Om ACTIONSCRIPT® 3.0

Juridiska meddelanden

Mer information om juridiska meddelanden finns i http://help.adobe.com/sv_SE/legalnotices/index.html.

Innehåll

Kapitel 1: Introduktion till ActionScript 3.0
ActionScript 1
Fördelar med ActionScript 3.0 1
Nyheter i ActionScript 3.0 1

Kapitel 2: Komma igång med ActionScript
Grunderna i programmering 5
Arbeta med objekt 7
Vanliga programelement 15
Exempel: Del i animeringsportfolio (Flash Professional) 17
Bygga program med ActionScript 19
Skapa egna klasser 23
Exempel: Skapa ett basprogram 25

Kapitel 3: ActionScript-språk och -syntax
Språköversikt 33
Objekt och klasser 34
Paket och namnutrymmen 34
Variabler 44
Datatyper 47
Syntax 59
Operatorer 64
Villkorliga satser 69
Looping 71
Funktioner 74

Kapitel 4: Objektorienterad programmering i ActionScript
Objektorienterad programmering 85
Klasser 85
Gränssnitt 100
Arv 102
Avancerade användningsområden 110
Exempel: GeometricShapes 116

Kapitel 1: Introduktion till ActionScript 3.0

ActionScript

ActionScript är programmeringsspråket för körningsmiljöerna Adobe® Flash® Player och Adobe® AIR™. Med det här språket möjliggörs samverkan, datahantering och mycket mera i Flash-, Flex- och AIR-innehåll och program.

ActionScript körs i ActionScripts virtuella motor (AVM), som ingår i Flash Player och AIR. ActionScript-kod omformas vanligtvis till bytekodformat av en kompilator. (*Bytekod* är en typ av programmeringsspråk som skrivs och tolkas av datorer.) Exempel på kompilatorer är den som är inbyggd i Adobe® Flash® Professional och den som är inbyggd i Adobe® Flash® Builder™ och som är tillgängliga i Adobe® Flex™ SDK. Bytekoden är inbäddad i SWF-filer, som körs av Flash Player och AIR.

ActionScript 3.0 är en robust programmeringsmodell som är bekant för utvecklare med grundläggande kunskaper i objektorienterad programmering. Nedan beskrivs några av huvudfunktionerna i ActionScript 3.0 som utgör förbättringar av tidigare ActionScript-versioner:

- En ny ActionScript Virtual Machine som heter AVM2, som använder en ny bytecode-instruktionsuppsättning och som ger stora prestandaförbättringar
- En mer modern kompilatorkodbas som utför djupare optimeringar än tidigare versioner av kompilatorn
- Ett utökat och förbättrat programmeringsgränssnitt (API), med lågnivåkontroll av objekt och en äkta objektorienterad modell
- Ett XML API baserat på specifikationen ECMAScript för XML (E4X) (ECMA-357 utgåva 2). E4X är ett språktillägg till ECMAScript som infogar XML som intern datatyp för språket.
- En händelsemodell som baseras på händelsespecifikation DOM (Document Object Model) Nivå 3

Fördelar med ActionScript 3.0

ActionScript 3.0 har större skriptmöjligheter än tidigare versioner av ActionScript. Detta gör att det blir enklare att skapa komplexa program med stora dataserier och objektorienterade, återanvändbara kodbaser. ActionScript 3.0 krävs inte för innehåll som körs i Adobe Flash Player. Det medger dock prestandaförbättringar som bara är tillgängliga via AVM2 (den virtuella motorn för ActionScript 3.0). ActionScript 3.0-kod kan köras upp till tio gånger fortare än äldre ActionScript-kod.

Den tidigare versionen av ActionScript Virtual Machine, AVM1, kör ActionScript 1.0- och ActionScript 2.0-kod. Flash Player 9 och 10 har stöd för AVM1 för bakåtkompatibilitet.

Nyheter i ActionScript 3.0

ActionScript 3.0 innehåller många klasser och funktioner som liknar de i ActionScript 1.0 och ActionScript 2.0. ActionScript 3.0 är dock arkitektoniskt och begreppsmässigt annorlunda än tidigare versioner av ActionScript. I ActionScript 3.0 finns nya funktioner i huvudspråket och ett förbättrat API, som ger ökad kontroll över lågnivåobjekt.

Huvudspråksfunktioner

Huvudspråket är grundstenarna i programmeringsspråket, till exempel programsatser, uttryck, villkor, slingor och typer. ActionScript 3.0 innehåller många funktioner som bidrar till att utvecklingsarbetet går fortare.

Körningsundantag

ActionScript 3.0 rapporterar fler feltillstånd än tidigare versioner av ActionScript. Körningsundantag används för vanliga feltillstånd och förbättrar felsökningen, vilket gör att du kan utveckla program som hanterar fel på ett robust sätt. Körningsfel kan förse stackkalkeringar med källfils- och radnummerinformation, vilket gör att du snabbt kan definiera felen.

Körningstyper

I ActionScript 3.0 bevaras typinformation vid körning. Den här informationen används för att utföra typkontroller vid körning, vilket förbättrar systemets typsäkerhet. Typinformation används också för att representera variabler i interna beteckningar, vilket förbättrar prestanda och minskar minnesanvändningen. Som jämförelse är typanteckningar i ActionScript 2.0 huvudsakligen en utvecklarhjälp, och alla värden skrivs dynamiskt vid körning.

