, maar een subklasse om deze te instantiëren. U kunt bijvoorbeeld een klasse maken met de naam ClassBase die kan worden uitgebreid (klassen met het attribuut `final` kunnen niet worden uitgebreid):

```
class ClassBase
{
}
```

Vervolgens kunt u als volgt een subklasse van ClassBase maken met de naam ClassExtender, die één eigenschap heeft met de naam `someString`:

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

Met behulp van beide klassen kunt u een klasseninstantie maken die is gedeclareerd met het gegevenstype ClassBase, maar geïnstantieerd met de constructor ClassExtender. Een upcast wordt als een veilige bewerking beschouwd, omdat de basisklasse geen eigenschappen of methoden bevat die zich niet in de subklasse bevinden.

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

Een subklasse bevat echter eigenschappen of methoden die de basisklasse niet heeft. De klasse ClassExtender bevat bijvoorbeeld de eigenschap `someString`, die niet bestaat in de klasse ClassBase. In de standaardmodus van ActionScript 3.0 kunt u naar deze eigenschap verwijzen met de instantie `myClass` zonder een compilatiefout te genereren, zoals in het volgende voorbeeld wordt getoond:

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

De operator is

Met de operator `is` kunt u testen of een variabele of expressie lid is van een bepaald gegevenstype. In eerdere versies van ActionScript vervulde de operator `instanceof` deze rol. In ActionScript 3.0 moet u de operator `instanceof` echter niet gebruiken om lidmaatschap van gegevenstypen te testen. U gebruikt de operator `is` in plaats van de operator `instanceof` om het type handmatig te controleren, omdat de expressie `x instanceof y` alleen maar de prototypeketen van `x` controleert op het bestaan van `y` (en in ActionScript 3.0 geeft de prototypeketen geen volledig beeld van de overervingshiërarchie).

De operator `is` controleert de juiste overervingshiërarchie. U kunt daarmee niet alleen controleren of een object een instantie is van een bepaalde klasse, maar ook of een object een instantie is van een klasse die een bepaalde interface implementeert. In het volgende voorbeeld wordt een instantie gemaakt van de klasse Sprite met de naam `mySprite` en wordt met de operator `is` getest of `mySprite` een instantie is van de klassen Sprite en DisplayObject en of implementatie van de interface IEventDispatcher plaatsvindt:

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

De operator `is` controleert de overervingshiërarchie en rapporteert correct dat `mySprite` compatibel is met de klassen Sprite en DisplayObject (de klasse Sprite is een subklasse van de klasse DisplayObject). De operator `is` controleert ook of `mySprite` overerft van enige andere klassen die de interface IEventDispatcher implementeren. Omdat de klasse Sprite overerft van de klasse EventDispatcher, die de interface IEventDispatcher implementeert, rapporteert de operator `is` correct dat `mySprite` dezelfde interface implementeert.

In het volgende voorbeeld worden dezelfde tests als in het vorige voorbeeld getoond, maar nu met de operator `instanceof` in plaats van `is`. De operator `instanceof` stelt correct vast dat `mySprite` een instantie is van Sprite of DisplayObject, maar retourneert `false` bij het testen of `mySprite` de interface IEventDispatcher implementeert.

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

De operator as

Met de operator `as` kunt u ook controleren of een expressie lid is van een bepaald gegevenstype. Anders dan bij de operator `is`, retourneert de operator `as` echter geen Booleaanse waarde. De operator `as` retourneert de waarde van de expressie in plaats van `true` en `null` in plaats van `false`. In het volgende voorbeeld worden de resultaten getoond van het gebruik van de operator `as` in plaats van de operator `is`. Het betreft een eenvoudige controle of een instantie van Sprite lid is van de gegevenstypen DisplayObject, IEventDispatcher en Number.

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

Wanneer u de operator `as` gebruikt, moet de operand aan de rechterkant een gegevenstype zijn. Een poging om aan de rechterkant een andere expressie dan een gegevenstype te gebruiken als operand, resulteert in een fout.

Dynamische klassen

Een *dynamische* klasse definieert een object dat bij uitvoering kan worden gewijzigd door eigenschappen en methoden toe te voegen of te wijzigen. Een klasse die niet dynamisch is, zoals de klasse String, is een *verzegelde* klasse. U kunt bij uitvoering geen eigenschappen of methoden toevoegen aan een verzegelde klasse.

U maakt dynamische klassen door het attribuut `dynamic` te gebruiken wanneer u een klasse declareert. Met de volgende code wordt bijvoorbeeld een dynamische klasse gemaakt met de naam `Protean`:

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

Als u later een instantie van de klasse `Protean` instantieert, kunt u er eigenschappen of methoden aan toevoegen buiten de klassendefinitie. Met de volgende code wordt bijvoorbeeld een instantie gemaakt van de klasse `Protean` en wordt aan de instantie een eigenschap toegevoegd met de naam `aString` en een eigenschap met de naam `aNumber`:

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

Eigenschappen die u toevoegt aan een instantie van een dynamische klasse zijn uitvoeringsentiteiten, zodat de controle van het type bij uitvoering plaatsvindt. U kunt geen typeannotatie toevoegen aan een eigenschap die u op deze manier toevoegt.

U kunt ook een methode toevoegen aan de instantie `myProtean` door een functie te definiëren en de functie te koppelen aan een eigenschap van de instantie `myProtean`. Met de volgende code wordt de instructie trace verplaatst naar een methode met de naam `traceProtean()`:

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

Methoden die u op deze manier maakt, hebben echter geen toegang tot eigenschappen of methoden van de klasse Protean van het type private. Bovendien moeten zelfs verwijzingen naar eigenschappen en methoden van de klasse `Protean` van het type public worden gekwalificeerd met het trefwoord `this` of met de klassennaam. In het volgende voorbeeld wordt getoond hoe de methode `traceProtean()` probeert toegang te krijgen tot variabelen van de klasse `Protean` van het type private en public.

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

Beschrijving van gegevenstypen

De primitieve gegevenstypen zijn Boolean, int, Null, Number, String, uint en void. De kernklassen van ActionScript definiëren ook de volgende complexe gegevenstypen: Object, Array, Date, Error, Function, RegExp, XML en XMLList.

Boolean, gegevenstype

Het gegevenstype Boolean bestaat uit twee waarden: `true` en `false`. Voor variabelen van het type Boolean zijn geen andere waarden geldig. De standaardwaarde van een Booleaanse variabele die is gedeclareerd maar niet is geïnitialiseerd, is `false`.

int, gegevenstype

Het gegevenstype int wordt intern opgeslagen als een 32-bits geheel getal en bestaat uit de set gehele getallen van

-2,147,483,648 (-2^{31}) tot en met 2.147.483.647 ($2^{31} - 1$). Eerdere versies van ActionScript kenden alleen het gegevenstype Number, voor zowel gehele getallen als getallen met drijvende komma. In ActionScript 3.0 hebt u nu toegang tot machinetypen op laag niveau voor 32-bits gehele getallen met en zonder teken. Als de variabele geen getallen met drijvende komma nodig heeft, is het gebruik van het gegevenstype int in plaats van het gegevenstype Number sneller en efficiënter.

Voor waarden van gehele getallen buiten het bereik van de minimale en maximale waarde voor int gebruikt u het gegevenstype Number. Daarmee zijn waarden mogelijk tussen positief en negatief 9.007.199.254.740.992 (53-bits gehele getallen). De standaardwaarde voor variabelen van het gegevenstype int is 0.

Null, gegevenstype

Het gegevenstype Null bevat maar één waarde: `null`. Dit is de standaardwaarde voor het gegevenstype String en alle klassen die complexe gegevenstypen definiëren, waaronder de klasse Object. Geen van de andere primitieve gegevenstypen, zoals Boolean, Number, int en uint, bevat de waarde `null`. Tijdens de uitvoering wordt de waarde `null` omgezet in de toepasselijke standaardwaarde wanneer u `null` probeert toe te wijzen aan variabelen van het type Boolean, Nummer, int of uint. U kunt dit gegevenstype niet als typeannotatie gebruiken.

Number, gegevenstype

In ActionScript 3.0 kan het gegevenstype Number gehele getallen, gehele getallen zonder teken en getallen met drijvende komma vertegenwoordigen. Voor maximale prestaties dient u het gegevenstype Number echter alleen te gebruiken voor gehele getallen met waarden groter dan de 32-bits typen `int` en `uint` kunnen opslaan, of voor getallen met drijvende komma. Als u een getal met drijvende komma wilt opslaan, moet u een decimaal teken opnemen in het getal. Als u het decimale teken weglaat, wordt het getal opgeslagen als een geheel getal.

Het gegevenstype Number gebruikt de 64-bits indeling met dubbele precisie zoals is gespecificeerd in de standaard IEEE-754 (voor berekeningen met binaire getallen met drijvende komma). Deze standaard schrijft voor hoe getallen met drijvende komma worden opgeslagen met behulp van de 64 beschikbare bits. Eén bit wordt gebruikt om aan te duiden of het getal positief of negatief is. Elf bits worden gebruikt voor de exponent, die wordt opgeslagen als grondtal 2. De resterende 52 bits worden gebruikt voor de opslag van de *significant* (ook wel *mantisse* genoemd). Dit is het getal dat tot de macht wordt verheven die de exponent aangeeft.

Door sommige bits voor de opslag van een exponent te gebruiken kan het gegevenstype Number getallen met drijvende komma opslaan die aanzienlijk hoger zijn dan wanneer alle bits voor de significant zouden worden gebruikt. Als het gegevenstype Number bijvoorbeeld alle 64 bits zou gebruiken voor de opslag van de significant, kan het een getal opslaan van maximaal 2^{65} - 1. Door 11 bits te gebruiken voor de opslag van een exponent, kan het gegevenstype Number de significant verheffen tot de macht 2^{1023}.

De maximale en minimale waarde die het gegevenstype Number kan vertegenwoordigen, wordt opgeslagen in statische eigenschappen van de klasse Number met de naam `Number.MAX_VALUE` en `Number.MIN_VALUE`.

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

Hoewel dit bereik van getallen enorm is, gaat dit ten koste van de precisie. Het gegevenstype Number gebruikt 52 bits voor de opslag van de significant. Dit heeft tot gevolg dat getallen waarvoor meer dan 52 bits nodig zijn voor een exacte representatie, zoals de breuk 1/3, slechts benaderingen zijn. Als voor uw toepassing absolute precisie is vereist voor decimale getallen, moet u software gebruiken die berekeningen met decimale getallen met drijvende komma implementeert in tegenstelling tot berekeningen met binaire getallen met drijvende komma.

Wanneer u waarden van gehele getallen opslaat met het gegevenstype Number, worden alleen de 52 bits van de significant gebruikt. Het gegevenstype Number gebruikt deze 52 bits en een speciale verborgen bit om gehele getallen te vertegenwoordigen van -9.007.199.254.740.992 (-2^{53}) tot 9.007.199.254.740.992 (2^{53}).

De waarde `NaN` wordt niet alleen gebruikt als de standaardwaarde voor variabelen van het type `Number`, maar ook als het resultaat van een bewerking die een getal moet retourneren maar dat niet doet. Als u bijvoorbeeld probeert de vierkantswortel van een negatief getal te berekenen, is het resultaat `NaN`. Andere speciale waarden voor Number zijn *positief oneindig* en *negatief oneindig*.

***Opmerking:** Het resultaat van delen door `0` is alleen `NaN` als de deler ook `0` is. Delen door `0` resulteert in `oneindig` wanneer het deeltal positief is of in `-oneindig` wanneer het deeltal negatief is.*

String, gegevenstype

Het gegevenstype String vertegenwoordigt een reeks van 16-bits tekens. Tekenreeksen worden intern opgeslagen als Unicode-tekens in de UTF-16-indeling. Tekenreeksen zijn, net als in de programmeertaal Java, onveranderlijk. Een bewerking op een tekenreekswaarde retourneert een nieuwe instantie van de tekenreeks. De standaardwaarde voor een variabele die met het gegevenstype String wordt gedeclareerd, is `null`. De waarde `null` is niet hetzelfde als de lege string (`""`). De waarde `null` betekent dat er in de variabele geen waarde is opgeslagen, terwijl de lege string betekent dat de variabele een waarde bevat, die bestaat uit een String zonder tekens.

uint, gegevenstype

Het gegevenstype uint wordt intern opgeslagen als een 32-bits geheel getal zonder teken en bestaat uit de set gehele getallen van 0 tot en met 4.294.967.295 (2^{32} - 1). Gebruik het gegevenstype uint in speciale situaties waarbij niet-negatieve gehele getallen nodig zijn. U moet het gegevenstype uint bijvoorbeeld gebruiken om pixelkleurwaarden te vertegenwoordigen. Het gegevenstype int heeft namelijk een interne tekenbit die niet geschikt is voor de verwerking van kleurwaarden. Voor waarden van gehele getallen boven de maximale waarde voor uint gebruikt u het gegevenstype Number. Daarmee zijn waarden mogelijk van 53-bits gehele getallen. De standaardwaarde voor variabelen van het gegevenstype uint is 0.

void, gegevenstype

Het gegevenstype void bevat maar één waarde: `undefined`. In eerdere versies van ActionScript was `undefined` de standaardwaarde voor instanties van de klasse Object. In ActionScript 3.0 is de standaardwaarde voor instanties van Object `null`. Wanneer u probeert de waarde `undefined` toe te wijzen aan een instantie van de klasse Object, wordt de waarde omgezet in `null`. U kunt alleen een waarde `undefined` toewijzen aan variabelen zonder type. Variabelen zonder type zijn variabelen zonder typeannotatie of met het asterisksymbool (`*`) als typeannotatie. U kunt `void` alleen gebruiken als retourneringstypeannotatie.

Object, gegevenstype

Het gegevenstype Object wordt gedefinieerd door de klasse Object. De klasse Object dient als basisklasse voor alle klassendefinities in ActionScript. De versie van het gegevenstype Object in ActionScript 3.0 verschilt op drie manieren van eerdere versies. Ten eerste is het gegevenstype Object niet meer het standaardgegevenstype dat wordt toegewezen aan variabelen zonder typeannotatie. Ten tweede omvat het gegevenstype Object niet meer de waarde `undefined`, die de standaardwaarde was voor instanties van Object. Ten derde is in ActionScript 3.0 de standaardwaarde voor instanties van de klasse Object `null`.

In eerdere versies van ActionScript werd aan een variabele zonder typeannotatie automatisch het gegevenstype Object toegewezen. Dit is niet meer het geval in ActionScript 3.0, waarin nu werkelijk variabelen zonder type voorkomen. Variabelen zonder typeannotatie worden nu als variabelen zonder type beschouwd. Als u aan lezers van de code duidelijk wilt maken dat u een variabele opzettelijk zonder type hebt gelaten, kunt u het asterisksymbool (`*`) gebruiken als typeannotatie. Dit staat gelijk aan het weglaten van een typeannotatie. In het volgende voorbeeld worden twee gelijkwaardige instructies getoond, waarbij in beide gevallen een variabele zonder type `x` wordt gedeclareerd:

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

Alleen variabelen zonder type kunnen de waarde `undefined` bevatten. Wanneer u probeert de waarde `undefined` toe te wijzen aan een variabele met een gegevenstype, wordt tijdens de uitvoering de waarde `undefined` omgezet in de standaardwaarde voor dat gegevenstype. Voor instanties van het gegevenstype Object, is de standaardwaarde `null`, wat betekent dat als u probeert om `undefined` aan een Objectinstantie toe te wijzen, de waarde wordt omgezet in `null`.

Typeomzettingen

Een typeomzetting vindt plaats wanneer een waarde wordt omgezet in een waarde van een ander gegevenstype. Typeomzettingen kunnen *impliciet* of *expliciet* zijn. Impliciete omzetting, wat ook wel *afgedwongen* wordt genoemd, wordt soms tijdens de uitvoering uitgevoerd. Als de waarde 2 bijvoorbeeld wordt toegewezen aan een variabele van het gegevenstype Boolean, wordt de waarde 2 omgezet in de Booleaanse waarde `true` voordat de waarde aan de variabele wordt toegewezen. Expliciete omzetting, ook wel *casting* genoemd, treedt op wanneer de code de compiler opdraagt een variabele van een bepaald gegevenstype te behandelen alsof deze tot een ander gegevenstype behoort. In het geval van primitieve waarden worden met casting waarden van het ene gegevenstype daadwerkelijk in een ander type omgezet. Wanneer u een object wilt casten naar een ander type, plaatst u het object tussen ronde haakjes en laat u deze voorafgaan door de naam van het nieuwe type. De volgende code neemt bijvoorbeeld een Booleaanse waarde en cast deze naar een geheel getal:

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

Impliciete omzettingen

Impliciete omzettingen treden bij uitvoering op in een aantal contexten:

- In toewijzingsinstructies
- Wanneer waarden worden doorgegeven als functieargumenten
- Wanneer waarden worden geretourneerd uit functies
- In expressies met bepaalde operatoren, zoals de operator voor optellen (+)

  Voor door de gebruiker gedefinieerde typen zijn impliciete omzettingen mogelijk wanneer de om te zetten waarde een instantie is van de doelklasse of van een klasse die van de doelklasse is afgeleid. Als een impliciete omzetting mislukt, treedt een fout op. De volgende code bevat bijvoorbeeld een geslaagde impliciete omzetting en een mislukte impliciete omzetting:

  ```
  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.
  ```

  Voor primitieve typen worden impliciete omzettingen afgehandeld door dezelfde interne omzettingsalgoritmen aan te roepen als voor expliciete omzettingsfuncties.

Expliciete omzettingen

Het toepassen van expliciete omzettingen, of casting, is nuttig wanneer u in de strikte modus compileert en bijvoorbeeld wilt voorkomen dat niet-overeenkomende typen in compilatiefouten resulteren. Dit kan het geval zijn wanneer u weet dat door afgedwongen omzetting de waarden bij uitvoering correct worden omgezet. Wanneer u bijvoorbeeld werkt met gegevens die afkomstig zijn uit een formulier, kunt u door afgedwongen omzetting ervoor zorgen dat bepaalde tekenreekswaarden worden omgezet in numerieke waarden. Met de volgende code wordt een compilatiefout gegenereerd, ook al zou de code in de standaardmodus correct worden uitgevoerd:

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

Als u verder wilt werken in de strikte modus maar de tekenreeks wilt omzetten in een geheel getal, kunt u als volgt expliciete omzetting toepassen:

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

Casting naar int, uint en Number

U kunt elk gegevenstype casten naar een van de drie getaltypen: int, uint en Number. Als het getal om een bepaalde reden niet kan worden omgezet, wordt de standaardwaarde 0 voor de gegevenstypen int en uint toegewezen en wordt de standaardwaarde van `NaN` voor het gegevenstype Nummer toegewezen. Als u een Booleaanse waarde omzet in een getal, wordt `true` de waarde 1 en `false` de waarde 0.

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

Tekenreekswaarden met alleen cijfers kunnen in een van de getaltypen worden omgezet. De getaltypen kunnen ook tekenreeksen omzetten die eruitzien als negatieve getallen of tekenreeksen die een hexadecimale waarde vertegenwoordigen (bijvoorbeeld `0x1A`). Bij het omzettingsproces worden voorafgaande en navolgende witruimtetekens in de tekenreekswaarde genegeerd. U kunt ook tekenreeksen casten die eruitzien als getallen met drijvende komma door `Number()` te gebruiken. Het opnemen van een decimaal teken zorgt ervoor dat `uint()` en `int()` een geheel getal retourneren met de tekens achter het decimale teken ingekort. De volgende tekenreekswaarden kunnen bijvoorbeeld naar getallen worden gecast.

```
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
```

Tekenreekswaarden met niet-numerieke tekens retourneren 0 bij het casten met `int()` of `uint()`, en `NaN` bij het casten met `Number()`. Bij het omzettingsproces worden voorafgaande en navolgende witruimtetekens genegeerd, maar wordt 0 of `NaN` geretourneerd als een tekenreeks witruimte bevat die twee getallen van elkaar scheidt.

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

In ActionScript 3.0 ondersteunt de functie `Number()` geen octale getallen (met grondtal 8). Als u een tekenreeks met een voorloopnul gebruikt voor de functie `Number()` in ActionScript 2.0, wordt het getal als octaal getal geïnterpreteerd en omgezet in het decimale equivalent ervan. Dat geldt niet voor de functie `Number()` in ActionScript 3.0, waarbij de voorloopnul wordt genegeerd. Met de volgende code wordt bijvoorbeeld verschillende uitvoer gegenereerd bij het compileren met verschillende versies van ActionScript:

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

Casting is niet nodig wanneer een waarde van een numeriek type wordt toegewezen aan een variabele van een ander numeriek type. Zelfs in de strikte modus wordt het numerieke type impliciet omgezet in het andere numerieke type. Dit kan in sommige gevallen resulteren in onverwachte waarden wanneer het bereik van een type wordt overschreden. De volgende voorbeelden worden allemaal gecompileerd in de strikte modus, hoewel bij sommige onverwachte waarden worden gegenereerd:

```
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 de volgende tabel wordt een overzicht gegeven van de resultaten van casting naar het gegevenstype Number, int of uint van andere gegevenstypen.

Gegevenstype of waarde	Resultaat van omzetting in Number, int of uint
Boolean	Als de waarde `true` is, 1; anders 0.
Date	De interne representatie van het object Date, ofwel het aantal milliseconden dat is verstreken sinds middernacht op 1 januari 1970, universele tijd.
`null`	0
Object	Als de instantie `null` is en omgezet in Number, `NaN`; anders 0.
Tekenreeks	Een getal als de string in een getal kan worden omgezet en anders `NaN` als het in een Number wordt omgezet of 0 als het wordt omgezet in int of uint.
`undefined`	Bij omzetting in Number, `NaN`; bij omzetting in int of uint, 0.

Casting naar Boolean

Casting naar Boolean van een numeriek gegevenstype (uint, int en Number) resulteert in `false` als de numerieke waarde 0 is, anders in `true`. Voor het gegevenstype Number resulteert de waarde `NaN` ook in `false`. In het volgende voorbeeld wordt het resultaat getoond van het casten van de getallen -1, 0 en 1:

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

De uitvoer in het voorbeeld laat zien dat van de drie getallen alleen 0 de waarde `false` retourneert:

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

Casting naar Boolean van een tekenreekswaarde retourneert `false` als de tekenreeks `null` is of leeg is (`""`). Anders wordt `true` geretourneerd.