Fasta klasser

ActionScript 3.0 omfattar begreppet fasta klasser. En fast klass har bara den fasta uppsättning egenskaper och metoder som definieras vid kompileringen, och ytterligare egenskaper och metoder kan inte läggas till. Eftersom det inte går att ändra en klass vid körning blir kontrollen vid kompilering striktare, vilket skapar mer robusta program. Det förbättrar också minnesanvändningen genom att det inte behövs någon intern hashtabell för varje objektinstans. Dynamiska klasser kan också användas med nyckelordet `dynamic`. Alla klasser i ActionScript 3.0 är som standard fasta, men de kan förklaras vara dynamiska med nyckelordet `dynamic`.

Metodslut

Med ActionScript 3.0 kan ett metodslut automatiskt komma ihåg sin ursprungliga objektinstans. Den här funktionen är användbar vid händelsehantering. I ActionScript 2.0 kommer metodslut inte ihåg vilken objektinstans de extraherats från, vilket leder till oväntat beteende när metodslutet anropas.

ECMAScript för XML (E4X)

ActionScript 3.0 implementerar ECMAScript för XML (E4X), nyligen standardiserat som ECMA-357. E4X har en naturlig, flytande uppsättning språkkonstruktioner för manipulering av XML. I motsats till traditionellt XML-analyserande API:er fungerar XML med E4X som en intern datatyp av språket. E4X effektiviserar utvecklingen av program som ändrar XML genom att mängden kod som krävs drastiskt minskas.

Om du vill läsa ECMA:s E4X-specifikation går du till www.ecma-international.org.

Reguljära uttryck

ActionScript 3.0 innehåller internt stöd för reguljära uttryck så att du snabbt kan söka efter och redigera strängar. ActionScript 3.0 implementerar stöd för reguljära uttryck som definierats i språkdefinitionen ECMAScript utgåva 3 (ECMA-262).

Namnutrymmen

Namnutrymmen liknar de traditionella åtkomstspecifikationssymboler som används för att kontrollera visning av deklarationer (`public`, `private`, `protected`). De fungerar som anpassade åtkomstspecifikationssymboler, och du kan ge dem vilka namn som helst. Namnutrymmen är utrustade med en URI (Universal Resource Identifier) för att undvika kollisioner, och de används också för att representera XML-namnutrymmen när du arbetar med E4X.

Nya primitiva typer

ActionScript 3.0 innehåller tre numeriska typer: Number, int och uint. Number representerar flyttal med dubbel precision. Typen int är ett 32-bitars heltal med tecken som låter ActionScript-koden dra fördel av CPU:ns snabba heltalsmatematikprocessor. Typen int är användbar för slingräknare och variabler där heltal används. Typen uint är ett 32-bitars heltal utan tecken som är användbar för bland annat RGB-färgvärden och byte-räknare. ActionScript 2.0 har däremot bara en enda numerisk typ, Number.

API-funktioner

API:erna i ActionScript 3.0 innehåller många klasser som du använder för att styra objekt på låg nivå. Språkets arkitektur är mer intuitiv än i tidigare versioner. Det finns för många klasser för att beskriva alla i detalj, men en del skillnader bör uppmärksammas.

DOM3-händelsemodell

Händelsemodellen DOM3 (Document Object Model Level 3) utgör en standardmetod för att generera och hantera händelsemeddelanden. Den här händelsemodellen är utformad för att tillåta objekt i program att samverka och kommunicera, bibehålla sina tillstånd och svara på förändringar. ActionScript 3.0-händelsemodellen bygger på World Wide Web Consortiums specifikationer för DOM Level 3-händelser. Den här modellen är tydligare och mer effektiv än de händelsesystem som fanns i tidigare versioner av ActionScript.

Händelser och felhändelser finns i paketet flash.events. Flash Professional-komponenterna och Flex-ramverket använder samma händelsemodell, vilket innebär att Flash Platform har ett enhetligt händelsesystem.

API för visningslista

API:n för åtkomst till visningslistan (det träd som innehåller alla visuella element i programmet) består av klasser som används vid arbete med visuella primitiver.

Klassen Sprite är utformad som en basklass för visuella element, som komponenterna i användargränssnittet. Klassen Shape representerar råvektorformer. Dessa klasser kan instansieras naturligt med operatorn `new` och kan när som helst omplaceras dynamiskt.

Djupet hanteras automatiskt. Metoder finns för angivande och hantering av objekts staplingsordning.

Hantera dynamiska data och innehåll

ActionScript 3.0 innehåller mekanismer för inläsning och hantering av resurser och data i dina program; mekanismer som är intuitiva och konsekventa över hela API:n. Med klassen Loader får du en mekanism för inläsning av SWF-filer och bildresurser samt ett sätt att komma åt detaljerad information om inläst innehåll. Med klassen URLLoader får du en separat mekanism för inläsning av text och binära data i datastyrda program. Med klassen Socket får du ett medel att läsa och skriva binära data till serversocketar i valfritt format.

Åtkomst till lågnivådata

Via olika API:er får du lågnivååtkomst till data. För data som laddas ned ger klassen URLStream åtkomst till data som binära rådata under tiden de laddas ned. Med klassen ByteArray kan du optimera läsning, skrivning och arbete med binära data. Ljud-API:n ger detaljkontroll över ljud via klasserna SoundChannel och SoundMixer. API:er som hanterar säkerhet ger information om säkerhetsbehörighet för en SWF-fil eller inläst innehåll, så att du kan hantera säkerhetsfel.

Arbeta med text

ActionScript 3.0 innehåller ett flash.text-paket för alla textrelaterade API:er. Klassen TextLineMetrics ger detaljerade mått för en textrad i ett textfält. Klassen ersätter metoden `TextFormat.getTextExtent()` i ActionScript 2.0. Klassen TextField innehåller lågnivåmetoder som tillför speciell information om en textrad eller ett enstaka tecken i ett textfält. Metoden `getCharBoundaries()` returnerar till exempel en rektangel som representerar ett teckens begränsningsram. Metoden `getCharIndexAtPoint()` returnerar index för tecknet vid en viss punkt. Metoden `getFirstCharInParagraph()` returnerar index för det första tecknet i ett stycke. Radnivåmetoderna är `getLineLength()`, som returnerar antalet tecken i en angiven textrad, samt `getLineText()`, som returnerar texten på den angivna raden. Klassen Font används för att hantera inbäddade teckensnitt i SWF-filer.

Med klasserna i flash.text.engine-paketet för Flash-textmotorn får du ännu bättre lågnivåkontroll över text. De här klasserna ger lågnivåkontroll över text och har utformats för att skapa ramverk och komponenter för text.

Kapitel 2: Komma igång med ActionScript

Grunderna i programmering

Eftersom ActionScript är ett programmeringsspråk underlättar det om du först lär känna några allmänna programmeringsbegrepp.

Det här gör dataprogram

Först och främst bör du känna till vad ett program är och vad det gör. Det finns två huvudaspekter på ett dataprogram:

- Ett program är en serie instruktioner eller steg som datorn kan utföra.
- Varje steg inbegriper ändring av vissa informations- eller datadelar.

Allmänt sett är ett dataprogram en lista på stegvisa instruktioner som du ger datorn, och som den utför ett steg i taget. Varje separat instruktion kallas en *programsats*. I ActionScript skrivs alla programsatser med ett semikolon på slutet.

En viss programinstruktion ändrar en datadel som är lagrad i datorns minne. Ett enkelt exempel är att instruera datorn att lägga samman två tal och spara resultatet i minnet. Ett mera komplicerat exempel är en instruktion om att flytta en rektangel som redan finns på skärmen. Datorn håller reda på viss information om rektangeln, till exempel x- och y-koordinaterna för dess position, hur bred och hög den är och dess färg. Alla dessa informationsbitar lagras någonstans i datorns minne. Ett program som ska flytta rektangeln till en annan plats måste då innehålla steg som ”ändra x-koordinaten till 200; ändra y-koordinaten till 150”. Det måste med andra ord ange nya värden för x- och y-koordinaterna. Dessa data bearbetas i bakgrunden, med resultatet att bilden visas på den nya positionen på skärmen. På den grundläggande nivån räcker det att veta att ”flytta en rektangel på skärmen” bara innebär att databitar i datorns minne ändras.

Variabler och konstanter

Programmering går i huvudsak ut på att ändra information i datorns minne. Det är följaktligen viktigt att ha ett sätt att representera en bit information i ett program. En *variabel* är ett namn som representerar ett värde i datorns minne. När du skriver programsatser för att ändra värden skriver du variabelns namn i stället för värdet. Varje gång datorn läser variabelnamnet i programmet söker den i minnet och använder det värde som finns där. Om du till exempel har två variabler som heter `value1` och `value2` som båda innehåller ett tal, skriver du följande programsats för att addera dessa tal:

```
value1 + value2
```

När datorn sedan kör instruktionerna läser den värdena i varje variabel och lägger ihop dem.

I ActionScript 3.0 består en variabel av tre olika delar:

- Variabelns namn
- Den typ av data som kan lagras i variabeln
- Det verkliga värdet i datorns minne

Nu har du sett hur datorn använder namnet som en platshållare för värdet. Datatypen är också viktig. När du skapar en variabel i ActionScript anger du den särskilda datatyp som den ska innehålla. Från och med det här steget kan programinstruktionerna bara spara den typen av data i variabeln. Du kan hantera värdet med hjälp av de särskilda egenskaper som är kopplade till dess datatyp. Om du vill skapa en variabel i ActionScript (kallas att *deklarera* värdet) använder du programsatsen `var`:

```
var value1:Number;
```

I det här exemplet instrueras datorn att skapa en variabel med namnet `value1`, som bara kan innehålla Number-data. (”Number” är en särskild datatyp som definieras i ActionScript.) Du kan också lagra ett värde i variabeln på en gång.

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

Adobe Flash Professional
I Flash Professional finns det ett annat sätt att deklarera en variabel. När du placerar en filmklippssymbol, knappsymbol eller ett textfält på scenen kan du ge det ett förekomstnamn i egenskapsinspektören. Flash Professional skapar i bakgrunden en variabel med samma namn som instansnamnet. Du kan använda det namnet i din ActionScript-kod för att representera det scenobjektet. Tänk dig till exempel att du har en filmklippssymbol på scenen och den får instansnamnet `rocketShip`. Varje gång du använder variabeln `rocketShip` i ActionScript-koden hanterar du i själva verket det filmklippet.