```
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
```

Casting naar Boolean van een instantie van de klasse Object retourneert `false` als de instantie `null` is; anders `true`:

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

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

Variabelen van het type Boolean worden in de strikte modus speciaal behandeld. U kunt namelijk zonder casting waarden van elk gegevenstype toewijzen aan een variabele van het type Boolean. Impliciete afgedwongen omzetting van alle gegevenstypen in het gegevenstype Boolean gebeurt zelfs in de strikte modus. Met andere woorden, casting naar Boolean is, anders dan bij bijna alle andere gegevenstypen, niet nodig om fouten in de strikte modus te voorkomen. De volgende voorbeelden worden allemaal gecompileerd in de strikte modus en het gedrag bij uitvoering is zoals wordt verwacht:

```
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 de volgende tabel wordt een overzicht gegeven van de resultaten van casting naar het gegevenstype Boolean van andere gegevenstypen:

Gegevenstype of waarde	**Resultaat van omzetting in Boolean**
Tekenreeks	`false` als de waarde `null` of een lege tekenreeks (`""`) is; anders `true`.
`null`	`false`
Number, int of uint	`false` als de waarde `NaN` of 0 is; anders `true`.
Object	`false` als de instantie `null` is; anders `true`.

Casting naar String

Casting naar het gegevenstype String van een numeriek gegevenstype retourneert een tekenreeksrepresentatie van het getal. Casting naar het gegevenstype String van een Booleaanse waarde retourneert de tekenreeks `'true'` als de waarde `true` is en retourneert de tekenreeks `'false'` als de waarde `false` is.

Casting naar het gegevenstype String van een instantie van de klasse Object retourneert de tekenreeks `'null'` als de instantie `null` is. Anders wordt bij het casten naar het gegevenstype String van een instantie van de klasse Object de tekenreeks `'[object Object]'` geretourneerd.

Casting naar String van een instantie van de klasse Array retourneert een tekenreeks die bestaat uit een door komma's gescheiden lijst met alle arrayelementen. Bij het casten naar het gegevenstype String wordt in het volgende voorbeeld één tekenreeks geretourneerd met de drie elementen van de array:

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

Casting naar String van een instantie van de klasse Date retourneert een tekenreeksrepresentatie van de datum die de instantie bevat. In het volgende voorbeeld wordt een tekenreeksrepresentatie geretourneerd van een instantie van de klasse Date:

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

In de volgende tabel wordt een overzicht gegeven van de resultaten van casting naar het gegevenstype String van andere gegevenstypen.

Gegevenstype of waarde	**Resultaat van omzetting in String**
Array	Een tekenreeks die bestaat uit alle arrayelementen.
Boolean	"`true`" of "`false`"
Date	Een tekenreeksrepresentatie van het object Date.
`null`	`"null"`
Number, int of uint	Een tekenreeksrepresentatie van het getal.
Object	Als de instantie null is `'null'`; anders `'[object Object]'`.

Syntaxis

De syntaxis van een taal definieert een set regels die bij het schrijven van uitvoerbare code moet worden gevolgd.

Hoofdlettergevoeligheid

ActionScript 3.0 is een hoofdlettergevoelige taal. Id's die alleen verschillen in hoofdlettergebruik, worden als verschillende id's beschouwd. Met de volgende code worden bijvoorbeeld twee verschillende variabelen gemaakt:

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

Puntnotatie

De puntoperator (`.`) biedt toegang tot de eigenschappen en methoden van een object. Met de puntnotatie kunt u verwijzen naar een klasseneigenschap of -methode door een instantienaam te gebruiken, gevolgd door de puntoperator en naam van de eigenschap of methode. Neem bijvoorbeeld de volgende klassendefinitie:

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

Met behulp van de puntnotatie hebt u toegang tot de eigenschap `prop1` en de methode `method1()` door de instantienaam te gebruiken die met de volgende code wordt gemaakt:

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

U kunt de puntnotatie gebruiken wanneer u pakketten definieert. U gebruikt de puntoperator voor het verwijzen naar geneste pakketten. De klasse EventDispatcher bevindt zich bijvoorbeeld in een pakket met de naam events dat is genest binnen het pakket met de naam flash. U kunt met de volgende expressie verwijzen naar het pakket events:

```
flash.events
```

U kunt ook naar de klasse EventDispatcher verwijzen met deze expressie:

```
flash.events.EventDispatcher
```

Slash-notatie

De slash-notatie wordt niet ondersteund in ActionScript 3.0. De slash-notatie werd in eerdere versies van ActionScript gebruikt om het pad van een filmclip of variabele aan te geven.

Letterlijke tekens

Een *letterlijk teken* is een waarde die direct wordt weergegeven in uw code. De volgende voorbeelden zijn allemaal letterlijke tekens:

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

U kunt letterlijke tekens ook groeperen om samengestelde letterlijke tekens te vormen. Arrays met letterlijke tekens worden ingesloten door vierkante haakjes (`[]`) en gebruiken de komma om arrayelementen te scheiden.

U kunt een array met letterlijke tekens gebruiken om een array te initialiseren. In de volgende voorbeelden worden twee arrays getoond die zijn geïnitialiseerd door middel van arrays met letterlijke tekens. U kunt de instructie `new` gebruiken en het samengestelde letterlijke teken als parameter doorgeven aan de klassenconstructor Array. U kunt echter ook direct letterlijke waarden toewijzen wanneer u instanties van de volgende kernklassen van ActionScript instantieert: Object, Array, String, Number, int, uint, XML, XMLList en Boolean.

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

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

U kunt letterlijke tekens ook gebruiken om een algemeen object te initialiseren. Een algemeen object is een instantie van de klasse Object. Letterlijke tekens voor objecten worden ingesloten door accolades (`{}`) en gebruiken komma's om objecteigenschappen te scheiden. Elke eigenschap wordt gedeclareerd met de dubbele punt (`:`), die de naam van de eigenschap scheidt van de waarde van de eigenschap.

U kunt een algemeen object maken met de instructie `new` en het letterlijke teken voor objecten als parameter doorgeven aan de klassenconstructor Object. U kunt ook het letterlijke teken voor objecten direct toewijzen aan de instantie die u declareert. Het volgende voorbeeld laat twee andere manieren zien om een nieuw algemeen object te maken en het object te initialiseren met drie eigenschappen (`propA`, `propB` en `propC`), elk met een waarde ingesteld op respectievelijk 1, 2 en 3:

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

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

Meer Help-onderwerpen

Werken met tekenreeksen

Reguliere expressies gebruiken

XML-variabelen initialiseren

Puntkomma's

U kunt het puntkommateken (`;`) gebruiken om een instructie te beëindigen. U kunt de puntkomma ook weglaten. De compiler gaat er dan van uit dat elke coderegel een enkele instructie vertegenwoordigt. Veel programmeurs zijn gewend met de puntkomma het einde van een instructie aan te geven. Uw code is dan ook wellicht beter leesbaar als u altijd puntkomma's gebruikt om instructies te beëindigen.

Wanneer u een puntkomma gebruikt om een instructie te beëindigen, kunt u meerdere instructies in een enkele regel plaatsen. Dit maakt uw code echter meestal minder leesbaar.

Ronde haakjes

U kunt ronde haakjes (`()`) op drie manieren gebruiken in ActionScript 3.0. Ten eerste kunt u ronde haakjes gebruiken om de volgorde van de bewerkingen in een expressie te wijzigen. Bewerkingen die tussen ronde haakjes zijn gegroepeerd, worden altijd het eerst uitgevoerd. Ronde haakjes worden bijvoorbeeld gebruikt om de volgorde van de bewerkingen in de volgende code te wijzigen:

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

Ten tweede kunt u ronde haakjes gebruiken met de kommaoperator (`,`) om een reeks expressies te evalueren en de resultaten te retourneren van de uiteindelijke expressie, zoals in het volgende voorbeeld wordt getoond:

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

Ten derde kunt u ronde haakjes gebruiken om een of meer parameters door te geven aan functies of methoden, zoals in het volgende voorbeeld wordt getoond. Hierin wordt een tekenreekswaarde doorgegeven aan de functie `trace()`:

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

Opmerkingen

Code in ActionScript 3.0 ondersteunt twee typen opmerkingen: opmerkingen van een enkele regel en opmerkingen van meerdere regels. Deze opmerkingsmechanismen lijken op die in C++ en Java. De compiler negeert tekst die als opmerking is gemarkeerd.

Opmerkingen van een enkele regel beginnen met twee slash-tekens (`//`) en lopen door tot het einde van de regel. De volgende code bevat bijvoorbeeld een opmerking van een enkele regel:

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

Opmerkingen van meerdere regels beginnen met een slash en asterisk (`/*`) en eindigen met een asterisk en slash (`*/`).

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

Trefwoorden en gereserveerde woorden

Gereserveerde woorden zijn woorden die u niet als id's in uw code kunt gebruiken, omdat de woorden zijn gereserveerd voor gebruik door ActionScript. Gereserveerde woorden omvatten *lexicale trefwoorden*, die uit de programmanaamruimte worden gehaald door de compiler. De compiler rapporteert een fout als u een lexicaal trefwoord als id gebruikt. In de volgende tabel worden de lexicale trefwoorden van ActionScript 3.0 weergegeven.

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

Er is een kleine set trefwoorden, *syntactische trefwoorden* genoemd, die als id's kunnen worden gebruikt maar in bepaalde contexten een speciale betekenis hebben. In de volgende tabel worden de syntactische trefwoorden van ActionScript 3.0 weergegeven.

each	get	set	namespace
include	dynamic	final	native
override	static		

Er zijn ook diverse id's die soms *toekomstig gereserveerde woorden* worden genoemd. Deze id's zijn niet gereserveerd door ActionScript 3.0, hoewel sommige ervan door software waarin ActionScript 3.0 is opgenomen mogelijk als trefwoorden worden behandeld. U kunt vele van deze id's wellicht in uw code gebruiken, maar Adobe geeft de aanbeveling dat niet te doen, omdat ze mogelijk als trefwoorden in een volgende versie van de taal worden gebruikt.

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

Constanten

ActionScript 3.0 ondersteunt de instructie `const`, waarmee u constanten kunt maken. Constanten zijn eigenschappen met een vaste, onveranderbare waarde. U kunt maar één keer een waarde aan een constante toewijzen en de toewijzing moet plaatsvinden dicht bij de declaratie van de constante. Als een constante bijvoorbeeld als lid van een klasse wordt gedeclareerd, kunt u alleen een waarde aan die constante toewijzen als onderdeel van de declaratie of binnen de klassenconstructor.

Met de volgende code worden twee constanten gedeclareerd. De eerste constante, `MINIMUM`, krijgt een waarde toegewezen als onderdeel van de declaratie-instructie. De tweede constante, `MAXIMUM`, krijgt een waarde toegewezen in de constructor. Dit voorbeeld kan alleen worden gecompileerd in de standaardmodus, omdat in de strikte modus de waarde van een constante alleen op het moment van initialisatie kan worden toegewezen.

```
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
```

Een fout treedt op als u op een andere wijze probeert een beginwaarde aan een constante toe te wijzen. Als u bijvoorbeeld probeert de beginwaarde van `MAXIMUM` buiten de klasse in te stellen, treedt een uitvoeringsfout op.

```
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 definieert een groot aantal constanten die u kunt gebruiken. Volgens goed gebruik worden voor constanten in ActionScript hoofdletters gebruikt, waarbij woorden door het onderstrepingsteken (_) worden gescheiden. De klassendefinitie MouseEvent gebruikt deze naamconventie bijvoorbeeld voor constanten, waarbij elke constante een gebeurtenis vertegenwoordigt die met de muisinvoer te maken heeft:

```
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 zijn speciale functies die een of meer operanden gebruiken en een waarde retourneren. Een *operand* is een waarde (meestal een letterlijke waarde, een variabele of een expressie) die door een operator als invoer wordt gebruikt. Met de volgende code worden bijvoorbeeld de operatoren voor optellen (`+`) en vermenigvuldigen (`*`) gebruikt met drie letterlijke operanden (2, 3, en 4) om een waarde te retourneren. Deze waarde wordt vervolgens gebruikt door de toewijzingsoperator (`=`) om de geretourneerde waarde (14) toe te wijzen aan de variabele `sumNumber`.

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

Operatoren kunnen unair, binair of ternair zijn. Een *unaire* operator werkt met één operand. De operator voor verhogen (`++`) is bijvoorbeeld een unaire operator, omdat maar één operand wordt gebruikt. Een *binaire* operator werkt met twee operanden. De operator voor delen (`/`) werkt bijvoorbeeld met twee operanden. Een *ternaire* operator werkt met drie operanden. De voorwaardelijke operator (`?:`) werkt bijvoorbeeld met drie operanden.

Sommige operatoren zijn *overbelast*, wat betekent dat ze zich anders gedragen afhankelijk van het type of de hoeveelheid operanden die eraan worden doorgegeven. De operator voor optellen (`+`) is een voorbeeld van een overbelaste operator die zich anders gedraagt afhankelijk van het gegevenstype van de operanden. Als beide operanden getallen zijn, retourneert de operator voor optellen de som van de waarden. Als beide operanden tekenreeksen zijn, retourneert de operator voor optellen de samenvoeging van de twee operanden. De volgende voorbeeldcode laat zien hoe de operator zich anders gedraagt afhankelijk van de operanden:

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

Operatoren kunnen zich ook anders gedragen op basis van het aantal gebruikte operanden. De operator voor aftrekken (`-`) is zowel een unaire als binaire operator. Met één operand maakt de operator voor aftrekken de operand negatief en retourneert het resultaat. Met twee operanden retourneert de operator voor aftrekken het verschil tussen de operanden. In het volgende voorbeeld wordt de operator voor aftrekken eerst als unaire operator gebruikt en vervolgens als binaire operator.

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

Operatorprioriteit en associatie

Operatorprioriteit en operatorassociatie bepalen de volgorde waarin operatoren worden uitgevoerd. Hoewel het vanzelfsprekend is voor personen die bekend zijn met wiskunde of eenvoudig programmeren dat de compiler de operator voor vermenigvuldigen (`*`) uitvoert vóór de operator voor optellen (`+`), heeft de compiler expliciete instructies nodig over welke operatoren eerst moeten worden uitgevoerd. Dergelijke instructies worden gezamenlijk *operatorprioriteit* genoemd. ActionScript definieert een standaardoperatorprioriteit die u kunt wijzigen door de ronde-haakjesoperator (`()`) te gebruiken. Met de volgende code wordt bijvoorbeeld de standaardprioriteit in het vorige voorbeeld gewijzigd, zodat de compiler wordt gedwongen de operator voor optellen te verwerken vóór de operator voor vermenigvuldigen:

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

Er kunnen situaties voorkomen waarin twee of meer operatoren met dezelfde prioriteit worden weergegeven in dezelfde expressie. In deze gevallen gebruikt de compiler de regels van *associatie* om te bepalen welke operator het eerst moet worden uitgevoerd. Alle binaire operatoren, behalve de toewijzingsoperatoren, hebben *associatie naar links*, wat betekent dat operatoren aan de linkerkant eerder worden uitgevoerd dan operatoren aan de rechterkant. De toewijzingsoperatoren en de voorwaardelijke operator `?:` hebben *associatie naar rechts*, wat betekent dat operatoren aan de rechterkant eerder worden uitgevoerd dan operatoren aan de linkerkant.

Neem bijvoorbeeld de operatoren kleiner dan (`<`) en groter dan (`>`), die dezelfde prioriteit hebben. Als beide operatoren in dezelfde expressie worden gebruikt, wordt de operator aan de linkerkant eerst uitgevoerd, omdat beide operatoren associatie naar links hebben. Dit betekent dat de volgende twee instructies dezelfde uitvoer produceren:

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

De operator groter dan wordt eerst uitgevoerd, wat resulteert in de waarde `true`, omdat de operand 3 groter is dan de operand 2. De waarde `true` wordt vervolgens doorgegeven aan de operator kleiner dan, samen met de operand 1. De volgende code vertegenwoordigt deze tussenstatus:

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

De operator kleiner dan zet de waarde `true` om in de numerieke waarde 1 en vergelijkt die numerieke waarde met de tweede operand 1 om de waarde `false` te retourneren (de waarde 1 is niet kleiner dan 1).

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

U kunt de standaardassociatie naar links wijzigen met de ronde-haakjesoperator. U kunt de compiler opgeven de operator kleiner dan eerst te verwerken door die operator en de operanden ervan tussen ronde haakjes te plaatsen. In het volgende voorbeeld wordt de ronde-haakjesoperator gebruikt om een andere uitvoer te produceren met dezelfde getallen als in het vorige voorbeeld:

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

De operator kleiner dan wordt eerst uitgevoerd, wat resulteert in de waarde `false`, omdat de operand 2 niet kleiner is dan de operand 1. De waarde `false` wordt vervolgens doorgegeven aan de operator groter dan, samen met de operand 3. De volgende code vertegenwoordigt deze tussenstatus:

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

De operator groter dan zet de waarde `false` om in de numerieke waarde 0 en vergelijkt die numerieke waarde met de andere operand 3 om de waarde `true` te retourneren (de waarde 3 is groter dan 0).

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

De volgende tabel bevat een lijst met operatoren voor ActionScript 3.0 in volgorde van afnemende prioriteit. Elke rij van de tabel bevat operatoren met dezelfde prioriteit. Elke rij operatoren heeft een hogere prioriteit dan de volgende rij in de tabel.

Groep	Operatoren
Primair	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Postfix	`x++ x--`
Unair	`++x --x + - ~ ! delete typeof void`
Multiplicatief	`* / %`
Additief	`+ -`
Bitsgewijs verplaatsen	`<< >> >>>`
Relationeel	`< > <= >= as in instanceof is`
Gelijkheid	`== != === !==`
Bitsgewijze AND	`&`
Bitsgewijze XOR	`^`
Bitsgewijze OR	`|`
Logische AND	`&&`
Logische OR	`||`
Voorwaardelijk	`?:`
Toewijzing	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Komma	`,`

Primaire operatoren

De primaire operatoren omvatten de operatoren die worden gebruikt voor het maken van letterlijke tekens voor arrays en objecten, groeperen van expressies, aanroepen van functies, instantiëren van klasseninstanties en de toegang tot eigenschappen.

Alle primaire operatoren in de volgende tabel hebben dezelfde prioriteit. Operatoren die deel uitmaken van de E4X-specificatie worden aangeduid met de notatie (E4X).

Operator	Bewerking die wordt uitgevoerd
`[]`	Initialiseert een array
`{x:y}`	Initialiseert een object
`()`	Groepeert expressies
`f(x)`	Roept een functie aan
`new`	Roept een constructor aan
`x.y x[y]`	Zorgt voor toegang tot een eigenschap
`<></>`	Initialiseert een object XMLList (E4X)
`@`	Initialiseert een attribuut (E4X)
`::`	Kwalificeert een naam (E4X)
`..`	Zorgt voor toegang tot een afstammend XML-element (E4X)

Postfix-operatoren

Postfix-operatoren richten zich op één operator en verhogen of verlagen de waarde. Hoewel deze operatoren unair zijn, worden zij apart van de andere unaire operatoren geclassificeerd vanwege hun hogere prioriteit en speciale gedrag. Als u een postfix-operator als deel van een grotere expressie gebruikt, wordt de waarde van de expressie geretourneerd voordat de postfix-operator wordt verwerkt. De volgende code toont bijvoorbeeld dat de waarde van expressie `xNum++` wordt geretourneerd voordat de waarde wordt verhoogd.

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

Alle postfix-operatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
`++`	Verhoogt (postfix)
`--`	Verlaagt (postfix)

Unaire operatoren

Unaire operatoren richten zich op één operand. De operatoren `++` (verhogen) en `--` (verlagen) in deze groep zijn *prefix-operatoren*, wat betekent dat zij worden weergegeven vóór de operand in een expressie. Prefix-operatoren verschillen van de postfix-operatoren, omdat de verhogende of verlagende bewerking wordt uitgevoerd voordat de waarde van de gehele expressie wordt geretourneerd. De volgende code toont bijvoorbeeld hoe de waarde van de expressie `++xNum` wordt geretourneerd nadat de waarde wordt verhoogd:

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

Alle unaire operatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
`++`	Verhoogt (prefix)
`--`	Verlaagt (prefix)
`+`	Unair +
`-`	Unair - (negatie)
`!`	Logische NOT
`~`	Bitsgewijze NOT
`delete`	Verwijdert een eigenschap
`typeof`	Retourneert informatie omtrent het type
`void`	Retourneert een ongedefinieerde waarde

Multiplicatieve operatoren

Multiplicatieve operatoren richten zich op twee operanden en voeren de berekeningen vermenigvuldigen, delen en restbepaling bij deling uit.