En *konstant* påminner om en variabel. Det är ett namn som motsvarar ett värde med en viss datatyp i datorns minne. Skillnaden är att en konstant bara kan tilldelas ett värde en gång i ett ActionScript-program. När en konstants värde har tilldelats är det detsamma hela tiden i programmet. Syntaxen vid deklaration av konstanter är nästan likadan som vid deklaration av variabler. Den enda skillnaden är att du använder nyckelordet `const` i stället för `var`:

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

En konstant är användbar när du vill definiera ett värde som används på flera ställen i ett projekt och som inte ändras under normala förhållanden. Om du använder en konstant i stället för ett litteralvärde blir koden enklare att läsa. Som exempel kan vi använda två versioner av samma kod. I den ena multipliceras ett pris med `SALES_TAX_RATE`. I den andra multipliceras priset med `0.07`. Den version som använder konstanten `SALES_TAX_RATE` är enklare att förstå. Anta dessutom att det värde som definieras av konstanten faktiskt ändras. Om du använder en konstant för att representera det värdet i hela projektet behöver du bara ändra värdet på ett ställe (i konstantdeklarationen). Om du däremot använder hårdkodade litteralvärden måste du ändra värdet på flera ställen.

Datatyper

I ActionScript finns det många datatyper som du kan använda som datatyper för de variabler du skapar. Vissa av dessa datatyper kan betraktas som ”enkla” eller ”grundläggande” datatyper:

- Sträng: ett textvärde, som ett namn på eller texten i en boks kapitel
- Numeric: ActionScript 3.0 innehåller tre speciella datatyper för numeriska data:
  - Antal: alla numeriska värden, däribland värden med eller utan bråkvärde
  - int: ett heltal (ett helt tal utan bråk)
  - uint: ett heltal utan tecken, d.v.s. ett heltal som inte kan vara negativt
- Boolean: ett true- eller false-värde, till exempel om en switch är aktiverad eller om två värden är lika

Dessa enkla datatyper representerar en enstaka informationsdel: en enstaka siffra eller en enstaka textsekvens. De flesta datatyper som definieras i ActionScript är emellertid komplexa datatyper. De representerar en uppsättning med värden i en enda behållare. En variabel med datatypen Date representerar till exempel ett enda värde (en tidpunkt). Oavsett detta representeras det datumvärdet som flera värden: dag, månad, år, timme, minut, sekund och så vidare, och alla är individuella tal. De flesta av oss ser i allmänhet ett datum som ett enda värde, och du kan behandla det så genom att skapa variabeln Date. Men datorn tolkar det däremot som en grupp med flera värden som tillsammans utgör ett enda datum.

De flesta interna datatyper, men även datatyper som definierats av programmerare, är komplexa datatyper. Vissa av de komplexa datatyper som du känner till kan vara:

- MovieClip: en filmklippssymbol
- TextField: ett dynamiskt eller inmatningstextfält
- SimpleButton: en knappsymbol
- Date: information om en enstaka tidpunkt (ett datum eller en tid)

Två ord som ofta används som synonymer för datatyper är ”klass” och ”objekt”. En *klass* är definitionen av en datatyp. Det är som en mall för alla objekt med den datatypen, som talar om att ”alla variabler med datatypen Example har dessa egenskaper: A, B och C”. Ett *objekt* å andra sidan är bara en faktisk instans av en klass. En variabel vars datatyp är MovieClip kan till exempel beskrivas som ett MovieClip-objekt. Följande är alltså olika sätt att säga samma sak.

- Datatypen för variabeln `myVariable` är Number.
- Variabeln `myVariable` är en Number-förekomst.
- Variabeln `myVariable` är ett Number-objekt.
- Variabeln `myVariable` är en förekomst av klassen Number.

Arbeta med objekt

ActionScript är känt som ett objektorienterat programmeringsspråk. Objektorienterad programmering är ett sätt att programmera. Det är helt enkelt ett sätt att ordna kod i ett program med hjälp av objekt.

Termen program definierades tidigare som en rad steg eller instruktioner som datorn utför. Du kan alltså se ett program som en enda lång lista med instruktioner. I objektorienterad programmering delas programinstruktionerna dock upp mellan olika objekt. Koden grupperas i funktionsgrupper, så att liknande funktioner eller information grupperas tillsammans i en behållare.

Adobe Flash Professional

Om du har arbetat med symboler i Flash Professional är du redan bekant med objekt. Låt oss anta att du har definierat en filmklippssymbol, till exempel en ritning av en rektangel, och du har placerat en kopia av den på scenen. Denna filmklippssymbol är också (bokstavligen) ett objekt i ActionScript. Det är en förekomst av klassen MovieClip.

Det finns flera egenskaper för filmklippet som du kan ändra. När det är markerat kan du ändra värdena i egenskapsinspektören, till exempel dess x-koordinat eller dess bredd. Du kan också göra olika färgjusteringar, som att ändra alfa (genomskinlighet) eller lägga till ett skuggfilter. Med andra Flash Professional-verktyg kan du göra fler ändringar, till exempel använda verktyget Omforma fritt för att rotera rektangeln. Alla de här sätten att ändra en filmklippssymbol i Flash Professional finns också i ActionScript. Du ändrar filmklippet i ActionScript genom att ändra de data som alla sammanförs i ett enda paket som kallas ett MovieClip-objekt.

I objektorienterad programmering i ActionScript finns det tre typer av egenskaper som alla klasser kan innehålla:

- Egenskaper
- Metoder
- Händelser

Dessa element används för att hantera de datadelar som används av programmet och för att bestämma vilka åtgärder som ska utföras och i vilken ordning.

Egenskaper

En egenskap representerar en av de datadelar som har paketerats till ett objekt. Ett exempelsångobjekt kan ha egenskaperna `artist` och `title`; klassen MovieClip har egenskaper som `rotation`, `x`, `width` och `alpha`. Du arbetar med egenskaper precis som med enskilda variabler. Vid närmare eftertanke kan du faktiskt tänka på egenskaper som underordnade variabler som finns i ett objekt.

Här är några exempel på ActionScript-kod som innehåller egenskaper. Den här kodraden flyttar ett filmklipp som heter `square` till x-koordinaten 100 pixlar:

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

I den här koden används egenskapen rotation för att rotera filmklippet `square` så att det stämmer överens med rotationen för filmklippet `triangle`:

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

Den här koden ändrar den horisontella skalan för filmklippet `square`, så att det blir en och en halv gång så brett som det var tidigare:

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

Observera den vanliga strukturen: du använder en variabel (`square`, `triangle`) som namn på objektet, följt av en punkt (`.`) och sedan namnet på egenskapen (`x`, `rotation`, `scaleX`). Punkten, som också kallas *punktoperatorn* används för att ange att du öppnar ett av objektets underordnade element. Hela strukturen, ”variabelnamn-punkt-egenskapsnamn”, används som en enda variabel, som namn på ett enstaka värde i datorns minne.

Metoder

En *metod* är en åtgärd som ett objekt kan utföra. Anta till exempel att du har skapat en filmklippssymbol i Flash Professional med flera nyckelrutor och animering på tidslinjen. Det filmklippet kan spelas upp, stoppas eller instrueras att flytta spelhuvudet till en viss bildruta.

I den här koden instrueras filmklippet `shortFilm` att starta uppspelningen:

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

Den här raden gör att filmklippet `shortFilm` avbryter uppspelningen (spelhuvudet stoppar på stället, som när en video pausar):

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

Den här koden gör att filmklippet `shortFilm` flyttar spelhuvudet till Bildruta 1 och stoppar uppspelningen (som när en video spolar tillbaka):

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

Du får alltså tillgång till metoder på samma sätt som egenskaper, d.v.s. genom att skriva objektets namn (en variabel), följt av en punkt och sedan namnet på metoden följt av parenteser. Med parenteserna anger du att du *anropar* metoden eller, med andra ord, instruerar objektet att utföra åtgärden. Ibland placeras värden (eller variabler) inom parenteserna för att skicka ytterligare information som behövs för att åtgärden ska kunna utföras. Dessa värden kallas metod*parametrar*. Metoden `gotoAndStop()` behöver till exempel information om vilken bildruta den ska gå till, och därför krävs en parameter inom parenteserna. Andra metoder, som `play()` och `stop()`, är självförklarande och behöver ingen extra information. De skrivs dock fortfarande inom parentes.

I motsats till egenskaper (och variabler) används metoder inte som platshållare för värden. Vissa metoder kan emellertid utföra beräkningar och returnera ett resultat som kan användas som en variabel. Exempel: Med metoden `toString()` för klassen Number konverteras det numeriska värdet till en textbeteckning:

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

Du använder till exempel metoden `toString()` om du vill visa värdet på en Number-variabel i ett textfält på skärmen. Egenskapen `text` för klassen TextField definieras som en sträng (String), så den kan bara innehålla textvärden. (Texten `property` representerar själva textinnehållet som visas på skärmen.) Den här kodraden konverterar det numeriska värdet i variabeln `numericData` till text. Sedan skickas värdet till skärmen, där det visas i det TextField-objekt som heter `calculatorDisplay`:

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

Händelser

Ett program är en serie instruktioner som datorn utför steg för steg. Vissa enkla program består bara av ett fåtal steg som datorn utför, och sedan avslutas programmet. ActionScript-program fortsätter emellertid att köra och väntar på att användaren matar in något eller att något annat händer. Händelser är den mekanism som avgör vilka instruktioner datorn utför och när.

Händelser är alltså saker som händer och som ActionScript identifierar och kan svara på. Många händelser rör användarinteraktion, till exempel att användaren klickar på en knapp eller trycker på en tangent. Det finns även andra typer av händelser. Om du till exempel använder ActionScript för att läsa in en extern bild, finns det en händelse som du kan använda när bilden är inläst. När ett ActionScript-program körs väntar det i princip bara på att vissa saker ska hända. När dessa saker händer körs den specifika ActionScript-kod som du har angett för de händelserna.