Alle multiplicatieve operatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
*	Vermenigvuldigen
/	Delen
%	Restbepaling bij deling

Additieve operatoren

Additieve operatoren richten zich op twee operanden en voeren de berekeningen optellen of aftrekken uit. Alle additieve operatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
+	Optellen
-	Aftrekken

Operatoren voor bitsgewijs verplaatsen

Operatoren voor bitsgewijs verplaatsen richten zich op twee operanden en verplaatsen de bits van de eerste operand met het aantal plaatsen opgegeven door de tweede operand. Alle operatoren voor bitsgewijs verplaatsen in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
<<	Bitsgewijs naar links verplaatsen
>>	Bitsgewijs naar rechts verplaatsen
>>>	Bitsgewijs zonder teken naar rechts verplaatsen

Relationele operatoren

Relationele operatoren richten zich op twee operanden, vergelijken de waarden hiervan en retourneren een Booleaanse waarde. Alle relationele operatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
<	Kleiner dan
>	Groter dan
<=	Kleiner dan of gelijk aan
>=	Groter dan of gelijk aan
as	Controleert het gegevenstype
in	Controleert de objecteigenschappen
instanceof	Controleert de prototypeketen
is	Controleert het gegevenstype

Gelijkheidsoperatoren

Gelijkheidsoperatoren richten zich op twee operanden, vergelijken de waarden hiervan en retourneren een Booleaanse waarde. Alle gelijkheidsoperatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
`==`	Gelijkheid
`!=`	Ongelijkheid
`===`	Strikte gelijkheid
`!==`	Strikte ongelijkheid

Bitsgewijze logische operatoren

Bitsgewijze logische operatoren richten zich op twee operanden en voeren logische handelingen uit op bitniveau. De prioriteit van bitsgewijze logische operatoren verschilt. De operatoren worden, gesorteerd op aflopende prioriteit, weergegeven in de volgende tabel:

Operator	Bewerking die wordt uitgevoerd
`&`	Bitsgewijze AND
`^`	Bitsgewijze XOR
`|`	Bitsgewijze OR

Logische operatoren

Logische operatoren richten zich op twee operanden en retourneren als resultaat een Booleaanse waarde. De prioriteit van logische operatoren verschilt. De operatoren worden, gesorteerd op aflopende prioriteit, weergegeven in de volgende tabel:

Operator	Bewerking die wordt uitgevoerd
`&&`	Logische AND
`||`	Logische OR

Voorwaardelijke operator

De voorwaardelijke operator is een ternaire operator, wat betekent dat deze zich op drie operanden richt. De voorwaardelijke operator is een snelle methode om de voorwaardelijke instructie `ifelse` toe te passen.

Operator	Bewerking die wordt uitgevoerd
`?:`	Voorwaardelijk

Toewijzingsoperatoren

Toewijzingsoperatoren richten zich op twee operanden en wijzen een waarde toe aan een operand, gebaseerd op de waarde van de andere operand. Alle toewijzingsoperatoren in de volgende tabel hebben dezelfde prioriteit:

Operator	Bewerking die wordt uitgevoerd
=	Toewijzing
*=	Vermenigvuldigen en toewijzen
/=	Delen en toewijzen
%=	Restbepaling bij deling en toewijzen
+=	Optellen en toewijzen
-=	Aftrekken en toewijzen
<<=	Bitsgewijs naar links verplaatsen en toewijzen
>>=	Bitsgewijs naar rechts verplaatsen en toewijzen
>>>=	Bitsgewijs zonder teken naar rechts verplaatsen en toewijzen
&=	Bitsgewijs AND en toewijzen
^=	Bitsgewijs XOR en toewijzen
|=	Bitsgewijs OR en toewijzen

Voorwaardelijke instructies

ActionScript 3.0 bevat drie voorwaardelijke basisinstructies waarmee u de programmeerstroom kunt beheren.

if..else

Met de voorwaardelijke instructie `if..else` kunt u een voorwaarde testen en vervolgens een codeblok uitvoeren als die voorwaarde bestaat of een ander codeblok uitvoeren als de voorwaarde niet bestaat. De volgende code test bijvoorbeeld of de waarde van `x` hoger is dan 20, genereert een functie `trace()` als dat zo is en genereert een andere functie `trace()` als dat niet zo is:

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

Als u geen alternatief codeblok wilt uitvoeren, kunt u de instructie `if` zonder de instructie `else` gebruiken.

if..else if

U kunt met de voorwaardelijke instructie `if..else if` meerdere voorwaarden testen. De volgende code test bijvoorbeeld niet alleen of de waarde van `x` hoger is dan 20, maar ook of de waarde van `x` negatief is:

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

Als een instructie `if` of `else` wordt gevolgd door slechts één instructie, hoeft de instructie niet tussen accolades te staan. In de volgende code worden bijvoorbeeld geen accolades gebruikt:

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

Adobe geeft echter de aanbeveling dat u altijd accolades gebruikt. Onverwacht gedrag kan zich namelijk voordoen wanneer later instructies aan een voorwaardelijke instructie zonder accolades worden toegevoegd. In de volgende code neemt de waarde van `positiveNums` bijvoorbeeld met 1 toe of de voorwaarde nu wel of niet resulteert in `true`:

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

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

trace(positiveNums); // 1
```

switch

De instructie `switch` is nuttig als u verschillende uitvoeringspaden hebt die afhankelijk zijn van dezelfde voorwaardelijke expressie. Deze instructie biedt een zelfde soort functionaliteit als een lange reeks instructies `if..else if` achter elkaar, maar is beter leesbaar. In plaats van het testen van een voorwaarde op een Booleaanse waarde, evalueert de instructie `switch` een expressie en gebruikt het resultaat om te bepalen welk codeblok moet worden uitgevoerd. Codeblokken beginnen met een instructie `case` en eindigen met een instructie `break`. De volgende instructie `switch` resulteert bijvoorbeeld in de dag van de week, op basis van het dagnummer dat de methode `Date.getDay()` retourneert:

```
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;
}
```

Herhaling

Met lusinstructies kunt u een specifiek codeblok herhaaldelijk uitvoeren met een reeks waarden of variabelen. Adobe geeft de aanbeveling dat u het codeblok altijd tussen accolades ({}) plaatst. U kunt de accolades weglaten als het codeblok maar één instructie bevat, maar dit wordt niet aanbevolen (om dezelfde reden als voor voorwaardelijke instructies): de toenemende kans dat later toegevoegde instructies ongewild worden buitengesloten van het codeblok. Als u later een instructie toevoegt die u in het codeblok wilt opnemen maar vergeet accolades te plaatsen, wordt de instructie niet als onderdeel van de lus uitgevoerd.

for

Met de lus `for` kunt u een variabele doorlopen voor een opgegeven reeks waarden. U moet drie expressies in een instructie `for` opgeven: een variabele die is ingesteld op een beginwaarde, een voorwaardelijke instructie die bepaalt wanneer de herhalingen eindigen en een expressie die bij elke lus de waarde wijzigt van een variabele. De volgende code wordt bijvoorbeeld vijf keer herhaald: De waarde van de variabele `i` begint bij 0 en eindigt bij 4. De uitvoer zijn de getallen 0 tot en met 4, die elk in een afzonderlijke regel worden weergegeven.

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

for..in

Met de lus `for..in` doorloopt u de eigenschappen van een object, of de elementen van een array. U kunt bijvoorbeeld een lus `for..in` gebruiken om de eigenschappen van een algemeen object te doorlopen (objecteigenschappen worden niet in een bepaalde volgorde bewaard, zodat eigenschappen in schijnbaar willekeurige volgorde worden weergegeven):

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

U kunt ook de elementen van een array doorlopen:

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

U kunt de eigenschappen van een object niet doorlopen als dit een instantie van een verzegelde klasse is (inclusief ingebouwde klassen en door de gebruiker gedefinieerde klassen). U kunt alleen eigenschappen van een dynamische klasse doorlopen. Zelfs bij instanties van dynamische klassen, kunt u alleen eigenschappen doorlopen die dynamisch zijn toegevoegd.

for each..in

Met de lus `for each..in` doorloopt u de items van een verzameling. Dit kunnen tags in een object XML of XMLList zijn, de waarden van objecteigenschappen of de elementen van een array. Het volgende voorbeeldfragment laat zien dat u met een lus `for each..in` de eigenschappen van een algemeen object kunt doorlopen. Anders dan bij de lus `for..in` bevat de variabele voor de iterator in een lus `for each..in` echter de waarde van de eigenschap in plaats van de naam van de eigenschap:

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

U kunt een object XML of XMLList doorlopen, zoals in het volgende voorbeeld wordt getoond:

```
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
*/
```

U kunt ook de elementen van een array doorlopen, zoals dit voorbeeld laat zien:

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

U kunt niet de eigenschappen van een object doorlopen wanneer het object een instantie van een verzegelde klasse is. Zelfs voor instanties van dynamische klassen kunt u vaste eigenschappen (eigenschappen die zijn gedefinieerd als onderdeel van de klassendefinitie) niet doorlopen.

while

De lus `while` lijkt op een instructie `if` die zich herhaalt zolang de voorwaarde `true` is. De volgende code levert bijvoorbeeld hetzelfde resultaat op als het voorbeeld met de lus `for`:

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

Het gebruik van een lus `while` in plaats van een lus `for` heeft als nadeel dat met de lus `while` heel makkelijk oneindige lussen kunnen worden gemaakt. Het codevoorbeeld van de lus `for` wordt niet gecompileerd als u de expressie weglaat die de tellervariabele verhoogt, maar de voorbeeldlus `while` wordt wel gecompileerd als u die stap weglaat. Zonder het gebruik van de expressie die `i` verhoogt, wordt de lus een oneindige lus.

do..while

De lus `do..while` is een lus `while` die garandeert dat het codeblok minstens één keer wordt uitgevoerd, omdat de voorwaarde wordt gecontroleerd nadat het codeblok is uitgevoerd. De volgende code is een eenvoudig voorbeeld van een lus `do..while` die zelfs uitvoer genereert wanneer niet aan de voorwaarde is voldaan:

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

Functies

Functies zijn blokken code die specifieke taken uitvoeren en opnieuw kunnen worden gebruikt in uw programma. Er zijn twee typen functies in ActionScript 3.0: *methoden* en *functie closure*. Of een functie een methode of functie closure wordt genoemd, hangt af van de context waarin de functie is gedefinieerd. Een functie wordt een methode genoemd als u deze definieert als onderdeel van een klassendefinitie of koppelt aan een instantie van een object. In alle andere gevallen is er sprake van een functie closure.

Functies hebben altijd een uiterst belangrijke rol gespeeld in ActionScript. In ActionScript 1.0 bestond het trefwoord `class` nog niet, zodat 'klassen' werden gedefinieerd met constructorfuncties. Hoewel het trefwoord `class` later in de taal is opgenomen, is een goed begrip van functies nog steeds belangrijk als u de taal optimaal wilt gebruiken. Dit kan een uitdaging vormen voor programmeurs die verwachten dat functies in ActionScript zich net zo gedragen als functies in talen zoals C++ of Java. Hoewel de basisbewerkingen voor het definiëren en aanroepen van functies voor ervaren programmeurs geen probleem hoeft te zijn, is voor sommige meer geavanceerde mogelijkheden van ActionScript enige uitleg vereist.

Elementaire functieconcepten

Functies aanroepen

U roept een functie aan door de id ervan te gebruiken, gevolgd door de ronde-haakjesoperator (`()`). Tussen de ronde haakjes plaatst u de functieparameters die u naar de functie wilt sturen. De functie `trace()` is bijvoorbeeld een functie op topniveau in ActionScript 3.0:

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

Als u een functie aanroept zonder parameters, gebruikt u een leeg paar ronde haakjes. Met de methode `Math.random()`, waarvoor geen parameters worden gebruikt, genereert u bijvoorbeeld een willekeurig getal:

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

Uw eigen functies definiëren

Er zijn twee manieren om een functie te definiëren in ActionScript 3.0: u kunt een functie-instructie of een functie-expressie gebruiken. De techniek die u kiest, hangt af van uw eigen voorkeur voor een meer statische of dynamische programmeerstijl. Definieer uw functies met functie-instructies als u de voorkeur geeft aan statisch programmeren (in de strikte modus). Definieer uw functies met functie-expressies als u daarvoor een specifieke reden hebt. Functie-expressies worden vaker gebruikt bij dynamisch programmeren (in de standaardmodus).

Functie-instructies

Functie-instructies genieten de voorkeur voor het definiëren van functies in de strikte modus. Een functie-instructie begint met het trefwoord `function`, gevolgd door:

- De functienaam

- De parameters, in een door komma's gescheiden lijst tussen ronde haakjes
- De hoofdtekst van de functie, ofwel de ActionScript-code die wordt uitgevoerd bij het aanroepen van de functie, tussen accolades

Met de volgende code wordt bijvoorbeeld een functie gemaakt die een parameter definieert en wordt vervolgens de functie aangeroepen met de tekenreeks 'hello' als parameterwaarde:

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

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

Functie-expressies

De tweede manier om een functie te declareren is met behulp van een toewijzingsinstructie en een functie-expressie, ook wel een letterlijke functie of anonieme functie genoemd. Deze manier leidt tot uitgebreidere code en werd in eerdere versies van ActionScript veel gebruikt.

Een toewijzingsinstructie met een functie-expressie begint met het trefwoord `var`, gevolgd door:

- De functienaam
- De operator `:` (dubbele punt)
- De klasse `Function` om het gegevenstype aan te duiden
- De toewijzingsoperator (=)
- Het trefwoord `function`
- De parameters, in een door komma's gescheiden lijst tussen ronde haakjes
- De hoofdtekst van de functie, ofwel de ActionScript-code die wordt uitgevoerd bij het aanroepen van de functie, tussen accolades

  De volgende code declareert bijvoorbeeld de functie `traceParameter` met behulp van een functie-expressie:

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

  Merk op dat u geen functienaam opgeeft, zoals bij een functie-instructie. Nog een belangrijk verschil is dat functie-expressies anders dan functie-instructies geen instructies zijn, maar expressies. Dit betekent dat een functie-expressie niet afzonderlijk kan bestaan en een functie-instructie wel. Een functie-expressie kan alleen als onderdeel van een instructie worden gebruikt, meestal een toewijzingsinstructie. In het volgende voorbeeld wordt een functie-expressie toegewezen aan een arrayelement:

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

Kiezen tussen instructies en expressies

Als algemene regel gebruikt u een functie-instructie tenzij specifieke omstandigheden het gebruik van een expressie vereisen. Functie-instructies zijn minder uitgebreid en zorgen voor meer consistentie bij het gebruik in zowel de strikte modus als de standaardmodus dan functie-expressies.

Functie-instructies zijn beter leesbaar dan toewijzingsinstructies met functie-expressies. Functie-instructies maken de code compacter en zijn minder verwarrend dan functie-expressies, waarbij u zowel het trefwoord `var` als `function` moet gebruiken.

Functie-instructies zorgen voor meer consistentie in de twee compilermodi. Zo kunt u de puntnotatie gebruiken in zowel de strikte modus als de standaardmodus om een methode aan te roepen die wordt gedeclareerd met een functie-instructie. Dit is niet altijd mogelijk voor methoden die worden gedeclareerd met een functie-expressie. Met de volgende code wordt bijvoorbeeld een klasse gedefinieerd met de naam Example met twee methoden: `methodExpression()`, gedeclareerd met een functie-expressie, en `methodStatement()`, gedeclareerd met een functie-instructie. In de strikte modus kunt u geen puntnotatie gebruiken om de methode `methodExpression()` aan te roepen.

```
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
```

Functie-expressies zijn over het algemeen beter geschikt voor programmeerdoeleinden die georiënteerd zijn op dynamisch gedrag bij uitvoering. Als u in de strikte modus wilt werken maar ook een methode moet aanroepen die wordt gedeclareerd met een functie-expressie, kunt u een van de volgende twee technieken toepassen. Met de eerste techniek roept u de methode aan met behulp van vierkante haakjes (`[]`) in plaats van de puntoperator (`.`). De volgende methodeaanroep is mogelijk in zowel de strikte modus als de standaardmodus:

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

Met de tweede techniek declareert u de gehele klasse als dynamische klasse. Hoewel u de methode zo wel kunt aanroepen met de puntoperator, gaat dit ten koste van enige functionaliteit van de strikte modus voor alle instanties van die klasse. De compiler genereert bijvoorbeeld geen fout wanneer u toegang probeert te krijgen tot een ongedefinieerde eigenschap voor een instantie van een dynamische klasse.

Er zijn enkele omstandigheden waarin functie-expressies nuttig zijn. Functie-expressies worden vaak gebruikt voor functies die maar eenmalig worden toegepast. Minder vaak worden ze gebruikt om een functie te koppelen aan een prototype-eigenschap. Zie Prototypeobject voor meer informatie.

Er zijn twee subtiele verschillen tussen functie-instructies en functie-expressies waarmee u rekening moet houden wanneer u kiest welke techniek u gebruikt. Het eerste verschil is dat functie-expressies niet als onafhankelijke objecten bestaan met betrekking tot geheugenbeheer en opschonen. Met andere woorden, wanneer u een functie-expressie toewijst aan een ander object, zoals een arrayelement of een objecteigenschap, maakt u de enige verwijzing naar die functie in de code. Als de array of het object waaraan de functie-expressie is gekoppeld buiten het bereik valt of om een andere reden niet meer beschikbaar is, hebt u geen toegang meer tot de functie-expressie. Als de array of het object wordt verwijderd, komt het geheugen van de functie-expressie in aanmerking voor opschonen ofwel beschikbaar voor andere doeleinden.

Het volgende voorbeeld laat zien dat voor een functie-expressie na het verwijderen van de eigenschap waaraan de expressie is toegewezen, de functie niet meer beschikbaar is. De klasse Test is van het type dynamic, wat betekent dat u een eigenschap met de naam `functionExp` kunt toevoegen met een functie-expressie. De functie `functionExp()` kan worden aangeroepen met de puntoperator, maar nadat de eigenschap `functionExp` is verwijderd, is de functie niet meer beschikbaar.

```
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
```

Als de functie eerst wordt gedefinieerd met een functie-instructie, bestaat deze als een eigen object, ook als de eigenschap wordt verwijderd waaraan de functie is gekoppeld. De operator `delete` werkt alleen voor eigenschappen van objecten. Een aanroep om de functie `stateFunc()` zelf te verwijderen, werkt dus niet.

```
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
```

Het tweede verschil tussen functie-instructies en functie-expressies is dat functie-instructies bestaan in het gehele bereik waarin ze worden gedefinieerd, ook in instructies die vóór de functie-instructie staan. Functie-expressies daarentegen worden alleen gedefinieerd voor daaropvolgende instructies. In de volgende code wordt de functie `scopeTest()` bijvoorbeeld aangeroepen voordat deze is gedefinieerd:

```
statementTest(); // statementTest

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

Functie-expressies zijn niet beschikbaar voordat ze worden gedefinieerd. De volgende code resulteert dan ook in een uitvoeringsfout:

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

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

Waarden retourneren uit functies

Als u een waarde wilt retourneren uit een functie, gebruikt u de instructie `return` gevolgd door de expressie of letterlijke waarde die u wilt laten retourneren. De volgende code retourneert bijvoorbeeld een expressie die de parameter vertegenwoordigt:

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

U ziet dat de instructie `return` de functie beëindigt, zodat instructies onder een instructie `return` niet worden uitgevoerd:

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

In de strikte modus moet u een waarde van het geschikte type retourneren als u ervoor kiest een retourneringstype op te geven. De volgende code genereert bijvoorbeeld een fout in de strikte modus, omdat geen geldige waarde wordt geretourneerd:

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

Geneste functies

U kunt functies nesten, wat betekent dat u functies kunt declareren binnen andere functies. Een geneste functie is alleen beschikbaar binnen de bovenliggende functie, tenzij een verwijzing naar de functie wordt doorgegeven naar externe code. De volgende code declareert bijvoorbeeld twee geneste functies binnen de functie `getNameAndVersion()`:

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

Wanneer geneste functies worden doorgegeven aan externe code, gebeurt dit als een functie closure. Dit betekent dat de functie de definities behoudt die binnen het bereik vallen wanneer de functie wordt gedefinieerd. Zie Functiebereik voor meer informatie.

Functieparameters

ActionScript 3.0 biedt enige functionaliteit voor functieparameters die mogelijk nieuw lijkt voor programmeurs zonder ervaring in deze taal. Hoewel het idee van parameters doorgeven als waarde of als verwijzing bij de meeste programmeurs bekend zal zijn, geldt dat mogelijk niet voor het object `arguments` en de parameter ... (rest).

Argumenten doorgeven als waarde of als verwijzing

In veel programmeertalen is het van belang het onderscheid te weten tussen het doorgeven van argumenten als waarde of als verwijzing. Dit onderscheid kan van invloed zijn op de wijze waarop code wordt samengesteld.

Doorgeven als waarde betekent dat de waarde van het argument wordt gekopieerd naar een lokale variabele om in de functie te gebruiken. Doorgeven als verwijzing betekent dat alleen een verwijzing naar het argument wordt doorgegeven en niet de feitelijke waarde. Er wordt geen kopie van het eigenlijke argument gemaakt. In plaats daarvan wordt een verwijzing naar de variabele doorgegeven als een argument wordt gemaakt en toegewezen aan een lokale variabele voor gebruik in de functie. Als verwijzing naar een variabele buiten de functie geeft de lokale variabele u de mogelijkheid om de waarde van de oorspronkelijke variabele te wijzigen.