Grundläggande händelsehantering

Tekniken för att ange att vissa åtgärder ska utföras som svar på andra händelser kallas *händelsehantering*. När du skriver ActionScript-kod för att utföra händelsehantering, finns det tre viktiga element som du måste identifiera:

- Händelsekällan: Vilket objekt är det som händelsen ska reagera på? Det kan till exempel vara den knapp användaren klickar på eller det Loader-objekt som ska läsa in bilden. Händelsekällan kallas även *händelsemål*. Den har fått det här namnet eftersom källan är målet för händelsen (d.v.s. där händelsen sker).
- Händelsen: Vad är det som kommer att hända; den sak som du vill svara på? Det är viktigt att identifiera den specifika händelsen, eftersom många objekt utlöser flera händelser.
- Svaret: Vilka åtgärder ska utföras när händelsen inträffar?

När du skriver ActionScript-kod som ska hantera händelser är dessa tre element nödvändiga. Koden följer den här grundläggande strukturen (element i fet stil är platshållare som du fyller i efter behov):

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

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

Den här koden gör två saker. Först definieras en funktion, vilket är ett sätt att ange de åtgärder som ska utföras som svar på händelsen. Därefter anropas metoden `addEventListener()` för källobjektet. Det här anropet till `addEventListener()` innebär i princip att funktionen ”prenumererar” på den angivna händelsen. När händelsen inträffar utförs funktionens åtgärder. Vi ska beskriva dessa delar mer ingående.

Med en *funktion* kan du gruppera flera åtgärder med ett enda namn, som fungerar som en genväg till åtgärderna. En funktion är samma sak som en metod, men med den skillnaden att den inte nödvändigtvis är kopplad till en viss klass. (Termen ”metod” kan faktiskt definieras som en funktion som är kopplad till en särskild klass.) När du skapar en funktion för händelsehantering väljer du ett namn för funktionen (i det här fallet `eventResponse`). Du anger också en parameter (i det här exemplet `eventObject`). Att ange en funktionsparameter är som att deklarera en variabel, och du måste också ange parameterns datatyp. (I det här exemplet är parameterns datatyp `EventType`.)

Alla typer av händelser som du vill avlyssna måste ha en ActionScript-klass kopplad till sig. Datatypen som du anger för funktionsparametern är alltid den klass som kopplas till händelsen som du vill svara på. En `click`-händelse (utlöses när användaren klickar på ett objekt med musen) kopplas till exempel till klassen MouseEvent. Om du vill skriva en avlyssnarfunktion för en `click`-händelse definierar du avlyssnarfunktionen med en parameter med datatypen MouseEvent. Mellan den inledande och avslutande klammerparentesen (`{ ... }`) skriver du de instruktioner som du vill att datorn ska utföra när händelsen inträffar.

Nu har du skrivit händelsehanteringsfunktionen. Härnäst ska du tala om för händelsekällobjektet (det objekt som händelsen utförs på, till exempel knappen) att det ska anropa din funktion när händelsen inträffar. Du registrerar funktionen hos händelsekällobjektet genom att anropa metoden `addEventListener()` för det objektet (alla objekt som har händelser har också en `addEventListener()`-metod). Metoden `addEventListener()` används med två parametrar:

- Först namnet på den speciella händelse som du vill svara på. Varje händelse är kopplad till en viss klass. Varje händelseklass har ett visst värde (som fungerar som ett unikt namn) definierat för var och en av dess händelser. Du använder det värdet för den första parametern.
- Sedan namnet på händelsens svarsfunktion. Observera att ett funktionsnamn skrivs utan parenteser när det skickas som en parameter.

Händelsehanteringsprocessen

Följande är en stegvis beskrivning av den process som inträffar när du skapar en händelseavlyssnare. I det här fallet visas ett exempel på hur du skapar en avlyssnarfunktion som anropas när objektet `myButton` klickas.

Den aktuella koden som skrivs av programmet är:

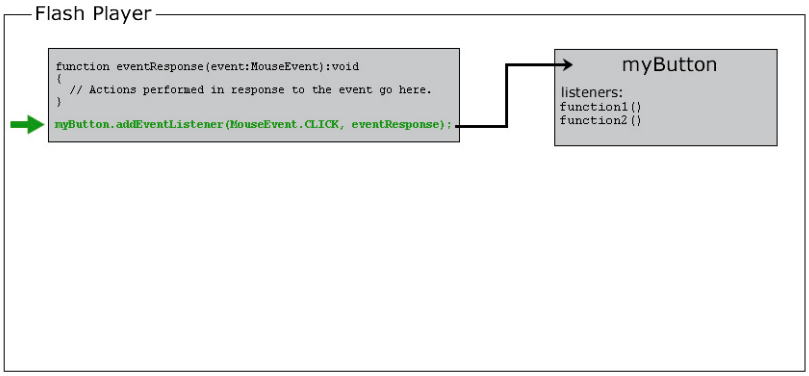
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

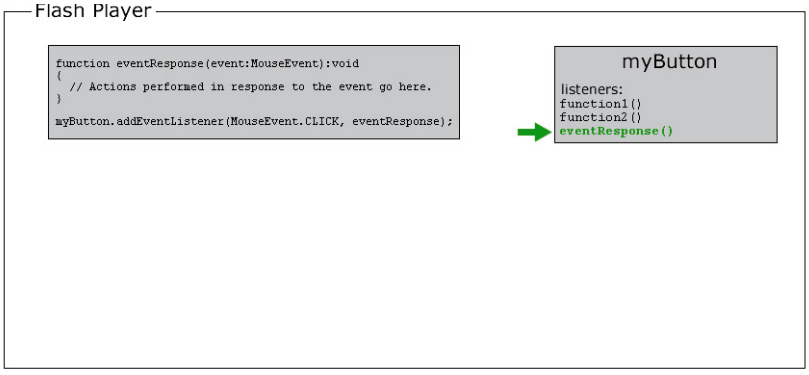
Så här fungerar koden när den körs:

1 När SWF-filen läses in identifieras funktionen `eventResponse()` av datorn.

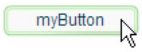
2 Koden körs då (närmare bestämt de kodrader som inte finns i en funktion). I det här fallet är det bara en kodrad: anropa metoden `addEventListener()` för händelsekällans objekt (som heter `myButton`) och skicka funktionen `eventResponse` som en parameter.