In ActionScript 3.0 worden alle argumenten doorgegeven als verwijzing, omdat alle waarden als objecten worden opgeslagen. Objecten die behoren tot de primitieve gegevenstypen (Boolean, Number, int, uint en String) hebben speciale operatoren waardoor ze gedrag vertonen alsof ze als waarde zijn doorgegeven. Met de volgende code wordt bijvoorbeeld een functie gemaakt met de naam `passPrimitives()` die twee parameters definieert met de naam `xParam` en `yParam`, beide van het type int. Deze parameters lijken op lokale variabelen die worden gedeclareerd in de hoofdtekst van de functie `passPrimitives()`. Wanneer de functie wordt aangeroepen met de argumenten `xValue` en `yValue`, worden de parameters `xParam` en `yParam` geïnitialiseerd met verwijzingen naar de objecten int vertegenwoordigd door `xValue` en `yValue`. Omdat de argumenten primitieven zijn, gedragen deze zich alsof ze als waarde zijn doorgegeven. Hoewel `xParam` en `yParam` eerst alleen verwijzingen bevatten naar de objecten `xValue` en `yValue`, genereren wijzigingen in de variabelen in de hoofdtekst van de functie nieuwe kopieën van de waarden in het geheugen.

```
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
```

Binnen de functie `passPrimitives()` worden de waarden van `xParam` en `yParam` verhoogd. Dit heeft echter geen invloed op de waarden van `xValue` en `yValue`, zoals wordt getoond in de laatste instructie `trace`. Dit geldt ook als de parameters dezelfde naam zouden hebben als de variabelen, `xValue` en `yValue`, omdat de `xValue` en `yValue` binnen de functie naar nieuwe locaties in het geheugen zouden wijzen die afzonderlijk bestaan van de variabelen met dezelfde naam buiten de functie.

Alle andere objecten (die niet behoren tot de primitieve gegevenstypen) worden altijd doorgegeven als verwijzing. Dit maakt het mogelijk de waarde van de oorspronkelijke variabele te wijzigen. Met de volgende code wordt bijvoorbeeld een object gemaakt met de naam `objVar` met twee eigenschappen: `x` en `y`. Het object wordt als argument aan de functie `passByRef()` doorgegeven. Aangezien het object geen primitief type is, wordt het object niet alleen als verwijzing doorgegeven, maar blijft het ook een verwijzing. Dit betekent dat wijzigingen in parameters binnen de functie van invloed zijn op de objecteigenschappen buiten de functie.

```
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
```

De parameter `objParam` verwijst naar hetzelfde object als de algemene variabele `objVar`. Zoals u kunt zien in de instructies `trace` in het voorbeeld, worden wijzigingen in de eigenschappen `x` en `y` van het object `objParam` weergegeven in het object `objVar`.

Standaardparameterwaarden

In ActionScript 3.0 kunt u voor een functie *standaardparameterwaarden* declareren. Als bij het aanroepen van een functie met standaardparameterwaarden een parameter met standaardwaarden wordt weggelaten, wordt de opgegeven waarde in de functiedefinitie voor die parameter gebruikt. Alle parameters met standaardwaarden moeten aan het eind van de lijst met parameters worden geplaatst. De toegewezen standaardwaarden moeten constanten bij compileren zijn. Het bestaan van een standaardwaarde voor een parameter maakt van die parameter feitelijk een *optionele parameter*. Een parameter zonder standaardwaarde wordt als *vereiste parameter* beschouwd.

Met de volgende code wordt bijvoorbeeld een functie met drie parameters gemaakt, waarvan er twee een standaardwaarde hebben. Wanneer de functie wordt aangeroepen met slechts één parameter, worden de standaardwaarden voor de parameters gebruikt.

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

Het object arguments

Wanneer parameters worden doorgegeven aan een functie, kunt u het object `arguments` gebruiken voor toegang tot informatie over de parameters die aan de functie worden doorgegeven. Sommige belangrijke aspecten van het object `arguments` zijn onder meer:

- Het object `arguments` is een array met alle parameters die aan de functie worden doorgegeven.
- De eigenschap `arguments.length` rapporteert het aantal parameters dat aan de functie wordt doorgegeven.
- De eigenschap `arguments.callee` zorgt voor een verwijzing naar de functie zelf, wat nuttig is voor recursieve aanroepen van functie-expressies.

  Opmerking: *Het object `arguments` is niet beschikbaar als een parameter de naam `arguments` heeft of als u de parameter ... (rest) gebruikt.*

  Als vanuit het hoofdgedeelte van een functie wordt verwezen naar het object `arguments`, kunnen functieoproepen in ActionScript 3.0 meer parameters bevatten dan in de functiedefinitie zijn gedefinieerd. In de strikte modus wordt echter een compilatiefout gegenereerd als het aantal parameters niet overeenkomt met het aantal vereiste parameters (en desgewenst eventuele optionele parameters). U kunt het arrayaspect van het object `arguments` gebruiken voor toegang tot parameters die aan de functie worden doorgegeven, of de parameter nu is gedefinieerd in de functiedefinitie of niet. In het volgende voorbeeld, dat alleen in de standaardmodus kan worden gecompileerd, wordt de array `arguments` gebruikt samen met de eigenschap `arguments.length` om alle parameters te traceren die aan de functie `traceArgArray()` worden doorgegeven:

```
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
```

De eigenschap `arguments.callee` wordt vaak gebruikt in anonieme functies om herhaling te bewerkstelligen. U kunt daarmee flexibiliteit toevoegen aan uw code. Als de naam van een recursieve functie verandert gedurende de ontwikkelingscyclus, hoeft u zich geen zorgen te maken om de recursieve aanroep in de hoofdtekst van de functie te wijzigen als u `arguments.callee` gebruikt in plaats van de functienaam. De eigenschap `arguments.callee` wordt gebruikt in de volgende functie om herhaling mogelijk te maken:

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

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

Als u de parameter ... (rest) gebruikt in uw functiedeclaratie, is het object `arguments` voor u niet beschikbaar. In plaats daarvan hebt u toegang tot de parameters met de parameternamen die u daarvoor declareert.

Zorg er ook voor dat u de tekenreeks `'arguments'` niet gebruikt als parameternaam, wat als schaduw van het object `arguments` zou werken. Als de functie `traceArgArray()` bijvoorbeeld wordt herschreven zodat een parameter `arguments` wordt toegevoegd, verwijzen de verwijzingen naar `arguments` in de hoofdtekst van de functie naar de parameter in plaats van naar het object `arguments`. De volgende code produceert geen uitvoer:

```
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
```

Het object `arguments` in eerdere versies van ActionScript bevatte ook een eigenschap met de naam `caller`, ofwel een verwijzing naar de functie die de huidige functie aanroept. De eigenschap `caller` wordt niet meer gebruikt in ActionScript 3.0. Als u echter moet verwijzen naar de aanroepende functie, kunt u de aanroepende functie een extra parameter laten doorgeven als verwijzing naar de functie zelf.

De parameter ... (rest)

ActionScript 3.0 introduceert een nieuwe parameterdeclaratie: de parameter ... (rest). Met deze parameter kunt u een arrayparameter opgeven die een willekeurig aantal door komma's gescheiden argumenten accepteert. De parameter kan elke naam hebben die geen gereserveerd woord is. Deze parameterdeclaratie moet de laatste parameter zijn die wordt opgegeven. Wanneer u deze parameter gebruikt, is het object `arguments` niet beschikbaar. Hoewel de parameter ... (rest) dezelfde functionaliteit biedt als de array `arguments` en de eigenschap `arguments.length`, ontbreekt de functionaliteit die `arguments.callee` biedt. U moet ervoor zorgen dat u `arguments.callee` niet hoeft te gebruiken voordat u de parameter ... (rest) gebruikt.

In het volgende voorbeeld wordt de functie `traceArgArray()` herschreven met de parameter ... (rest) in plaats van het object `arguments`:

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

traceArgArray(1, 2, 3);

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

U kunt de parameter ... (rest) (als laatste parameter) ook gebruiken met andere parameters. In het volgende voorbeeld wordt de functie `traceArgArray()` zo gewijzigd dat de eerste parameter, `x`, van het type int is en de tweede parameter de parameter ... (rest) gebruikt. Bij de uitvoer wordt de eerste waarde overgeslagen, omdat de eerste parameter geen onderdeel meer vormt van de array die met de parameter ... (rest) wordt gemaakt.

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

traceArgArray(1, 2, 3);

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

Functies als objecten

Functies in ActionScript 3.0 zijn objecten. Wanneer u een functie maakt, maakt u een object dat als parameter aan een andere functie kan worden doorgegeven én waaraan eigenschappen en methoden kunnen worden gekoppeld.

Het doorgeven van functies als argumenten aan een andere functie gebeurt als verwijzing en niet als waarde. Wanneer u een functie doorgeeft als argument, gebruikt u alleen de id en niet de ronde-haakjesoperator waarmee u de methode aanroept. In de volgende code wordt bijvoorbeeld een functie met de naam `clickListener()` doorgegeven als argument aan de methode `addEventListener()`:

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

Hoewel dit voor programmeurs zonder ervaring in ActionScript wat vreemd kan lijken, kunnen functies eigenschappen en methoden hebben, net als andere objecten. In feite heeft elke functie een alleen-lezen eigenschap met de naam `length` waarin het aantal gedefinieerde parameters voor de functie wordt opgeslagen. Dit verschilt van de eigenschap `arguments.length`, die het aantal argumenten aangeeft dat aan de functie wordt doorgegeven. Zoals al eerder is opgemerkt, kan in ActionScript het aantal doorgegeven argumenten aan een functie hoger zijn dan het aantal gedefinieerde parameters voor die functie. In het volgende voorbeeld wordt het verschil getoond tussen de twee eigenschappen (compilatie kan alleen in de standaardmodus, omdat voor de strikte modus is vereist dat het aantal doorgegeven argumenten en het aantal gedefinieerde parameters exact gelijk zijn):

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

In de standaardmodus kunt u uw eigen functie-eigenschappen definiëren buiten de hoofdtekst van de functie. U kunt functie-eigenschappen als quasi-statische eigenschappen gebruiken, zodat u de status van een variabele in relatie tot de functie kunt opslaan. U wilt bijvoorbeeld het aantal keren bijhouden dat een bepaalde functie wordt aangeroepen. Een dergelijke functionaliteit kan nuttig zijn als u een spel ontwikkelt en wilt bijhouden hoe vaak een gebruiker een bepaalde opdracht gebruikt, hoewel u daar ook een statische klasseneigenschap voor kunt gebruiken. Met de volgende code wordt een functie-eigenschap gemaakt buiten de functiedeclaratie en wordt de eigenschap verhoogd telkens wanneer de functie wordt aangeroepen. Dit voorbeeld kan alleen worden gecompileerd in de standaardmodus omdat het in de strikte modus niet mogelijk is dynamische eigenschappen toe te voegen aan functies.

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

someFunction.counter = 0;

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

Functiebereik

Het bereik van een functie bepaalt waar in een programma die functie kan worden aangeroepen en tot welke definities de functie toegang heeft. Dezelfde bereikregels die gelden voor variabele-id's gelden ook voor functie-id's. Een functie die wordt gedeclareerd in het algemene bereik, is in de gehele code beschikbaar. ActionScript 3.0 bevat bijvoorbeeld algemene functies, zoals `isNaN()` en `parseInt()`, die overal in de code beschikbaar zijn. Een geneste functie (gedeclareerd binnen een andere functie) kan overal in de functie worden gebruikt waarin het is gedeclareerd.

De bereikketen

Aan het begin wanneer een functie wordt uitgevoerd, wordt een aantal objecten en eigenschappen gemaakt. Eerst wordt een speciaal *activeringsobject* gemaakt met de parameters en de lokale variabelen of functies die in de hoofdtekst van de functie zijn gedeclareerd. U hebt geen directe toegang tot het activeringsobject, omdat het een intern mechanisme betreft. Vervolgens wordt een *bereikketen* gemaakt met een geordende lijst met objecten die tijdens de uitvoering wordt gecontroleerd op id-declaraties. Elke uitgevoerde functie heeft een bereikketen die in een interne eigenschap wordt opgeslagen. Voor een geneste functie begint de bereikketen met het eigen activeringsobject, gevolgd door het activeringsobject van de bovenliggende functie. De keten gaat op deze wijze verder tot het algemene object is bereikt. Het algemene object wordt gemaakt wanneer een ActionScript-programma begint en dit object bevat alle algemene variabelen en functies.

Functie closure

Een *functie closure* is een object met een momentopname van een functie en de bijbehorende *lexicale omgeving*. De lexicale omgeving van een functie omvat alle variabelen, eigenschappen, methoden en objecten in de bereikketen van de functie, samen met de waarden ervan. Een functie closure wordt gemaakt telkens wanneer een functie wordt uitgevoerd apart van een object of een klasse. Het feit dat een functie closure het bereik behoudt waarin deze is gedefinieerd, levert interessante resultaten op wanneer een functie als argument of geretourneerde waarde in een ander bereik wordt doorgegeven.

Met de volgende code worden bijvoorbeeld twee functies gemaakt: `foo()`, die een geneste functie retourneert met de naam `rectArea()` waarmee de oppervlakte van een rechthoek wordt berekend, en `bar()`, die `foo()` aanroept en de geretourneerde functie closure opslaat in een variabele met de naam `myProduct`. Ook al definieert de functie `bar()` de eigen lokale variabele `x` (met een waarde van 2), wordt bij het aanroepen van de functie closure `myProduct()` de variabele `x` behouden (met een waarde van 40) die is gedefinieerd in de functie `foo()`. De functie `bar()` retourneert dan ook de waarde `160` in plaats van `8`.

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

Methoden vertonen hetzelfde gedrag wat betreft het behouden van informatie over de lexicale omgeving waarin ze zijn gemaakt. Dit valt het meest op wanneer een methode wordt geëxtraheerd uit de instantie ervan, wat resulteert in een gebonden methode. Het belangrijkste verschil tussen een functie closure en een gebonden methode is dat de waarde van het trefwoord `this` in een gebonden methode altijd verwijst naar de instantie waaraan deze eerst was gekoppeld, terwijl in een functie closure de waarde van het trefwoord `this` kan veranderen.

Hoofdstuk 4: Objectgeoriënteerd programmeren in ActionScript

Inleiding tot objectgeoriënteerd programmeren

Objectgeoriënteerd programmeren (OOP) is een manier om de code in een programma te organiseren door deze te groeperen in objecten. De term *Object* betekent in dit geval een individueel element dat informatie (gegevenswaarden) en functionaliteit bevat. Wanneer u een objectgeoriënteerde aanpak gebruikt om een programma te organiseren, groepeert u bepaalde delen informatie met een gemeenschappelijke functionaliteit of acties die gerelateerd zijn aan die informatie. U kunt bijvoorbeeld muziekinformatie zoals albumtitel, tracktitel of artiestennaam groeperen met functionaliteiten zoals 'track aan afspeellijst toevoegen' of 'alle liedjes van deze artiest afspelen'. Deze delen worden in een enkel item gecombineerd: een object (bijvoorbeeld 'Album' of 'Muzieknummer'). Er zijn meerdere voordelen verbonden aan het samenvoegen van waarden en functies. Een belangrijk voordeel is dat u slechts één variabele hoeft te gebruiken. Het houdt gerelateerde functionaliteiten samen. Dankzij de combinatie van informatie en functionaliteit kunt u programma's structureren op een manier die beter overeenkomt met de realiteit.

Klassen

Een klasse is een abstracte representatie van een object. Een klasse slaat informatie op over het type gegevens dat een object kan bevatten en het gedrag dat een object kan vertonen. Het nut van een dergelijke abstractie is mogelijk niet duidelijk wanneer u kleine scripts schrijft die slechts een paar met elkaar communicerende objecten bevatten. Als het bereik van een programma groeit, neemt het aantal objecten dat beheerd moet worden toe. In dat geval kunt u met behulp van klassen beter bepalen hoe objecten worden gemaakt en hoe ze met elkaar communiceren.

In het verouderde ActionScript 1.0 konden ActionScript-ontwikkelaars objecten Function gebruiken om constructies te maken die op klassen leken. In ActionScript 2.0 werd formele ondersteuning toegevoegd voor klassen met trefwoorden zoals `class` en `extends`. ActionScript 3.0 blijft de trefwoorden van ActionScript 2.0 ondersteunen en voegt nieuwe mogelijkheden toe. ActionScript 3.0 bevat bijvoorbeeld een verbeterde toegangscontrole met de attributen `protected` en `internal`. Dit biedt ook een betere controle over overerving met de trefwoorden `final` en `Negeren`.

Voor ontwikkelaars die klassen hebben gemaakt in programmeertalen zoals Java, C++ of C# biedt ActionScript een bekende ervaring. ActionScript heeft veel dezelfde trefwoorden en attribuutnamen, zoals `class`, `extends` en `public`.

Opmerking: *In de Adobe ActionScript-documentatie staat de term eigenschap voor een lid van een object of klasse, inclusief variabelen, constanten en methoden. Hoewel vaak geen onderscheid wordt gemaakt tussen de termen class en static, hebben ze hier verschillende betekenissen. In dit voorbeeld verwijst de woordgroep 'eigenschappen van een klasse' bijvoorbeeld naar alle leden van een klasse, in plaats van alleen naar leden van het type static.*

Klassendefinities

Klassendefinities in ActionScript 3.0 gebruiken syntaxis die overeenkomt met de syntaxis die werd gebruikt voor klassendefinities in ActionScript 2.0. De juiste syntaxis voor een klassendefinitie roept het trefwoord `class` aan, gevolgd door de naam van de klasse. De hoofdtekst van de klasse, die door accolades ({}) wordt ingesloten, volgt de naam van de klasse. De volgende code maakt bijvoorbeeld een klasse met de naam Shape die een variabele bevat, met de naam `visible`:

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

Een belangrijke syntaxiswijziging betreft de klassendefinities in een pakket. Als in ActionScript 2.0 een klasse zich in een pakket bevindt, moet de pakketnaam in de klassendeclaratie zijn opgenomen. In ActionScript 3.0 is de instructie `package` geïntroduceerd en moet de pakketnaam in de pakketdeclaratie zijn opgenomen in plaats van in de klassendeclaratie. De volgende klassendeclaraties tonen bijvoorbeeld hoe de klasse BitmapData, deel van het pakket flash.display, in ActionScript 2.0 en ActionScript 3.0 wordt gedefinieerd:

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

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

Klassenattributen

Met ActionScript 3.0 kunt u klassendefinities met een van de volgende vier attributen bewerken:

attribuut	Definitie
`dynamic`	Hiermee kunnen eigenschappen bij uitvoering aan instanties worden toegevoegd.
`final`	Mag niet door een andere klasse worden uitgebreid.
`internal` (standaard)	Zichtbaar voor verwijzingen binnen het huidige pakket.
`public`	Zichtbaar voor alle verwijzingen.

Voor elk van deze attributen, behalve `internal`, moet u expliciet het attribuut opnemen om het gekoppelde gedrag op te halen. Als u bijvoorbeeld niet het attribuut `dynamic` opneemt wanneer u een klasse definieert, kunt u bij uitvoering geen eigenschappen aan een klasseninstantie toevoegen. U wijst een attribuut expliciet toe door deze aan het begin van de klassendefinitie te plaatsen, zoals getoond in de volgende code:

```
dynamic class Shape {}
```

In de lijst is het attribuut `abstract` niet opgenomen. Abstracte klassen worden niet ondersteund in ActionScript 3.0. Verder bevat de lijst ook geen attributen `private` en `protected`. Deze attributen zijn alleen belangrijk in een klassendefinitie en kunnen zelf niet op klassen worden toegepast. Als u niet wilt dat een klasse buiten een pakket zichtbaar is voor gebruikers, plaatst u de klasse in een pakket en markeert u de klasse met het attribuut `internal`. U kunt ook de attributen `internal` en `public` weglaten. De compiler voegt in dat geval automatisch het attribuut `internal` toe. U kunt ook een klasse definiëren zodat deze alleen zichtbaar is binnen het gedefinieerde bronbestand. Plaats de klasse onderaan in uw bronbestand, onder de accolade sluiten van de pakketdefinitie.

Hoofdtekst van de klasse

De hoofdtekst van de klasse is ingesloten door accolades. Hierin worden de variabelen, constanten en methoden van uw klasse gedefinieerd. Het volgende voorbeeld geeft de declaratie weer van de Toegankelijkheidsklasse in ActionScript 3.0:

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

U kunt ook een naamruimte definiëren binnen de hoofdtekst van een klasse. Het volgende voorbeeld toont hoe een naamruimte binnen de hoofdtekst van een klasse kan worden gedefinieerd en gebruikt als een attribuut van een methode in die klasse:

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

Met ActionScript 3.0 kunt u niet alleen definities in de hoofdtekst van een klasse opnemen, maar ook instructies. Instructies in de hoofdtekst van een klasse maar buiten een methodedefinitie worden precies één keer uitgevoerd. Deze uitvoering vindt plaats wanneer de klassedefinitie voor het eerst wordt herkend en het gerelateerde klassenobject wordt gemaakt. Het volgende voorbeeld bevat een aanroep naar een externe functie `hello()` en een instructie `trace` die een bevestiging weergeeft wanneer de klasse is gedefinieerd:

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

In ActionScript 3.0 is het toegestaan om in dezelfde hoofdtekst van een klasse een statische eigenschap en een instantie-eigenschap met dezelfde naam te definiëren. De volgende code declareert bijvoorbeeld een variabele van het type static met de naam `message` en een instantievariabele met dezelfde naam:

```
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
```

Attributen voor klasseneigenschappen

In beschrijvingen van het ActionScript-objectmodel staat de term *eigenschap* voor alles wat een lid van een klasse kan zijn, inclusief variabelen, constanten en methoden. In de Naslaggids voor Adobe ActionScript 3.0 voor het Adobe Flash-platform wordt deze term strikter toegepast. In die context omvat de term Eigenschap alleen leden van klassen die variabelen zijn of worden gedefinieerd door een methode getter/setter. In ActionScript 3.0 hebt u een set attributen die voor elke eigenschap van een klasse kunnen worden gebruikt. De volgende tabel somt deze attributen op.