Internt har `myButton` en lista över funktioner som lyssnar på var och en av dess händelser. När dess `addEventListener()`-metod anropas sparar `myButton` funktionen `eventResponse()` i sin lista över händelseavlyssnare.

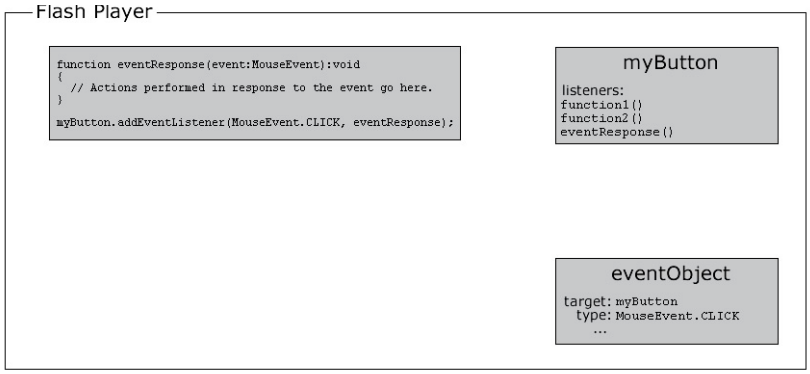


3 I ett visst läge klickar användaren på objektet `myButton` vilket utlöser händelsen `click` (identifieras som `MouseEvent.CLICK` i koden).

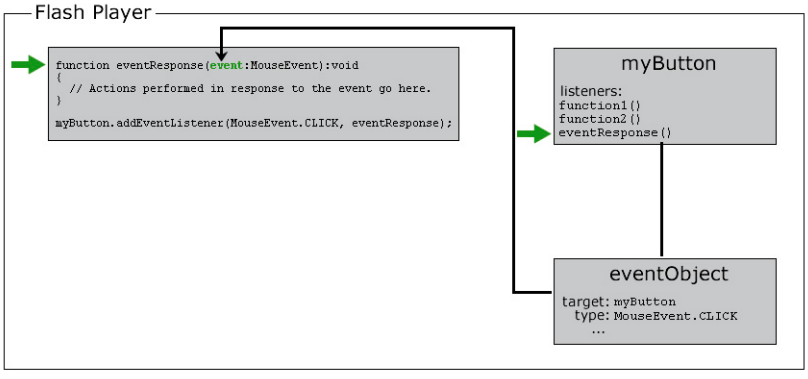


I det läget händer följande:

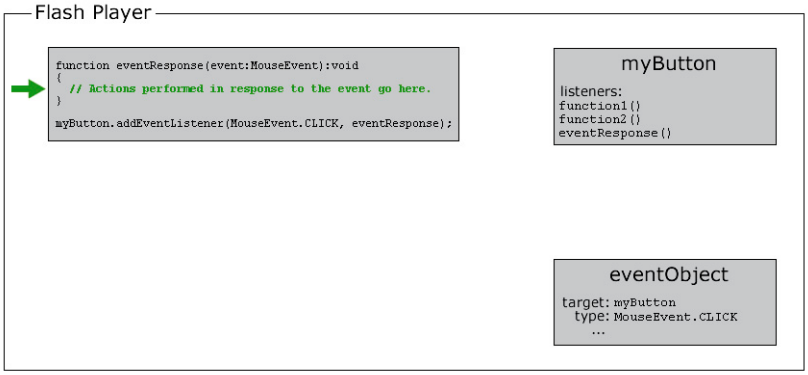
a Ett objekt, som är en instans av den klass som är kopplad till den aktuella händelsen (MouseEvent i det här exemplet), skapas. För många händelser är det här objektet en instans av klassen Event. För mushändelser är det en MouseEvent-instans. För andra händelser är det en instans av den klass som är koppad till händelsen. Objektet som skapas kallas *händelseobjekt* och innehåller speciell information om händelsen som inträffade: vilken typ av händelse det är, var den inträffade och annan händelsespecifik information.



b Datorn letar sedan i listan över händelseavlyssnare som sparas av `myButton`. Varje funktion gås igenom en i taget, varje funktion anropas och händelseobjektet skickas till funktionen som en parameter. Eftersom funktionen `eventResponse()` är en avlyssnare till `myButton` anropar datorn funktionen `eventResponse()` som en del av den här processen.



c När funktionen `eventResponse()` anropas körs koden i funktionen, så att dina angivna åtgärder utförs.