Attribuut	Definitie
`internal` (standaard)	Zichtbaar voor verwijzingen binnen hetzelfde pakket.
`private`	Zichtbaar voor verwijzingen in dezelfde klasse.
`protected`	Zichtbaar voor verwijzingen in dezelfde klasse en afgeleide klassen.
`public`	Zichtbaar voor alle verwijzingen.
`static`	Hiermee wordt opgegeven dat een eigenschap tot een klasse behoort, in tegenstelling tot de instanties van een klasse.
`UserDefinedNamespace`	Aangepaste naam voor naamruimte, die door de gebruiker is gedefinieerd.

Naamruimte-attributen voor toegangsbeheer

ActionScript 3.0 bevat vier speciale attributen die de toegang beheren tot eigenschappen die binnen een klasse zijn gedefinieerd: `public`, `private`, `protected` en `internal`.

Het attribuut `public` maakt een eigenschap in uw script zichtbaar. Als u bijvoorbeeld een methode buiten zijn pakket zichtbaar wilt maken voor code, moet u de methode met het attribuut `public` declareren. Dit gaat voor alle eigenschappen op, ongeacht of ze zijn gedeclareerd met het trefwoord `var`, `const` of `function`.

Het attribuut `private` maakt een eigenschap alleen zichtbaar voor aanroepen binnen de definiërende klasse van de eigenschap. Dit gedrag verschilt van het gedrag van het attribuut `private` in ActionScript 2.0, die een subklasse toegang bood tot een eigenschap van het type private in een superklasse. Een andere belangrijke gedragsverandering heeft te maken met toegang bij uitvoering. In ActionScript 2.0 werd met het trefwoord `private` alleen toegang toegestaan tijdens het compileren, wat bij uitvoering eenvoudig te omzeilen was. In ActionScript 3.0 is dit niet langer zo. Eigenschappen die zijn gemarkeerd als `private`, zijn zowel tijdens het compileren als bij uitvoering niet beschikbaar.

De volgende code maakt bijvoorbeeld een eenvoudige klasse met de naam PrivateExample met een variabele van het type private en probeert vervolgens de variabele van buitenaf de klasse te benaderen.

```
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 is het verkrijgen van toegang tot een eigenschap van het type private met gebruik van de puntoperator (`myExample.privVar`) resulteert in een fout bij compilatie als u de strikte modus gebruikt. Anders wordt de fout bij uitvoering gerapporteerd, op dezelfde manier als wanneer u de operator voor eigenschaptoegang (`myExample["privVar"]`) zou gebruiken.

De volgende tabel bevat de resultaten van een poging om een eigenschap van het type private te benaderen die tot een verzegelde (niet dynamische) klasse behoort:

	Strikte modus	Standaardmodus
puntoperator (`.`)	fout bij compilatie	uitvoeringsfout
haakjesoperator (`[]`)	uitvoeringsfout	uitvoeringsfout

Voor klassen die zijn gedeclareerd met het attribuut `dynamic`, resulteren pogingen tot het verkrijgen van toegang tot een variabele van het type private niet in een uitvoerfout. De variabele is in plaats daarvan niet zichtbaar, zodat het de waarde `undefined` oplevert. Er treedt echter een fout bij compilatie op als u de puntoperator in de strikte modus gebruikt. Het volgende voorbeeld is hetzelfde als het vorige voorbeeld, behalve dat de klasse PrivateExample als een klasse van het type dynamic is gedeclareerd:

```
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
```

Dynamische klassen retourneren doorgaans de waarde `undefined` in plaats van dat ze een fout genereren wanneer externe code toegang probeert te krijgen tot een eigenschap van het type private. De volgende tabel toont dat er alleen een fout wordt gegenereerd wanneer de puntoperator in de strikte modus wordt gebruikt om toegang tot een eigenschap van het type private te krijgen:

	Strikte modus	**Standaardmodus**
puntoperator (`.`)	fout bij compilatie	`undefined`
haakjesoperator (`[]`)	`undefined`	`undefined`

Het attribuut `protected`, nieuw in ActionScript 3.0, maakt een eigenschap zichtbaar voor aanroepers binnen zijn eigen klasse of in een subklasse. Een eigenschap van het type protected is met andere woorden binnen zijn eigen klasse beschikbaar of voor klassen die onder deze klasse in de overervingshiërarchie liggen. Dit is waar ongeacht of de subklasse zich in hetzelfde pakket bevindt of in een ander pakket.

Voor degenen die bekend zijn met ActionScript 2.0, is deze functionaliteit vergelijkbaar met het attribuut `private` in ActionScript 2.0. Het attribuut van ActionScript 3.0 `protected` is ook vergelijkbaar met het attribuut `protected` in Java. Het verschil met de Java-versie staat ook toegang toe aan oproepers met hetzelfde pakket. Het attribuut `protected` is nuttig wanneer u met een variabele of methode werkt die uw subklassen nodig hebben, maar u deze wilt verbergen voor code die geen deel uitmaakt van de overervingsketen.

Het attribuut `internal`, nieuw in ActionScript 3.0, maakt een eigenschap zichtbaar voor aanroepers binnen zijn eigen pakket. Dit is het standaardattribuut voor code binnen een pakket en geldt voor alle eigenschappen die niet een van de volgende attributen bevatten:

- `public`
- `private`
- `protected`
- een door een gebruiker gedefinieerde naamruimte

Het attribuut `internal` lijkt op het standaardtoegangsbeheer in Java, hoewel dit toegangsniveau in Java geen expliciete naam heeft en alleen wordt bereikt wanneer alle andere toegangwijzigingsopties worden weggelaten. Het attribuut `internal` is beschikbaar in ActionScript 3.0 zodat u expliciet kunt proberen een eigenschap alleen zichtbaar te maken voor aanroepers binnen zijn eigen pakket.

Het attribuut static

Met het attribuut `static`, dat kan worden gebruikt met de eigenschappen die zijn gedeclareerd met het trefwoord `var`, `const` of `function`, kunt u een eigenschap aan een klasse koppelen in plaats van aan de instantie van een klasse. Code die zich niet in de klasse bevindt, moet eigenschappen van het type static met de klassenaam in plaats van de instantienaam aanroepen.

Eigenschappen van het type static worden niet door subklassen overgeërfd, maar zijn deel van de bereikketen van de subklasse. Dit houdt in dat er in de hoofdtekst van een subklasse een variabele of methode van het type static kan worden gebruikt, zonder dat verwezen hoeft te worden naar de klasse waarin deze is gedefinieerd.

Door de gebruiker gedefinieerde naamruimte-attributen

Als alternatief voor de vooraf gedefinieerde attributen voor toegangsbeheer kunt u een aangepaste naamruimte maken die u als attribuut gebruikt. Er kan slechts één naamruimte-attribuut per definitie worden gebruikt en u kunt niet een naamruimte-attribuut in combinatie met een van de attributen voor toegangsbeheer (`public`, `private`, `protected`, `internal`) gebruiken.

Variabelen

Variabelen kunnen met de trefwoorden `var` of `const` worden gedeclareerd. De waarde van een variabele die wordt gedeclareerd met het trefwoord `var`, kan meerdere keren wijzigen tijdens het uitvoeren van het script. Variabelen die zijn gedeclareerd met het trefwoord `const`, zijn *constanten* en ze kunnen slechts eenmaal waarden toegewezen krijgen. Als u een nieuwe waarde aan een geïnitialiseerde constante probeert toe te wijzen, wordt een fout gegenereerd.

Variabelen van het type static

Variabelen van het type static worden gedeclareerd met gebruik van het trefwoord `static` en de instructie `var` of `const`. Variabelen van het type static, die aan een klasse zijn gekoppeld in plaats van aan een instantie van een klasse, zijn nuttig voor het opslaan en delen van informatie die van toepassing is op de gehele klasse van objecten. Een variabele van het type static is bijvoorbeeld nuttig als u het aantal keer dat een klasse wordt geïnstantieerd, wilt bijhouden of als u het maximale aantal toegestane klasseninstanties wilt opslaan.

Het volgende voorbeeld maakt een variabele `totalCount` om het aantal instanties van een klasse bij te houden en een constante `MAX_NUM` om het maximale aantal instanties op te slaan. De variabelen `totalCount` en `MAX_NUM` zijn statisch omdat ze alleen waarden bevatten die van toepassing zijn op de klasse als een geheel en niet op een bepaalde instantie.

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

Code die zich niet in de klasse StaticVars en een van zijn subklassen bevindt, kan alleen via de klasse zelf naar de eigenschappen `totalCount` en `MAX_NUM` verwijzen. De volgende code werkt bijvoorbeeld als volgt:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

U hebt via de instantie van de klasse geen toegang tot variabelen van het type static, daarom retourneert de code hieronder de volgende fout:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Variabelen die worden gedeclareerd met de trefwoorden `static` en `const`, moeten op hetzelfde moment worden geïnitialiseerd als dat de constante wordt gedeclareerd, zoals de klasse StaticVars dit ook doet voor `MAX_NUM`. U kunt binnen de constructor of een instantiemethode geen waarde toewijzen aan `MAX_NUM`. De volgende code genereert een fout, omdat dit geen geldige manier is om een statische constante te initialiseren:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Instantievariabelen

Instantievariabelen bevatten eigenschappen die zijn gedeclareerd met het trefwoord `var` en `const`, maar zonder het trefwoord `static`. Instantievariabelen, die aan klasseninstanties zijn gekoppeld in plaats van aan een gehele klasse, zijn nuttig voor het opslaan van waarden die specifiek zijn voor een instantie. De klasse Array bevat bijvoorbeeld een instantie-eigenschap met de naam `length` die het aantal arrayelementen opslaat dat een bepaalde instantie van een klasse Array bevat.

Instantievariabelen, ongeacht of ze zijn gedeclareerd als `var` of `const`, kunnen niet worden overschreven in een subklasse. Door de methode getter en setter te overschrijven kunt u echter een vergelijkbaar resultaat behalen.

Methoden

Methoden zijn functies die deel zijn van een klassendefinitie. Wanneer een instantie van de klasse is gemaakt, wordt een methode aan die instantie gekoppeld. In tegenstelling tot een functie die buiten een klasse wordt gedeclareerd, kan een methode niet worden gebruikt zonder de instantie waaraan deze is gekoppeld.

Methoden worden gedefinieerd met het trefwoord `function`. Zoals bij elke klasseneigenschap, kunt u een van de attributen van klasseneigenschap toepassen op methoden, zoals private, protected, public, internal, static of een aangepaste naamruimte. U kunt een instructie function gebruiken, zoals:

```
public function sampleFunction():String {}
```

U kunt ook een variabele gebruiken waaraan u een functie-expressie toewijst:

```
public var sampleFunction:Function = function () {}
```

In de meeste gevallen zult u de instructie function gebruiken in plaats van een functie-expressie, dit vanwege de volgende redenen:

- Instructies function zijn beknopter en eenvoudiger te lezen.
- Met instructies function kunt u de trefwoorden `override` en `final` gebruiken.
- Instructies function zorgen voor een sterkere band tussen de id, de naam van de functie en de code in de hoofdtekst van de methode. Omdat de waarde van een variabele met een toewijzingsinstructie kan worden gewijzigd, kan de verbinding tussen een variabele en zijn functie-expressie op elk moment worden verbroken. Hoewel u dit probleem kunt vermijden door de variabele te declareren met `const` in plaats van `var`, wordt deze techniek niet aangeraden, omdat het de code moeilijk te lezen maakt en dit het gebruik van de trefwoorden `override` en `final` verhindert.

Een van de gevallen waarin u een functie-expressie moet gebruiken, is wanneer u een functie aan het prototypeobject wilt koppelen.

Constructormethoden

Constructormethoden, oftewel *constructors*, zijn functies die de naam van de klasse delen waarin ze zijn gedefinieerd. Elke code die u in een constructormethode opneemt, wordt uitgevoerd wanneer een instantie van de klasse wordt gemaakt met het trefwoord `new`. De volgende code definieert bijvoorbeeld een eenvoudige klasse met de naam Example, die een enkele eigenschap met de naam `status` bevat. De oorspronkelijke waarde van de variabele `status` wordt binnen de constructorfunctie ingesteld.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Constructormethoden kunnen alleen public zijn, maar het gebruik van het attribuut `public` is optioneel. U kunt geen van de andere toegangsbeheerspecificaties gebruiken, ook niet `private`, `protected` of `internal` voor een constructor. U kunt ook geen door de gebruiker gedefinieerde naamruimte gebruiken met een constructormethode.

Een constructor kan een expliciete aanroep naar de constructor of naar een van zijn directe superklassen maken met de instructie `super()`. Als de constructor van superklassen niet expliciet wordt aangeroepen, voegt de compiler automatisch een aanroep in voor de eerste instructie in de constructorhoofdtekst. U kunt methoden van de superklasse ook aanroepen met het voorvoegsel `super` als een verwijzing naar de superklasse. Als u besluit om zowel `super()` als `super` in dezelfde constructorhoofdtekst te gebruiken, moet u zorgen dat u eerst `super()` aanroept. Anders gedraagt de verwijzing `super` zich niet als verwacht. De constructor `super()` moet ook worden aangeroepen voor een instructie `throw` of `return`.

Het volgende voorbeeld demonstreert wat er gebeurt als u de verwijzing `super` probeert te gebruiken voordat u de constructor `super()` aanroept. Een nieuwe klasse, ExampleEx, breidt de klasse Example uit. De constructor ExampleEx probeert toegang te krijgen tot de variabele status die in zijn superklasse is gedefinieerd, maar doet dit voordat deze `super()` aanroept. De instructie `trace()` binnen de constructor ExampleEx produceert de waarde `null` omdat de variabele `status` niet beschikbaar is totdat de constructor `super()` wordt uitgevoerd.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Hoewel u de instructie `return` binnen een constructor kunt gebruiken, is het niet toegestaan een waarde te retourneren. Met andere woorden, de instructie `return` moet geen gekoppelde expressies of waarden hebben. Constructormethoden kunnen dan ook geen waarden retourneren, wat inhoudt dat er geen retourneringstype mag worden opgegeven.

Als u geen constructormethode in uw klasse definieert, maakt de compiler automatisch een lege constructor. Als uw klasse een andere klasse uitbreidt, voegt de compiler een aanroep `super()` toe aan de constructor die door het programma wordt gemaakt.

Methoden van het type static

Statische methoden, oftewel *klassenmethoden*, zijn methoden die met het trefwoord `static` zijn gedeclareerd. Statische methoden, die aan een klasse zijn gekoppeld in plaats van aan de instantie van een klasse, zijn nuttig voor het inkapselen van functies die effect hebben op iets anders dan de status van een individuele instantie. Omdat statische methoden als een geheel aan een klasse zijn gekoppeld, hebt u alleen via een klasse toegang tot statische methoden en niet via de instantie van een klasse.

Statische methoden zijn nuttig voor het inkapselen van functies die niet beperkt zijn tot het beïnvloeden van de status van klasseninstanties. Met andere woorden, een methode zou statisch moeten zijn als deze functies biedt die niet direct effect hebben op de waarde van een klasseninstantie. De klasse Date bevat bijvoorbeeld een statische methode met de naam `parse()`, die een tekenreeks gebruikt en het in een getal omzet. De methode is statisch, omdat deze geen effect heeft op een individuele instantie van de klasse. In plaats hiervan gebruikt de methode `parse()` een tekenreeks die een datumwaarde vertegenwoordigt, parseert deze tekenreeks en retourneert een getal met een indeling die compatibel is met de interne representatie van een object Date. Deze methode is geen instantiemethode, omdat het geen nut heeft om de methode op een instantie van de klasse Date toe te passen.

Vergelijk de statische methode `parse()` met een van de instantiemethoden van de klasse Date, zoals `getMonth()`. De methode `getMonth()` is een instantiemethode, omdat deze direct met de waarde van een instantie werkt door een bepaald component, de maand, van een instantie Date op te halen.

Omdat statische methoden niet gekoppeld zijn aan individuele instanties, kunt u de trefwoorden `this` en `super` niet binnen de hoofdtekst van een statische methode gebruiken. Zowel de verwijzing `this` als `super` zijn alleen nuttig binnen de context van een instantiemethode.

In tegenstelling tot sommige op klassen gebaseerde programmeertalen worden statische methoden in ActionScript 3.0 niet overgeërfd.

Instantiemethoden

Instantiemethoden zijn methoden die zijn gedeclareerd zonder het trefwoord `static`. Instantiemethoden, die aan instanties van een klasse zijn gekoppeld in plaats van aan een klasse zelf, zijn nuttig voor het implementeren van functionaliteit die individuele instanties van een klasse beïnvloedt. De klasse Array bevat bijvoorbeeld een instantiemethode met de naam `sort()`, die rechtstreeks met instanties Array werkt.

In de hoofdtekst van een instantiemethode zijn zowel statische als instantievariabelen bereikbaar. Dit houdt in dat met een eenvoudige id naar variabelen kan worden verwezen die in dezelfde klasse zijn gedefinieerd. De volgende klasse, CustomArray, breidt bijvoorbeeld de klasse Array uit. De klasse CustomArray definieert een statische variabele met de naam `arrayCountTotal` om het totale aantal klasseninstanties bij te houden, een instantievariabele met de naam `arrayNumber` om de volgorde bij te houden waarin de instanties zijn gemaakt en een instantiemethode met de naam `getPosition()` die de waarden van deze variabelen retourneert.

```
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);
    }
}
```

Hoewel code buiten de klasse naar de statische variabele `arrayCountTotal` moet via het klassenobject toegang krijgen met gebruik van `CustomArray.arrayCountTotal`, kan code in de hoofdtekst van de methode `getPosition()` direct naar de statische variabele `arrayCountTotal` verwijzen. Dit geldt ook voor statische variabelen in superklassen. Statische eigenschappen worden in ActionScript 3.0 niet overgeërfd, maar statische eigenschappen in superklassen liggen in het bereik. De klasse Array heeft bijvoorbeeld enkele statische variabelen, waarvan één een constante is met de naam `DESCENDING`. Code in een subklasse Array kan toegang hebben tot de statische constante `DESCENDING` met een eenvoudige id:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

De waarde van de verwijzing `this` in de hoofdtekst van een instantiemethode is een verwijzing naar de instantie waaraan de methode is gekoppeld. De volgende code demonstreert dat de verwijzing `this` naar de instantie verwijst die de methode bevat:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Overerving van instantiemethoden kan worden beheerd met de trefwoorden `override` en `final`. U kunt het attribuut `override` gebruiken om een overgeërfde methode opnieuw te definiëren, en het attribuut `final` gebruiken om te voorkomen dat subklassen een methode overschrijven.

Accessormethoden get en set

Met de accessorfuncties get en set, ook bekend als *getters* en *setters*, kunt u voldoen aan de programmeerprincipes voor het verbergen van informatie en inkapseling en beschikt u over een eenvoudig te gebruiken programmeerinterface voor de klassen die u maakt. Met de functies get en set kunt u klasseneigenschappen als private instellen voor de klasse, maar kunnen gebruikers van de klasse deze eigenschappen wel benaderen alsof zij een klassenvariabele benaderen in plaats van een klassenmethode aan te roepen.

Het voordeel van deze aanpak is dat u hiermee het gebruik van traditionele accessorfuncties met onpraktische namen kunt vermijden, zoals `getPropertyName()` en `setPropertyName()`. Een ander voordeel van getters en setters is dat u het gebruik kunt vermijden van twee publieksgerichte functies voor elke eigenschap waarmee u lees- en schrijftoegang toestaat.

De volgende voorbeeldklasse, GetSet genaamd, bevat accessorfuncties get en set met de naam `publicAccess()` die toegang bieden tot een variabele van het type private met de naam `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Als u probeert direct toegang te krijgen tot de eigenschap `privateProperty`, treedt een fout op; dit wordt aan de hand van het volgende voorbeeld getoond:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

In plaats hiervan maakt de gebruiker van de klasse GetSet gebruik van iets dat vergelijkbaar is met de eigenschap met de naam `publicAccess`, maar dat eigenlijk een paar accessorfuncties get en set is die werken met de eigenschap van het type private met de naam `privateProperty`. Het volgende voorbeeld instantieert de klasse GetSet en stelt vervolgens de waarde van `privateProperty` in met de accessor van het type public met de naam `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Functies getter en setter maken het ook mogelijk eigenschappen te overschrijven die zijn overgeërfd van een superklasse; dit is niet mogelijk wanneer u standaardklassenlidvariabelen gebruikt. Klassenlidvariabelen die zijn gedeclareerd met het trefwoord `var`, kunnen in een subklasse niet worden overschreven. Eigenschappen die zijn gemaakt met functies getter en setter, hebben deze beperking echter niet. U kunt het attribuut `override` gebruiken voor functies getter en setter die van een superklasse zijn overgeërfd.

Gebonden methoden

Een gebonden methode, ook bekend als een *methode closure*, is een methode die is opgehaald uit zijn instantie. Voorbeelden van gebonden methoden zijn methoden die als argumenten aan een functie zijn doorgegeven of zijn geretourneerd als waarden van een functie. Gebonden methoden zijn nieuw in ActionScript 3.0 en zijn vergelijkbaar met een functie closure door het feit dat deze zelfs na het ophalen uit zijn instantie zijn lexicale omgeving behoudt. Het belangrijkste verschil tussen een gebonden methode en een functie closure is dat de verwijzing `this` van een gebonden methode gekoppeld (gebonden) blijft aan de instantie die de methode implementeert. Met andere woorden, de verwijzing `this` verwijst in een gebonden methode altijd naar het oorspronkelijke object dat de methode heeft geïmplementeerd. Voor functies closure is de verwijzing `this` algemeen; dit houdt in dat de functie verwijst naar een object waaraan de functie is gekoppeld op het moment dat deze wordt aangeroepen.

Het is belangrijk dat u de werking van gebonden methoden begrijpt als u het trefwoord `this` gebruikt. Het trefwoord `this` biedt een verwijzing naar het bovenliggende object van een methode. De meeste ActionScript-programmeurs verwachten dat het trefwoord `this` altijd het object of de klasse vertegenwoordigt die de definitie van een methode bevat. Zonder het binden van methoden is dit echter niet altijd waar. In vorige versies van ActionScript verwees het gereserveerde woord `this` bijvoorbeeld niet altijd naar de instantie die de methode heeft geïmplementeerd. Wanneer methoden uit een instantie in ActionScript 2.0 worden opgehaald, is de verwijzing `this` niet alleen aan de oorspronkelijk instantie gebonden, maar zijn ook de lidvariabelen en methoden van de klasse van de instantie niet beschikbaar. Dit vormt geen probleem in ActionScript 3.0, omdat gebonden methoden automatisch worden gemaakt wanneer u een methode als een parameter doorgeeft. Gebonden methoden zorgen ervoor dat het trefwoord `this` altijd naar het object of de klasse verwijst waarin een methode is gedefinieerd.

De volgende code definieert een klasse met de naam ThisTest; de klasse bevat een methode met de naam `foo()` die de gebonden methode definieert en een methode met de naam `bar()` die de gebonden methode retourneert. Externe klassencode maakt een instantie van de klasse ThisTest, roept de methode `bar()` aan en slaat de geretourneerde waarde in een variabele op met de naam `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 */
```

De laatste twee coderegels laten zien dat de verwijzing `this` in de gebonden methode `foo()` nog steeds naar een instantie van de klasse ThisTest verwijst, zelfs nadat de verwijzing `this` in de regel hiervoor naar het algemene object verwijst. Bovendien heeft de gebonden methode die is opgeslagen in de variabele `myFunc`, nog steeds toegang tot de lidvariabelen van de klasse ThisTest. Als dezelfde code in ActionScript 2.0 wordt uitgevoerd, zouden de verwijzingen `this` overeenkomen en zou de variabele `numundefined` zijn.

De toegevoegde waarde van gebonden methoden is vooral merkbaar bij het gebruik van gebeurtenishandlers, omdat voor de methode `addEventListener()` is vereist dat u een functie of methode als argument doorgeeft.

Opsommingen met klassen

Opsommingen zijn aangepaste gegevenstypen die u maakt om een kleine set waarden in te kapselen. ActionScript 3.0 ondersteunt geen specifieke opsomfaciliteit, dit in tegenstelling met C++ die het trefwoord `enum` kent of Java met een opsommingsinterface. U kunt opsommingen echter maken met klassen en statische constanten. De klasse PrintJob in ActionScript 3.0 gebruikt bijvoorbeeld een opsomming met de naam PrintJobOrientation om de waarden `"landscape"` (liggend) en `"portrait"` (staand) op te slaan, zoals geïllustreerd in de volgende code:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Normaal gesproken wordt een opsommingsklasse gedeclareerd met het attribuut `final`, omdat het niet nodig is de klasse uit te breiden. De klasse bestaat alleen uit leden van het type static; dit houdt in dat u geen instanties van de klasse hoeft te maken. In plaats hiervan hebt u via het klassenobject direct toegang tot de waarde van de opsomming, zoals getoond in het volgende codefragment:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Elk van de opsommingsklassen in ActionScript 3.0 bevat alleen variabelen van het type String, int of uint. Het voordeel van het gebruik van opsommingen in plaats van een letterlijke tekenreeks of getalwaarden is dat typografische fouten eenvoudiger op te sporen zijn. Als u de naam van een opsomming fout spelt, genereert de ActionScript-compiler een fout. Als u letterlijke waarden gebruikt, genereert de compiler geen fout als u een woord onjuist spelt of een fout getal gebruikt. In het vorige voorbeeld genereert de compiler een fout als de naam van de constante in de opsomming onjuist is, zoals getoond door het volgende fragment:

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

De compiler genereert echter geen fout als u een letterlijke tekenreekswaarde fout spelt:

```
    if (pj.orientation == "portrai") // no compiler error
```

Een tweede manier waarop u opsommingen kunt maken, is met een afzonderlijke klasse met statische eigenschappen voor de opsomming. Deze techniek verschilt echter in het feit dat elk van de statische eigenschappen een instantie van de klasse bevat in plaats van een tekenreeks of een geheel getal. De volgende code maakt bijvoorbeeld een opsommingsklasse voor de dagen van de week:

```
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();
}
```

Deze techniek wordt niet door ActionScript 3.0 gebruikt, maar door veel ontwikkelaars die de verbeterde typecontrole die de techniek biedt willen benutten. Een methode die bijvoorbeeld een opsommingswaarde retourneert, kan de geretourneerde waarde beperken tot het gegevenstype van de opsomming. De volgende code toont niet alleen een functie die een weekdag retourneert, maar ook een functieaanroep die het opsommingstype gebruikt als een typeannotatie:

```
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();
```

U kunt de klasse Day ook op een dergelijke manier verbeteren dat deze een geheel getal aan elke dag van de week koppelt en een methode `toString()` biedt die een tekenreeksrepresentatie van de dag retourneert.

Ingesloten klassenelementen

ActionScript 3.0 gebruikt special klassen, *ingesloten klassenelementen* genoemd, om ingesloten elementen te vertegenwoordigen. Een *ingesloten element* is een element, zoals een geluid, afbeelding of lettertype dat bij uitvoering in een SWF-bestand is opgenomen. Als u een element insluit in plaats van het dynamisch te laden, zorgt u ervoor dat het bij uitvoering beschikbaar is; dit gaat echter ten koste van een verhoogde SWF-bestandsgrootte.

Ingesloten klassenelementen in Flash Professional gebruiken

Als u een element wilt insluiten, plaatst u het element eerst in de bibliotheek van een FLA-bestand. Gebruik vervolgens de koppelingseigenschap van het element om een naam op te geven voor het ingesloten klassenelement van het element. Als er geen klasse met die naam in het klassenpad kan worden gevonden, wordt er automatisch een klasse voor u gegenereerd. U kunt vervolgens een instantie van het ingesloten klassenelement maken en eigenschappen en methoden gebruiken die door die klasse zijn gedefinieerd of overgeërfd. De volgende code kan bijvoorbeeld worden gebruikt om een ingesloten geluid af te spelen dat is gekoppeld aan een ingesloten klassenelement met de naam PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

U kunt ook de metagegevenstag `[Embed]` gebruiken om middelen in een Flash Professional-project in te sluiten, zoals beschreven in de volgende sectie. Als u de metagegevenstag `[Embed]` in uw code gebruikt, gebruikt Flash Professional in plaats van de Flash Professional-compiler de Flex-compiler om uw project te compileren.

Met de ingesloten klassen van middelen met de Flex-compiler.

Om een middel in een ActionScript-code in te sluiten, als u uw code compileert met de Flex-compiler, gebruikt u de metagegevenstag `[Embed]`. Plaats het element in de hoofdbronmap of in een andere map die zich in het bouwpad van uw project bevindt. Wanneer de compiler van Flex een metagegevenstag Embed aantreft, wordt het ingesloten klassenelement automatisch gemaakt. U kunt toegang krijgen tot de klasse met behulp van een variabele van het gegevenstype Class dat u direct na de metagegevenstag `[Embed]` declareert. De volgende code sluit bijvoorbeeld een geluid met de naam geluid1.mp3 in en gebruikt een variabele met de naam `soundCls` om een verwijzing naar het ingesloten klassenelement op te slaan dat aan dat geluid is gekoppeld. In het voorbeeld wordt vervolgens een instantie van het ingesloten klassenelement gemaakt en de methode `play()` voor die instantie aangeroepen:

```
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

Als u de metagegevenstag `[Embed]` wilt gebruiken in een Flash Builder ActionScript-project, moet u eventueel noodzakelijke klassen uit het Flex-framework importeren. Als u bijvoorbeeld geluiden wilt insluiten, moet u de klasse mx.core.SoundAsset importeren. Als u het Flex-framework wilt gebruiken, neemt u het bestand framework.swc op in uw ActionScript-bouwpad. Daarmee wordt het SWF-bestand vergroot.

Adobe Flex

U kunt in Flex ook een element insluiten met de aanwijzing `@Embed()` in een MXML-tagdefinitie.

Interfaces

Een interface is een verzameling methodedeclaraties waarmee ongerelateerde objecten met elkaar kunnen communiceren. ActionScript 3.0 definieert bijvoorbeeld de interface IEventDispatcher; deze bevat methodedeclaraties waarvan een klasse gebruik kan maken om gebeurtenisobjecten af te handelen. De interface IEventDispatcher zorgt voor een standaardmanier voor objecten om gebeurtenisobjecten aan elkaar door te geven. De volgende code toont de definitie van de interface IEventDispatcher:

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Interfaces zijn gebaseerd op het verschil tussen de interface van een methode en zijn implementatie. De interface van een methode bevat alle informatie die nodig is om die methode aan te roepen, inclusief de naam van de methode, alle parameters en het retourneringstype. De implementatie van een methode bevat niet alleen de interface-informatie, maar ook de uitvoerbare instructies die het gedrag van de methode uitvoeren. Een interfacedefinitie bevat alleen methode-interfaces en elke klasse die de interface implementeert, is verantwoordelijk voor het definiëren van de implementaties van methoden.

In ActionScript 3.0 implementeert de klasse EventDispatcher de interface IEventDispatcher door alle methoden van de interface IEventDispatcher te definiëren en door hoofdteksten van methoden aan elke methode toe te voegen. De volgende code is een fragment uit de klassendefinitie van EventDispatcher:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

De interface IEventDispatcher dient als een protocol dat de instanties EventDispatcher gebruiken om gebeurtenisobjecten te verwerken en ze aan andere objecten door te geven die de interface IEventDispatcher ook hebben geïmplementeerd.

U kunt een interface ook beschrijven door te zeggen dat een interface een gegevenstype op dezelfde manier als een klasse definieert. Een interface kan dan ook net als een klasse als een typeannotatie worden gebruikt. Net als een gegevenstype kan een interface ook met operatoren worden gebruikt, zoals de operatoren `is` en `as`, die een gegevenstype vereisen. In tegenstelling tot een klasse kan een interface echter niet worden geïnstantieerd. Door dit verschil denken veel programmeurs dat interfaces abstracte gegevenstypen zijn en klassen concrete gegevenstypen.

Interface definiëren

De structuur van een interfacedefinitie is vergelijkbaar met die van een klassendefinitie, behalve dat een interface alleen methoden zonder hoofdteksten kan bevatten. Interfaces kunnen geen variabelen of constanten bevatten, maar wel getters en setters. Als u een interface wilt definiëren, gebruikt u het trefwoord `interface`. De volgende interface, IExternalizable, maakt bijvoorbeeld deel uit van het pakket flash.utils in ActionScript 3.0. De interface IExternalizable definieert een protocol voor het serialiseren van een object; dat betekent het converteren van een object naar een indeling die geschikt is voor opslag op een apparaat of voor transport over een netwerk.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

De interface IExternalizable wordt gedeclareerd met de toegangsbeheerspecificatie `public`. Interfacedefinities kunnen alleen met de toegangsbeheerspecificaties `public` en `internal` worden gewijzigd. De methodedeclaraties binnen een interfacedefinitie kunnen geen toegangsbeheerspecificaties bevatten.

ActionScript 3.0 volgt een conventie waarin interfacenamen beginnen met een hoofdletter `I`, maar u kunt elke legale id als een interfacenaam gebruiken. Interfacedefinities worden vaak op het hoofdniveau van een pakket geplaatst. Interfacedefinities kunnen niet binnen een klassendefinitie of binnen een andere interfacedefinitie worden geplaatst.

Interfaces kunnen een of meer andere interfaces uitbreiden. De volgende interface, IExample, breidt bijvoorbeeld de interface IExternalizable uit:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Elke klasse die de interface IExample uitbreidt, moet niet alleen implementaties voor de methode `extra()` opnemen, maar ook voor de methoden `writeExternal()` en `readExternal()` die zijn overgeërfd van de interface IExternalizable.

Interface in een klasse implementeren

Een klasse is het enige taalelement in ActionScript 3.0 dat een interface kan implementeren. Gebruik het trefwoord `implements` in een klassendeclaratie om een of meer interfaces te implementeren. Het volgende voorbeeld definieert twee interfaces, IAlpha en IBeta , en een klasse, Alpha, die beide implementeert:

```
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 een klasse die een interface implementeert, moeten de geïmplementeerde methoden het volgende doen:

- De toegangsbeheer-id `public` gebruiken.
- Dezelfde naam gebruiken als de interfacemethode.
- Hetzelfde aantal parameters hebben, elk met gegevenstypen die overeenkomen met de gegevenstypen van de interfacemethodeparameter.
- Hetzelfde retourneringstype gebruiken.

  ```
  public function foo(param:String):String {}
  ```

U hebt niettemin enige flexibiliteit bij het benoemen van de parameters van de methoden die u implementeert. Hoewel het aantal parameters en het gegevenstype van elke parameter in de geïmplementeerde methode overeen moet komen met die van de interfacemethode, hoeven de namen van de parameters niet overeen te komen. In het vorige voorbeeld heeft de parameter van de methode `Alpha.foo()` de naam `param`:

Maar de parameter in de interfacemethode `IAlpha.foo()` heeft de naam `str`:

```
function foo(str:String):String;
```

U hebt tevens enige flexibiliteit met standaardparameterwaarden. Een interfacedefinitie kan functiedeclaraties met standaardparameterwaarden bevatten. Een methode die een dergelijke functiedeclaratie implementeert, moet een standaardparameterwaarde hebben die lid is van hetzelfde gegevenstype als de waarde die is opgegeven in de interfacedefinitie, maar de daadwerkelijke waarde hoeft niet overeen te komen. De volgende code definieert bijvoorbeeld een interface die code bevat met een standaardparameterwaarde 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

De volgende klassendefinitie implementeert de interface Igamma, maar gebruikt een andere standaardparameterwaarde:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

De reden voor deze flexibiliteit ligt in het feit dat de regels voor het implementeren van een interface specifiek ontworpen zijn voor het verzekeren van de compatibiliteit tussen gegevenstypen en identieke parameternamen; standaardparameterwaarden zijn niet nodig om dit te realiseren.

Overerving

Overerving is een soort hergebruik van code waardoor programmeurs in staat zijn nieuwe klassen te ontwikkelen die op bestaande klassen zijn gebaseerd. De bestaande klassen worden vaak *basisklassen* of *superklassen* genoemd, terwijl de nieuwe klassen doorgaans *subklassen* worden genoemd. Het grootste voordeel van overerving is dat u de code van basisklassen opnieuw kunt gebruiken en de bestaande code ongewijzigd laat. Bovendien hoeft de manier waarop andere klassen met de basisklasse communiceren niet te worden gewijzigd. In plaats van een bestaande klasse te wijzigen die waarschijnlijk al grondig is getest of al in gebruik is, kunt u met overerving een klasse als een geïntegreerde module behandelen die u met aanvullende eigenschappen of methoden kunt uitbreiden. U kunt het trefwoord `extends` dus gebruiken om aan te geven dat een klasse overerft van een andere klasse.

Dankzij overerving kunt u gebruikmaken van *polymorfisme* in de code. Met polymorfisme kunt u een enkele methodenaam gebruiken voor een methode die zich anders gedraagt, afhankelijk van het gegevenstype waarop deze wordt toegepast. Een eenvoudig voorbeeld is een basisklasse met de naam Shape, die twee subklassen met de naam Circle en Square bevat. De klasse Shape definieert een methode met de naam `area()`, die het gebied van de vorm retourneert. Als polymorfisme wordt geïmplementeerd, kunt u de methode `area()` aanroepen voor objecten van het type Circle en Square en deze de correcte berekeningen voor u laten uitvoeren. Overerving staat polymorfisme toe door subklassen toe te staan methoden van de basisklasse te overerven, opnieuw te definiëren of te *overschrijven*. In het volgende voorbeeld wordt de methode `area()` opnieuw gedefinieerd door de klassen Circle en Square:

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Aangezien elke klasse een gegevenstype definieert, brengt het gebruik van overerving een speciale relatie tot stand tussen een basisklasse en de klasse die deze uitbreidt. Een subklasse heeft altijd beschikking tot de eigenschappen van de basisklasse; dit houdt in dat een instantie van een subklasse altijd kan worden vervangen door een instantie van een basisklasse. Als een methode bijvoorbeeld een parameter van het type Shape definieert, is het mogelijk een argument van het type Circle door te geven, omdat Circle de Shape uitbreidt. Dit wordt getoond met de volgende code:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Instantie-eigenschappen en overerving

Een instantie-eigenschap, of die nu met het trefwoord `function`, `var` of `const` is gedefinieerd, wordt overgeërfd door alle subklassen, op voorwaarde dat de eigenschap in de basisklasse niet met het attribuut `private` is gedefinieerd. De klasse Event in ActionScript 3.0 bevat bijvoorbeeld een aantal subklassen die eigenschappen overerven die van toepassing zijn op alle gebeurtenisobjecten.

Voor sommige gebeurtenissen bevat de klasse Event alle benodigde eigenschappen om de gebeurtenis te definiëren. Deze gebeurtenissen vereisen naast de gedefinieerde instantie-eigenschappen in de klasse Event geen aanvullende instantie-eigenschappen. Voorbeelden van dergelijke gebeurtenissen zijn de gebeurtenis `complete`, die optreedt wanneer gegevens zijn geladen, en de gebeurtenis `connect`, die optreedt wanneer een netwerkverbinding tot stand is gebracht.

Het volgende voorbeeld is een fragment uit de klasse Event die enkele van de eigenschappen en methoden toont die door subklassen worden overgeërfd. Omdat de eigenschappen worden overgeërfd, heeft een instantie van een subklasse toegang tot deze eigenschappen.

```
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 {}
    ...
}
```

Overige typen gebeurtenissen vereisen unieke eigenschappen die niet beschikbaar zijn in klasse Event. Deze gebeurtenissen worden gedefinieerd met gebruik van de subklassen van de klasse Event zodat nieuwe eigenschappen aan de eigenschappen kunnen worden toegevoegd die in de klasse Event zijn gedefinieerd. Een voorbeeld van een dergelijke subklasse is de klasse MouseEvent; deze klasse voegt eigenschappen toe die uniek zijn voor gebeurtenissen die zijn gekoppeld aan muisbewegingen of muisklikken, zoals de gebeurtenissen `mouseMove` en `click`. Het volgende voorbeeld is een fragment van de klasse MouseEvent die de definitie van eigenschappen toont die bestaat in de subklasse, maar niet in de basisklasse:

```
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 {}
    ...
}
```

Toegangsbeheerspecificaties en overerving

Als een eigenschap is gedeclareerd met het trefwoord `public`, is de eigenschap zichtbaar voor alle code. Dit houdt in dat het trefwoord `public`, in tegenstelling tot de trefwoorden `private`, `protected` en `internal`, geen beperkingen oplevert voor het overerven van eigenschappen.

Als een eigenschap is gedeclareerd met het trefwoord `private`, is deze alleen zichtbaar in de klasse die de eigenschap definieert; de eigenschap kan dus niet door subklassen worden overgeërfd. Dit gedrag verschilt met vorige versies van ActionScript, waar het trefwoord `private` zich meer als het trefwoord `protected` van ActionScript 3.0 gedroeg.

Het trefwoord `protected` geeft aan dat een eigenschap niet alleen zichtbaar is binnen de klasse die de eigenschap heeft gedefinieerd, maar ook voor alle subklassen. In tegenstelling tot het trefwoord `protected` in de Java-programmeertaal, maakt het trefwoord `protected` in ActionScript 3.0 een eigenschap niet zichtbaar voor alle andere klassen in hetzelfde pakket. In ActionScript 3.0 hebben alleen subklassen toegang tot een eigenschap die is gedeclareerd met het trefwoord `protected`. Bovendien is een beschermde eigenschap zichtbaar voor een subklasse, ongeacht of de subklasse zich in hetzelfde pakket als de basisklasse bevindt of in een ander pakket.

Als u de zichtbaarheid van een eigenschap wilt beperken voor het pakket waarin deze is gedefinieerd, gebruikt u het trefwoord `internal` of maakt u geen gebruik van een toegangsbeheerspecificatie. De toegangsbeheerspecificatie `internal` is de toegangsbeheerspecificatie die standaard wordt gebruikt wanneer er geen is opgegeven. Een eigenschap die is gemarkeerd als `internal`, wordt alleen door een subklasse overgeërfd die zich in hetzelfde pakket bevindt.

U kunt aan de hand van het volgende voorbeeld zien hoe elk van de toegangsbeheerspecificaties pakketgrensoverschrijdende overerving beïnvloedt. De volgende code definieert een hoofdtoepassingsklasse met de naam AccessControl en twee andere klassen met de naam Base en Extender. De klasse Base bevindt zich in een pakket met de naam foo en de klasse Extender, een subklasse van de klasse Base, bevindt zich in een pakket met de naam bar. De klasse AccessControl importeert alleen de klasse Extender en maakt een instantie van Extender die toegang probeert te krijgen tot een variabele met de naam `str`, die in de klasse Base is gedefinieerd. De variabele `str` is gedeclareerd als `public`, zodat de code wordt gecompileerd en uitgevoerd zoals getoond in het volgende fragment:

```
// 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
        }
    }
}
```

Als u het effect van de overige toegangsbeheerspecificaties wilt zien op de compilatie en uitvoering van het vorige voorbeeld, wijzigt u de toegangsbeheerspecificatie van de variabele `str` in `private`, `protected` of `internal` nadat u de volgende regel uit de klasse `AccessControl` hebt verwijderd of genegeerd:

```
trace(myExt.str);// error if str is not public
```

Variabelen overschrijven niet toegestaan

Eigenschappen die zijn gedeclareerd met de trefwoorden `var` of `const`, worden overgeërfd, maar kunnen niet worden overschreven. Als u een eigenschap wilt overschrijven, moet de eigenschap in een subklasse opnieuw worden gedefinieerd. De enige type eigenschap die u kunt overschrijven, zijn get/set-methoden, oftewel eigenschappen die zijn gedeclareerd met het trefwoord `function)`. Hoewel u een instantievariabele niet kunt overschrijven, kunt u een vergelijkbare functionaliteit verkrijgen door methoden getter en setter te maken voor de instantievariabele en de methoden te overschrijven.

Methoden overschrijven

Als u een methode wilt overschrijven, moet u het gedrag van een overgeërfde methode opnieuw definiëren. Statische methoden worden niet overgeërfd en kunnen niet worden overschreven. Instantiemethoden worden echter door subklassen overgeërfd en kunnen worden overschreven zolang er aan de volgende twee criteria wordt voldaan:

- De instantiemethode wordt niet gedeclareerd met het trefwoord `final` in de basisklasse. Bij gebruik met een instantiemethode geeft het trefwoord `final` de poging van de programmeur aan te verhinderen dat subklassen de methode overschrijven.
- De instantiemethode wordt niet gedeclareerd met de toegangsbeheerspecificatie `private` in de basisklasse. Als een methode in de basisklasse als `private` is gemarkeerd, is het niet nodig het trefwoord `override` te gebruiken wanneer u een methode met een identieke naam in de subklasse definieert, omdat de basisklassenmethode niet zichtbaar is voor de subklasse.

  Als u een instantiemethode wilt overschrijven die aan deze criteria voldoet, moet de methodedefinitie in de subklasse gebruik maken van het trefwoord `override` en moet als volgt overeenkomen met de superklassenversie van de methode:

- De overschrijfmethode moet hetzelfde toegangsbeheerniveau hebben als de basisklassenmethode. Methoden die als internal zijn gemarkeerd, hebben hetzelfde toegangsbeheerniveau als methoden die geen toegangsbeheerspecificatie hebben.
- De overschrijfmethode moet hetzelfde aantal parameters hebben als de basisklassenmethode.
- De overschrijfmethodeparameters moeten dezelfde gegevenstypeannotaties hebben als de parameters in de basisklassenmethode.
- De overschrijfmethode moet hetzelfde retourneringstype hebben als de basisklassenmethode.

De namen van de parameters in de overschrijfmethoden hoeven echter niet overeen te komen met de namen van de parameters in de basisklasse, zolang als het aantal parameters en het gegevenstype van elke parameter overeenkomt.

Instructie super

Wanneer een programmeur een methode overschrijft, wil deze vaak het gedrag van de superklassenmethode uitbreiden in plaats van het gedrag volledig te vervangen. Dit vereist een mechanisme dat een methode in een subklasse toestaat de superklassenversie van zichzelf aan te roepen. De instructie `super` biedt een dergelijk mechanisme, deze bevat namelijk een verwijzing naar de directe superklasse. Het volgende voorbeeld definieert een klasse met de naam Base die een methode bevat met de naam `thanks()` en een subklasse van de klasse Base met de naam Extender, die de methode `thanks()` overschrijft. De methode `Extender.thanks()` gebruikt de instructie `super` om `Base.thanks()` aan te roepen.

```
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";
    }
}
```

Getters en setters overschrijven

Hoewel u geen variabelen kunt overschrijven die in een superklasse zijn gedefinieerd, kunt u getters en setters overschrijven. De volgende code heeft bijvoorbeeld de getter `currentLabel` op die in de klasse MovieClip van ActionScript 3.0 is gedefinieerd:

```
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;
        }
    }
}
```

De uitvoer van de instructie `trace()` in de klassenconstructor OverrideExample is `Override: null`, waaruit blijkt dat in het voorbeeld de overgeërfde eigenschap `currentLabel` kon worden overschreven.

Statische niet-overgeërfde eigenschappen

Statische eigenschappen worden niet door subklassen overgeërfd. Dit houdt in dat statische eigenschappen niet kunnen worden benaderd via de instantie van een subklasse. Een statische eigenschap kan alleen worden benaderd via het klassenobject waarvoor die is gedefinieerd. De volgende code definieert bijvoorbeeld een basisklasse met de naam Base en een subklasse die Base uitbreidt en de naam Extender draagt. Een statische variabele met de naam `test` wordt in de klasse Base gedefinieerd. De code zoals geschreven in het volgende fragment compileert niet in strikte modus en genereert een uitvoerfout in de standaardmodus.

```
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 { }
```

De enige manier waarop de statische variabele `test` kan worden benaderd, is via het klassenobject, zoals getoond in de volgende code:

```
Base.test;
```

Het is echter toegestaan om een instantie-eigenschap te definiëren met dezelfde naam als een statische eigenschap. Een dergelijke instantie-eigenschap kan worden gedefinieerd in dezelfde klasse als de statische eigenschap of in een subklasse. De klasse Base in het vorige voorbeeld kan bijvoorbeeld een instantie-eigenschap met de naam `test` hebben. De volgende code wordt gecompileerd en uitgevoerd, omdat de instantie-eigenschap is overgeërfd door de klasse Extender. De code zou ook worden gecompileerd en uitgevoerd als de definitie van de testinstantievariabele wordt verplaatst, maar niet gekopieerd, naar de klasse Extender.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Statische eigenschappen en de bereikketen

Hoewel statische eigenschappen niet worden overgeërfd, bevinden ze zich in de bereikketen van de klasse die ze definieert en alle subklassen van die klasse. Hierdoor wordt gezegd dat statische eigenschappen *in het bereik* liggen van de klasse waarin ze zijn gedefinieerd en alle subklassen. Dit houdt in dat een statische eigenschap direct toegankelijk is binnen de hoofdtekst van de klasse die de statische eigenschap definieert en alle subklassen van die klasse.

Het volgende voorbeeld wijzigt de klassen die in het vorige voorbeeld zijn gedefinieerd, om aan te tonen dat de statische variabele `test` die is gedefinieerd in de klasse Base, in het bereik ligt van de klasse Extender. Met andere woorden, de klasse Extender kan de statische variabele `test` benaderen zonder de naam van de klasse die `test` definieert als voorvoegsel aan de variabele toe te voegen.

```
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
    }

}
```

Als een instantie-eigenschap is gedefinieerd die dezelfde naam gebruikt als een statische eigenschap in dezelfde klasse of een superklasse, heeft de instantie-eigenschap een hogere prioriteit in de bereikketen. De instantie-eigenschap *schaduwt* de statische eigenschap, wat betekent dat de waarde van de instantie-eigenschap wordt gebruikt in plaats van de waarde van de statische eigenschap. De volgende code toont bijvoorbeeld dat als de klasse Extender een instantievariabele met de naam `test` definieert, de instructie `trace()` gebruikmaakt van de waarde van de instantievariabele in plaats van de waarde van de statische variabele.

```
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
    }

}
```

Geavanceerde onderwerpen

Geschiedenis van OOP-ondersteuning in ActionScript

Omdat ActionScript 3.0 voortborduurt op vorige versies van ActionScript, is het wellicht nuttig te begrijpen hoe het ActionScript-objectmodel is ontwikkeld. ActionScript is begonnen als een eenvoudig mechanisme voor het schrijven van scripts in vorige versies van Flash professional. De programmeurs maakten echter steeds complexere toepassingen met ActionScript. Om dergelijke programmeurs tegemoet te komen, zijn er taalfuncties aan elke nieuwe uitgave toegevoegd die het maken van complexe toepassingen vereenvoudigen.

ActionScript 1.0

ActionScript 1.0 verwijst naar de taalversie die in Flash Player 6 en lager is gebruikt. Zelfs in deze vroege ontwikkelingsfase was het ActionScript-objectmodel al gebaseerd op het concept van het object als een fundamenteel gegevenstype. Een ActionScript-object is een samengesteld gegevenstype met een groep *eigenschappen*. In de beschrijving van het objectmodel staat de term *eigenschappen* voor alles dat aan een object is gekoppeld, zoals variabelen, functies of methoden.

Hoewel de eerste generatie van ActionScript de definitie van klassen met het trefwoord `class` niet ondersteunt, kunt u een klasse definiëren met het gebruik van een speciaal object dat een prototypeobject heet. In plaats van dat u het trefwoord `class` gebruikt om een abstracte klassendefinitie te maken die u instantieert in concrete objecten, zoals in talen die op klassen zijn gebaseerd zoals Java en C++, gebruiken op prototypen gebaseerde talen zoals ActionScript 1.0 een bestaand object als model (of prototype) voor andere objecten. Objecten in een taal die op klassen is gebaseerd, wijzen wellicht naar een klasse die als sjabloon dient, maar objecten in een taal die op prototypen is gebaseerd wijzen naar een ander object, het prototype, dat als sjabloon dient.

Als u een klasse in ActionScript 1.0 wilt maken, definieert u een constructorfunctie voor die klasse. In ActionScript zijn functies eigenlijk objecten, niet alleen abstracte definities. De constructorfunctie die u maakt, dient als het prototypeobject voor instanties van die klasse. De volgende code maakt een klasse met de naam Shape en definieert een eigenschap met de naam `visible` die standaard is ingesteld op `true`:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Deze constructorfunctie definieert een klasse Shape die u als volgt kunt instantiëren met de operator `new`:

```
myShape = new Shape();
```

Het constructorfunctieobject `Shape()` dient als het prototype voor instanties van de klasse Shape, maar kan ook als het prototype dienen voor subklassen van Shape, oftewel voor andere klassen die de klasse Shape uitbreiden.

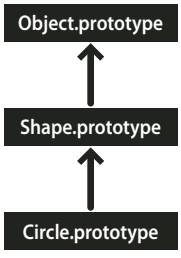
Het maken van een klasse die een subklasse van de klasse Shape is, gebeurt in twee stappen. Maak eerst de klasse door als volgt een constructorfunctie voor de klasse te definiëren:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Gebruik vervolgens de operator `new` om te declareren dat de klasse Shape het prototype voor de klasse Circle is. Elke klasse die u maakt, gebruikt standaard de klasse Object als het prototype; dit houdt in dat `Circle.prototype` momenteel een algemeen object bevat (een instantie van de klasse Object). Als u wilt opgeven dat de prototype van Circle Shape is in plaats van Object, gebruikt u de volgende code om de waarde van `Circle.prototype` te wijzigen zodat deze een object Shape bevat in plaats van een algemeen object:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

De klasse Shape en de klasse Circle zijn nu aan elkaar gekoppeld in een overervingsrelatie die bekend staat als de *prototypeketen*. Het diagram illustreert de relaties in een prototypeketen:



De basisklasse aan het eind van elke prototypeketen is de klasse Object. De klasse Object bevat een eigenschap van het type static met de naam `Object.prototype` die wijst naar het basisprototypeobject voor alle objecten die zijn gemaakt in ActionScript 1.0. Het volgende object in de prototypeketen in het voorbeeld is het object Shape. Dit wegens het feit dat de eigenschap `Shape.prototype` nooit expliciet is ingesteld, zodat het nog steeds een algemeen object bevat (een instantie van de klasse Object). De laatste koppeling in deze keten is de klasse Circle, die is gekoppeld aan zijn prototype, de klasse Shape (de eigenschap `Circle.prototype` bevat een object Shape).

Als we een instantie van de klasse Circle maken, zoals in het volgende voorbeeld, overerft de instantie de prototypeketen van de klasse Circle:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Er is reeds een eigenschap met de naam `visible` gemaakt als een lid van de klasse Shape. In ons voorbeeld bestaat de eigenschap `visible` niet als een deel van het object `myCircle`, alleen als een lid van het object Shape, maar de volgende coderegel geeft toch `true` weer:

```
trace(myCircle.visible); // output: true
```

De uitvoering kan controleren of het object `myCircle` de eigenschap `visible` overerft door de prototypeketen naar boven te doorlopen. Wanneer deze code wordt uitgevoerd, zoekt de uitvoering eerst in de eigenschappen van het object `myCircle` naar een eigenschap met de naam `visible`, maar vindt een dergelijke eigenschap niet. Deze zoekt vervolgens in het object `Circle.prototype`, maar vindt nog steeds geen eigenschap met de naam `visible`. Naarmate deze de prototypeketen naar boven doorloopt, wordt uiteindelijk de eigenschap `visible` gevonden die is gedefinieerd voor het object `Shape.prototype` en geeft hij de waarde van die eigenschap weer.

Om het eenvoudig te houden, zijn veel details van de prototypeketen weggelaten. Het is in plaats daarvan de bedoeling om u voldoende informatie te geven, zodat u het objectmodel van ActionScript 3.0 beter begrijpt.

ActionScript 2.0

ActionScript 2.0 introduceerde nieuwe trefwoorden zoals `class`, `extends`, `public` en `private`, waarmee u klassen op een herkenbare manier kon definiëren voor iedereen die met talen werkt die op klassen zijn gebaseerd, zoals Java en C++. Het is belangrijk om te weten dat het onderliggende overervingsmechanisme niet veranderde tussen ActionScript 1.0 en ActionScript 2.0. In ActionScript 2.0 werd alleen een nieuwe syntaxis toegevoegd voor het definiëren van klassen. De prototypeketen werkt in beide taalversies op dezelfde manier.

Met de nieuwe syntaxis die in ActionScript 2.0 werd geïntroduceerd, zoals getoond in het volgende fragment, kunt u klassen op een manier definiëren die voor vele programmeurs intuïtiever was.

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

ActionScript 2.0 heeft ook typeannotaties geïntroduceerd die werden gebruikt wanneer bij compilatie de typen werden gecontroleerd. Hierdoor kunt u declareren dat de eigenschap `visible` in het vorige voorbeeld slechts een Booleaanse waarde bevat. Het nieuwe trefwoord `extends` vereenvoudigt ook het maken van een subklasse. In het vorige voorbeeld wordt het proces van twee stappen dat nodig is in ActionScript 1.0 met het trefwoord `extends` in slechts één stap gerealiseerd:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

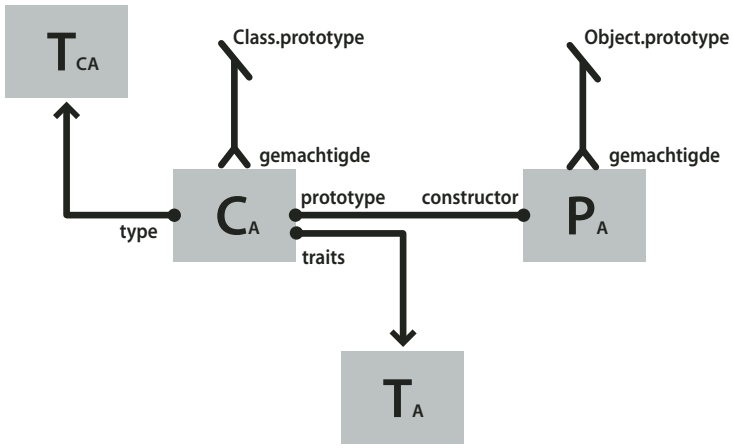
De constructor wordt nu gedeclareerd als deel van de klassendefinitie en de klasseneigenschappen `id` en `radius` moeten ook expliciet worden gedeclareerd.

ActionScript 2.0 heeft ook ondersteuning toegevoegd voor de definitie van interfaces, waarmee u objectgeoriënteerde programma's kunt verfijnen met formeel gedefinieerde protocollen voor interobjectcommunicatie.

ActionScript 3.0-klassenobject

Een algemeen objectgeoriënteerd programmeerparadigma, normaal gesproken geassocieerd met Java en C++, gebruikt klassen om de typen van objecten te definiëren. Programmeertalen die dit paradigma gebruiken, gebruiken doorgaans ook klassen om instanties van het gegevenstype te maken die de klasse definieert. ActionScript gebruikt klassen voor beide doeleinden, maar zijn grondslag als een op prototypen gebaseerde taal voegt een interessant attribuut toe. ActionScript maakt voor elke klassendefinitie een speciaal klassenobject dat het delen van gedrag en status toestaat. Voor veel ActionScript-programmeurs heeft dit verschil echter geen praktische gevolgen voor de code. ActionScript 3.0 is zo ontworpen dat u geavanceerde objectgeoriënteerde ActionScript-toepassingen kunt maken zonder het gebruik van, of enige kennis van, deze speciale klassenobjecten.

Het volgende diagram toont de structuur van een klassenobject die een eenvoudige klasse met de naam A vertegenwoordigt die is gedefinieerd met de instructie `class A {}`:



Elke rechthoek in het diagram vertegenwoordigt een object. Elk object in het diagram heeft een subscript teken A om aan te geven dat het bij klasse A hoort. Het klassenobject (CA) bevat verwijzingen naar een getal of andere belangrijke objecten. Een instantieobject traits (TA) slaat de instantie-eigenschappen op die binnen een klassendefinitie zijn gedefinieerd. Een klassenobject traits (TCA) vertegenwoordigt het interne type van de klasse en slaat de statische eigenschappen op die zijn gedefinieerd door de klasse (het subscript teken C staat voor 'klasse'). Het prototypeobject (PA) verwijst altijd naar het klassenobject waaraan het oorspronkelijk was gekoppeld via de eigenschap `constructor`.

Object traits

Het object traits, nieuw in ActionScript 3.0, is geïmplementeerd met het oog op prestaties. In vorige versies van ActionScript was het opzoeken van namen een tijdrovende zaak, omdat Flash Player de prototypeketen moest doorlopen. In ActionScript 3.0 verloopt het opzoeken van namen veel efficiënter en neemt het aanzienlijk minder tijd in beslag, omdat overgeërfde eigenschappen van superklassen naar het object traits van subklassen worden gekopieerd.

Het object traits is niet direct toegankelijk voor programmeercode, maar de aanwezigheid is merkbaar door prestatieverbetering en geheugenverbruik. Het object traits biedt de AVM2 gedetailleerde informatie over de lay-out en inhoud van een klasse. Met deze informatie is de AVM2 in staat de uitvoertijd aanzienlijk te verminderen, omdat deze vaak directe machine-instructies kan maken om de eigenschappen te benaderen of methoden direct aan te roepen zonder eerst tijd te moeten besteden aan het opzoeken van de naam.

Dankzij het object traits is de geheugenruimte van een object aanzienlijk kleiner dan bij een vergelijkbaar object in vorige versies van ActionScript. Als een klasse bijvoorbeeld is verzegeld (de klasse wordt niet dynamisch gedeclareerd), heeft een instantie van de klasse geen hash-tabel nodig voor dynamisch toegevoegde eigenschappen en kan deze niet meer bevatten dan een verwijzing naar objecten traits en enkele ruimten voor vaste eigenschappen die in de klasse zijn gedefinieerd. Het resultaat is dat een object dat 100 bytes geheugen in ActionScript 2.0 nodig had, nu slechts 20 bytes geheugen in ActionScript 3.0 nodig heeft.

Opmerking: *Het object traits is een intern implementatiedetail en er wordt niet gegarandeerd dat het niet wordt gewijzigd of wellicht verdwijnt in toekomstige versies van ActionScript.*

Prototypeobject

Elk ActionScript-klassenobject heeft een eigenschap met de naam `prototype`, die een verwijzing is naar het prototypeobject van de klasse. Het prototypeobject stamt uit de tijd dat ActionScript een op prototypen gebaseerde taal was. Zie Geschiedenis van ActionScript OOP-ondersteuning voor meer informatie.

De eigenschap `prototype` is alleen-lezen; dit houdt in dat de eigenschap niet kan worden gewijzigd om naar andere objecten te wijzen. Dit verschilt van de klasseneigenschap `prototype` in vorige versies van ActionScript, het prototype kon opnieuw worden toegewezen zodat het naar een andere klasse wees. Hoewel de eigenschap `prototype` alleen-lezen is, is het prototypeobject waarnaar deze verwijst dit niet. Met andere woorden, er kunnen nieuwe eigenschappen aan het prototypeobject worden toegevoegd. Eigenschappen die aan het prototypeobject zijn toegevoegd, worden door alle instanties van de klasse gedeeld.

De prototypeketen, het enige overervingsmechanisme in eerdere versies van ActionScript, speelt slechts een ondergeschikte rol in ActionScript 3.0. Het primaire overervingsmechanisme, overerving van vaste eigenschappen, wordt intern afgehandeld door het object traits. Een vaste eigenschap is een variabele of methode die is gedefinieerd als deel van een klassendefinitie. De overerving van vaste eigenschappen wordt ook klassenovererving genoemd omdat dit het overervingsmechanisme is dat is gekoppeld aan trefwoorden zoals `class`, `extends` en `override`.

De prototypeketen biedt een alternatief overervingsmechanisme dat dynamischer is dan de overerving van vaste eigenschappen. U kunt eigenschappen niet alleen aan het prototypeobject van een klasse toevoegen als deel van de klassendefinitie, maar ook bij uitvoering via de eigenschap `prototype` van het klassenobject. Als u de compiler echter instelt op een strikte modus, hebt u mogelijk geen toegang tot eigenschappen die aan een prototypeobject zijn toegevoegd tenzij u een klasse declareert met het trefwoord `dynamic`.

Een goed voorbeeld van een klasse waarvan verschillende eigenschappen aan het prototypeobject zijn gekoppeld, is de klasse Object. De methode `toString()` en `valueOf()` van de klasse Object zijn eigenlijk functies die zijn toegewezen aan de eigenschappen van het prototypeobject van de klasse Object. De volgende code is een voorbeeld van hoe de declaratie van deze methoden er in feite moet uitzien (de daadwerkelijke implementatie verschilt iets vanwege de implementatiedetails):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Buiten de klassendefinitie kunt u een eigenschap aan het prototypeobject van een klasse koppelen. De methode `toString()` kan bijvoorbeeld ook als volgt buiten de klassendefinitie Object worden gedefinieerd:

```
Object.prototype.toString = function()
{
    // statements
};
```

In tegenstelling tot overerving van vaste eigenschappen, vereist een prototypeovererving niet het trefwoord `override` als u een methode in een subklasse opnieuw wilt definiëren. Bijvoorbeeld. Als u de methode `valueOf()` opnieuw wilt definiëren in een subklasse van de klasse Object, hebt u drie opties. Ten eerste kunt u een methode `valueOf()` definiëren voor het prototypeobject van de subklasse binnen de klassendefinitie. De volgende code maakt een subklasse van Object met de naam Foo en definieert de methode `valueOf()` opnieuw voor het prototypeobject van Foo als deel van de klassendefinitie. Omdat elke klasse overerft van Object, is het niet nodig het trefwoord `extends` te gebruiken.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Ten tweede kunt u een methode `valueOf()` definiëren voor het prototypeobject van Foo buiten de klassendefinitie, zoals getoond in de volgende code:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Ten slotte kunt u een vaste eigenschap met de naam `valueOf()` definiëren als deel van de klasse Foo. Deze techniek verschilt van de andere technieken in het feit dat overerving van vaste eigenschappen en prototypeovererving door elkaar wordt gebruikt. Als een subklasse van Foo `valueOf()` opnieuw wil definiëren, moet deze het trefwoord `override` gebruiken. De volgende code toont `valueOf()` gedefinieerd als een vaste eigenschap in Foo:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

AS3-naamruimte

De aanwezigheid van twee afzonderlijke overervingsmechanismen, overerving van vaste eigenschappen en prototypeovererving, biedt een interessante uitdaging wat betreft de compatibiliteit ten opzichte van de eigenschappen en methoden van de kernklassen. Compatibiliteit met het concept ECMAScript Language Specification waarop ActionScript is gebaseerd, vereist het gebruik van prototypeovererving. Dit houdt in dat de eigenschappen en methoden van een kernklasse worden gedefinieerd voor het prototypeobject van die klasse. Aan de andere kant vereist compatibiliteit met ActionScript 3.0 het gebruik van de overerving van vaste eigenschappen. Dit betekent dat de eigenschappen en methoden van een kernklasse in de klassendefinitie worden gedefinieerd met gebruik van de trefwoorden `const`, `var` en `function`. Bovendien kan het gebruik van overerving van vaste eigenschappen in plaats van prototypeovererving tot aanzienlijk betere uitvoerprestaties leiden.

Dit probleem is in ActionScript 3.0 opgelost door het gebruik van zowel prototypeovererving als overerving van vaste eigenschappen voor de kernklassen. Elke kernklasse bevat twee sets eigenschappen en methoden. Eén set wordt voor het prototypeobject gedefinieerd om voor compatibiliteit met de ECMAScript-specificatie te zorgen en de andere set wordt met vaste eigenschappen en de AS3-naamruimte gedefinieerd om voor compatibiliteit met ActionScript 3.0 te zorgen.

De AS3-naamruimte biedt een vereenvoudigd mechanisme voor het kiezen tussen de twee sets eigenschappen en methoden. Als u geen gebruik maakt van de AS3-naamruimte, overerft een instantie van de kernklasse de eigenschappen en methoden die zijn gedefinieerd voor het prototypeobject van de kernklasse. Als u de AS3-naamruimte wel gebruikt, overerft een instantie van de kernklasse de AS3-versies, omdat vaste eigenschappen altijd de voorkeur krijgen boven prototype-eigenschappen. Met andere woorden, als er een vaste eigenschap beschikbaar is, wordt deze altijd gebruikt in plaats van een prototype-eigenschap met identieke naam.

U kunt de AS3-naamruimteversie van een eigenschap of methode selectief gebruiken door deze te kwalificeren met de AS3-naamruimte. De volgende code gebruikt bijvoorbeeld de AS3-versie van de methode `Array.pop()`:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Als alternatief kunt u de aanwijzing `use namespace` gebruiken om de AS3-naamruimte te openen voor alle definities binnen een codeblok. De volgende code gebruikt bijvoorbeeld de aanwijzing `use namespace` om de AS3-naamruimte voor de methoden `pop()` en `push()` te openen:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 biedt ook compileropties voor elke set eigenschappen zodat u de AS3-naamruimte op het gehele programma kunt toepassen. De compileroptie `-as3` vertegenwoordigt de AS3-naamruimte en de compileroptie `-es` vertegenwoordigt prototypeovererving (`es` staat voor ECMAScript). Als u de AS3-naamruimte voor het gehele programma wilt openen, stelt u de compileroptie `-as3` in op `true` en de compileroptie `-es` op `false`. Als u het prototypemodel wilt gebruiken, stelt u de compileropties in op de tegenovergestelde waarden. De standaard compileropties voor Flash Builder en Flash Professional zijn `-as3 = true` en `-es = false`.

Als u van plan bent kernklassen uit te breiden en methoden te overschrijven, moet u weten hoe de AS3-naamruimte de manier beïnvloedt waarop u een overschreven methode moet definiëren. Als u de AS3-naamruimte gebruikt, moet een methodeoverschrijving van een kernklassenmethode gebruikmaken van de AS3-naamruimte en van het attribuut `override`. Als u de AS3-naamruimte echter niet gebruikt en u wilt een kernklassenmethode in een subklasse opnieuw definiëren, moet u niet de AS3-naamruimte of het trefwoord `override` gebruiken.

Voorbeeld: GeometricShapes

Met de voorbeeldtoepassing GeometricShapes wordt getoond hoe een aantal objectgeoriënteerde concepten en functies met ActionScript 3.0 kunnen worden toegepast, waaronder:

- Klassen definiëren
- Klassen uitbreiden
- Polymorfisme en het trefwoord `override`

- Interfaces definiëren, uitbreiden en implementeren

Het bevat ook een 'fabrieksmethode' die klasseninstanties maakt, waarmee wordt getoond hoe een geretourneerde waarde moet worden gedeclareerd als een instantie van een interface en hoe het geretourneerde object op een algemene manier wordt gebruikt.

Zie www.adobe.com/go/learn_programmingAS3samples_flash_nl als u de toepassingsbestanden voor dit voorbeeld wilt downloaden. De toepassingsbestanden van GeometricShapes bevinden zich in de map Samples/GeometricShapes. De toepassing bestaat uit de volgende bestanden:

Bestand	**Beschrijving**
GeometricShapes.mxml of GeometricShapes.fla	Het hoofdtoepassingsbestand in Flash (FLA) of Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	De basisinterface die de methoden definieert die door alle toepassingsklassen van GeometricShapes moeten worden geïmplementeerd.
com/example/programmingas3/geometricshapes/IPolygon.as	Een interface die de methoden definieert die moeten worden geïmplementeerd door de GeometricShapes-toepassingsklassen met meerdere zijden.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Een type geometrische vorm met zijden van gelijke lengte die symmetrisch rond het midden van de vorm zijn geplaatst.
com/example/programmingas3/geometricshapes/Circle.as	Een geometrische vorm die een cirkel definieert.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Een subklasse van RegularPolygon die een driehoek met zijden van gelijke lengte definieert.
com/example/programmingas3/geometricshapes/Square.as	Een subklasse van RegularPolygon die een rechthoek met vier zijden van gelijke lengte definieert.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Een klasse die een fabrieksmethode bevat voor het maken van vormen op basis van een opgegeven vormtype en grootte.

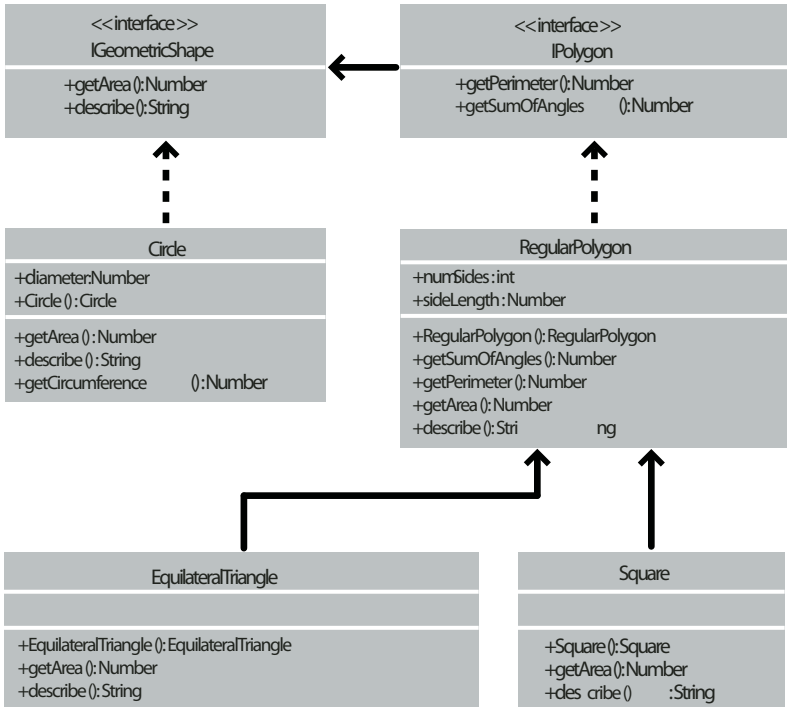
Klassen GeometricShapes definiëren

De toepassing Geometric laat de gebruiker een type geometrische vorm en een grootte opgeven. Vervolgens wordt er gereageerd met een beschrijving van de vorm, het gebied en de afstand rond zijn omtrek.

De gebruikersinterface van de toepassing is eenvoudig; deze bevat een aantal besturingselementen voor het selecteren van het vormtype, het instellen van de grootte en het weergeven van de beschrijving. Het interessante deel van deze toepassing ligt in de structuur van de klassen en de interfaces zelf.

Deze toepassing werkt met geometrische vormen, maar geeft ze niet grafisch weer.

De klassen en interfaces die de geometrische vormen in dit voorbeeld definiëren, worden in het volgende diagram getoond met de Unified Modeling Language-notatie (UML):



Voorbeeldklassen GeometricShapes

Algemeen gedrag met interfaces definiëren

Deze toepassing van GeometricShapes betreft drie typen vormen : cirkels, vierkanten en gelijkzijdige driehoeken. De klassentructuur van GeometricShapes begint met een zeer eenvoudige interface, IGeometricShape, die de methoden opsomt die op alle drie de vormtypen van toepassing zijn:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

De interface definieert twee methoden: de methode `getArea()`, die het gebied van de vorm berekent en retourneert en de methode `describe()`, die een tekstbeschrijving van de vormeigenschappen samenstelt.

De afstand rond de omtrek van elke vorm moet ook worden berekend. De omtrek van een cirkel wordt echter op een unieke manier berekend, zodat het gedrag afwijkt van dat van een driehoek of vierkant. Er bestaan echter nog genoeg overeenkomsten tussen driehoeken, vierkanten en andere veelhoeken, zodat het de moeite waard is een eigen interfaceklasse voor ze te definiëren: IPolygon. De interface IPolygon is ook tamelijk eenvoudig, zoals hieronder getoond:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Deze interface definieert twee methoden die voor alle veelhoeken gelden: de methode `getPerimeter()`, waarmee de gecombineerde lengte van alle zijden wordt gemeten, en de methode `getSumOfAngles()`, waarmee alle binnenhoeken bij elkaar worden opgeteld.

De interface IPolygon breidt de interface IGeometricShape uit, dat betekent dat elke klasse die de interface IPolygon implementeert, alle vier methoden moet declareren; de twee methoden van de interface IGeometricShape en de twee methoden van de interface IPolygon.

Klassen Shape definiëren

Zodra u een goed idee hebt van de methoden die op elk vormtype van toepassing zijn, kunt u de klassen Shape zelf definiëren. Als u wilt weten hoeveel methoden u moet implementeren, kunt u dit baseren op de eenvoudigste vorm; de klasse Circle, zoals hieronder getoond:

```
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;
        }
    }
}
```

De klasse Circle implementeert de interface IGeometricShape, daarom moet deze code bieden voor zowel de methode `getArea()` als de methode `describe()`. Bovendien definieert deze de methode `getCircumference()`, uniek voor de klasse Circle. De klasse Circle declareert ook een eigenschap, `diameter`, die niet in de andere veelhoekklassen voorkomt.

De andere twee typen vormen, vierkanten en gelijkzijdige driehoeken, hebben enkele andere zaken gemeen: ze hebben allebei zijden van gelijke lengte en er zijn formules die u voor beide kunt gebruiken om de omtrek en de som van de binnenhoeken te berekenen. In feite zijn deze algemene formules van toepassing op alle normale veelhoeken die u in de toekomst zult definiëren.

De klasse RegularPolygon wordt de superklasse voor de klasse Square en de klasse EquilateralTriangle. Met een superklasse kunt u algemene methoden op één plaats definiëren, zodat u ze niet afzonderlijk in elke subklasse hoeft te definiëren. Hier is de code voor de klasse RegularPolygon:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

Eerst declareert de klasse RegularPolygon twee eigenschappen die gemeenschappelijk zijn voor alle regelmatige veelhoeken: de lengte van elke zijde (de eigenschap `sideLength`) en het aantal zijden (de eigenschap `numSides`).

De klasse RegularPolygon implementeert de interface IPolygon en declareert alle vier de interfacemethoden van IPolygon. Twee van de methoden (de methoden `getPerimeter()` en `getSumOfAngles()`) worden met algemene formules geïmplementeerd.

Omdat de formule voor de methode `getArea()` verschilt van vorm tot vorm, kan de basisklassenversie van de methode niet de algemene logica opnemen die kan worden overgeërfd door de methoden van de subklassen. In plaats hiervan wordt de standaardwaarde 0 geretourneerd om aan te geven dat het gebied niet is berekend. Als u het gebied van elke vorm correct wilt berekenen, moeten de subklassen van de klasse RegularPolygon de methode `getArea()` zelf overschrijven.

De volgende code voor de klasse EquilateralTriangle toont hoe de methode `getArea()` is overschreven:

```
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;
        }
    }
}
```

Het trefwoord `override` geeft aan dat de methode `EquilateralTriangle.getArea()` met opzet de methode `getArea()` van de superklasse RegularPolygon overschrijft. Wanneer de methode `EquilateralTriangle.getArea()` wordt aangeroepen, berekent deze het gebied met gebruik van de formule in de vorige code en wordt de code in de methode `RegularPolygon.getArea()` nooit uitgevoerd.

De klasse EquilateralTriangle daarentegen definieert zijn eigen versie van de methode `getPerimeter()` niet. Wanneer de methode `EquilateralTriangle.getPerimeter()` wordt aangeroepen, doorloopt de aanroep de overervingsketen naar boven en voert de code in de methode `getPerimeter()` van de superklasse RegularPolygon uit.

De constructor `EquilateralTriangle()` gebruikt de instructie `super()` om expliciet de constructor `RegularPolygon()` van de superklasse aan te roepen. Als beide constructors dezelfde set parameters hebben, kunt u de constructor `EquilateralTriangle()` weglaten en zou de constructor `RegularPolygon()` worden uitgevoerd. De constructor `RegularPolygon()` heeft echter een extra parameter `numSides` nodig. De constructor `EquilateralTriangle()` roept dus `super(len, 3)` aan, die de invoerparameter `len` en de waarde 3 doorloopt om aan te geven dat de driehoek drie zijden heeft.

De methode `describe()` gebruikt ook de instructie `super()`, maar op een andere manier. Deze gebruikt deze instructie voor het aanroepen van de versie van de superklasse RegularPolygon van de methode `describe()`. De methode `EquilateralTriangle.describe()` stelt eerst de tekenreeksvariabele `desc` in op een instructie over het vormtype. Vervolgens haalt de methode het resultaat van de methode `RegularPolygon.describe()` op door `super.describe()` aan te roepen; het resultaat wordt vervolgens toegevoegd aan de tekenreeks `desc`.

De klasse Square wordt hier niet uitgebreid besproken, maar is vergelijkbaar met de klasse EquilateralTriangle, die een constructor en eigen implementaties van de methoden `getArea()` en `describe()` biedt.

Polymorfisme en de fabrieksmethode

Een set klassen die goed gebruikmaakt van interfaces en overerving, kan op veel manieren worden gebruikt. Alle klassen die bijvoorbeeld tot nu toe zijn beschreven, implementeren ofwel de interface IGeometricShape of breiden een superklasse uit die dit doet. Als u dus een variabele definieert als instantie van IGeometricShape, hoeft u niet te weten of deze een instantie van de klasse Circle of Square is om de methode `describe()` aan te roepen.

De volgende code toont hoe dit werkt:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Wanneer `myShape.describe()` wordt aangeroepen, wordt de methode `Circle.describe()` uitgevoerd omdat Circle de onderliggende klasse is, zelfs als de variabele is gedefinieerd als een instantie van de interface IGeometricShape.

In dit voorbeeld ziet u het begrip polymorfisme in actie: exact dezelfde methodeaanroep resulteert in de uitvoering van andere code, afhankelijk van de klasse van het object waarvan de methode wordt aangeroepen.

De toepassing GeometricShapes past de op interface gebaseerde polymorfisme toe met gebruik van een vereenvoudigde versie van een ontwerppatroon, bekend als de fabrieksmethode. De term *fabrieksmethode* verwijst naar een functie die een object retourneert waarvan het onderliggende gegevenstype of inhoud verschilt, afhankelijk van de context.

De klasse GeometricShapeFactory die hier wordt getoond, definieert een fabrieksmethode met de naam `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();
        }
    }
}
```

De fabrieksmethode `createShape()` laat de constructors van de vormsubklasse de details definiëren van de instanties die ze maken, terwijl deze de nieuwe objecten als instanties IGeometricShape retourneert zodat ze op een algemenere manier door de toepassing kunnen worden verwerkt.

De methode `describeShape()` in het vorige voorbeeld toont hoe een toepassing de fabrieksmethode kan gebruiken om een algemene verwijzing op te halen naar een meer specifiek object. De toepassing kan de beschrijving voor een pas gemaakt object Circle ophalen, zoals:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

De methode `describeShape()` roept vervolgens de fabrieksmethode `createShape()` met dezelfde parameters aan en slaat het nieuwe object Circle op in een statische variabele met de naam `currentShape`, die als een object IGeometricShape is ingevoerd. Vervolgens wordt de methode `describe()` aangeroepen voor het object `currentShape` en die aanroep wordt automatisch omgezet om de methode `Circle.describe()` uit te voeren, waardoor een gedetailleerde beschrijving van de cirkel wordt geretourneerd.

Voorbeeldtoepassing verbeteren

De echte kracht van interfaces en overerving wordt zichtbaar wanneer u de toepassing verbetert of wijzigt.

Stel dat u een nieuwe vorm, een vijfhoek, aan deze voorbeeldtoepassing wilt toevoegen. U moet dan een klasse Pentagon maken die de klasse RegularPolygon uitbreidt en zijn eigen versies van de methode `getArea()` en `describe()` definieert. Vervolgens moet u een nieuwe Pentagon-optie toevoegen aan de keuzelijst in de gebruikersinterface van de toepassing. Meer hoeft u niet te doen. De klasse Pentagon krijgt door overerving automatisch de functionaliteit van de methoden `getPerimeter()` en `getSumOfAngles()` van de klasse RegularPolygon. Omdat er wordt overgeërfd van een klasse die de interface IGeometricShape implementeert, kan een instantie Pentagon ook als een instantie IGeometricShape worden behandeld. Dat betekent dat als u een nieuw type vorm wilt toevoegen, u de methodehandtekening van de methoden in de klasse GeometricShapeFactory niet hoeft te wijzigen (en dus ook geen code hoeft te wijzigen die de klasse GeometricShapeFactory gebruikt).

Voeg ter oefening een klasse Pentagon aan het voorbeeld GeometricShapes toe om te zien hoe interfaces en overerving de werkbelasting van het toevoegen van nieuwe functies aan een toepassing kunnen verminderen.