Exempel på händelsehantering

Nedan följer några mer konkreta exempel på kod för händelsehantering. Dessa exempel ger dig förhoppningsvis en bild av en del vanliga händelseelement och olika varianter som du kan prova när du skriver händelsehanteringskod:

- Klicka på en knapp för att starta uppspelningen av aktuellt filmklipp. I följande exempel är `playButton` förekomstnamn på knappen och `this` är ett specialnamn som betyder ”aktuellt objekt”:

```
this.stop();

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

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

- Identifiera skrift i ett textfält. I det här exemplet är `entryText` ett inmatningsfält och `outputText` ett dynamiskt textfält:

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

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

- Klicka på en knapp för att gå till en URL. I det här fallet är `linkButton` förekomstnamnet på knappen:

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

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

Skapa objektsförekomster

Innan du kan använda ett objekt i ActionScript måste det naturligtvis finnas till. En del av att skapa ett objekt är att deklarera en variabel, men detta skapar bara en tom plats i datorns minne. Tilldela alltid variabeln ett verkligt värde (d.v.s. skapa ett objekt och lagra det i variabeln) innan du försöker använda eller hantera den. Själva processen att skapa ett objekt kallas även att *instansiera* objektet. Du skapar med andra ord en instans av en viss klass.

Ett enkelt sätt att skapa en objektsförekomst inbegriper inte ActionScript alls. I Flash Professional placerar du en filmklippssymbol eller ett textfält på scenen och tilldelar det ett instansnamn. En variabel med det instansnamnet deklareras då automatiskt i Flash Professional, en objektinstans skapas och objektet sparas i variabeln. På ett liknande sätt skapar du i Flex en komponent i MXML genom att koda en MXML-tagg eller genom att placera komponenten i redigeraren i designläget i Flash Builder. När du tilldelar ett ID till komponenten blir det ID:t namnet på en ActionScript-variabel som innehåller den komponentinstansen.

Men du vill kanske inte alltid skapa ett objekt visuellt, och för icke-visuella objekt kan du inte göra det. Du kan skapa objektinstanser på flera andra sätt enbart med ActionScript.

Med flera olika ActionScript-datatyper kan du skapa en instans med hjälp av ett *litteraluttryck*, ett värde som skrivs direkt i ActionScript-koden. Här är några exempel:

- Litteralt numeriskt värde (ange talet direkt):

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

- Litteralt strängvärde (omge texten med citattecken):

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

- Litteralt booleskt värde (använd de litterala värdena `true` eller `false`):

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

- Litteralt Array-värde (radbryt en kommaavgränsad lista med värden inom hakparentes):

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

- Litteralt XML-värde (ange XML direkt):

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

I ActionScript definieras litterala uttryck för datatyperna Array, RegExp, Object och Function.

Det vanligaste sättet att skapa en instans för en datatyp är att använda operatorn `new` med klassnamnet, enligt nedanstående:

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

Att skapa ett objekt med hjälp av operatorn `new` beskrivs ofta som att ”anropa klassens konstruktor”. En *konstruktor* är en speciell metod som anropas som en del av processen med att skapa en förekomst av en klass. Tänk på att du sätter parenteser efter klassnamnet när du skapar en instans på det här sättet. Ibland anger du parametervärden innanför parenteserna. Det finns två saker som du också gör när du anropar en metod.

Du kan använda operatorn `new` för att skapa en objektinstans även för sådana datatyper, som medger att du skapar instanser med hjälp av litterala uttryck. De här två kodraderna gör till exempel exakt samma sak:

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

Det är viktigt att du känner till hur du skapar objekt med `new` *ClassName*`()`. Många ActionScript-datatyper kan inte representeras visuellt. De kan därför inte heller skapas genom att du placerar ett objekt på scenen i Flash Professional eller via designläget i MXML-redigeraren i Flash Builder. Du kan bara skapa en instans av sådana datatyper i ActionScript med operatorn `new`.

Adobe Flash Professional

I Flash Professional kan operatorn `new` också användas för att skapa en instans av en filmklippssymbol, som har definierats i biblioteket men inte placerats på scenen.

Fler hjälpavsnitt

Arbeta med arrayer

Använda reguljära uttryck

Skapa MovieClip-objekt med ActionScript

Vanliga programelement

Det finns ytterligare några ”byggklossar” som du använder för att skapa ett ActionScript-program.

Operatorer

Operatorer är speciella symboler (eller ibland ord) som används för att utföra beräkningar. De används mest för matematiska beräkningar men även när man vill jämföra värden med varandra. I allmänhet gäller att en operator använder ett eller flera värden för att ”arbeta fram” ett enda resultat. Till exempel:

- Additionsoperatorn (`+`) lägger ihop två värden, vilket resulterar i ett enstaka tal:

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

- Multiplikationsoperatorn (`*`) multiplicerar ett värde med ett annat, vilket resulterar i ett enstaka tal:

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

- Likhetsoperatorn (`==`) jämför två värden för att se om de är lika, vilket resulterar i ett enstaka booleskt true- eller false-värde:

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

  Som du ser här används likhetsoperatorn och de andra jämförelseoperatorerna vanligtvis med programsatsen `if` för att bestämma om vissa instruktioner utförs eller inte.

Kommentarer

När du skriver ActionScript-kod kan du behöva skapa minnesanteckningar till dig själv. Ibland vill du kanske förklara hur vissa kodrader fungerar eller varför du har gjort på ett visst sätt. *Kodkommentarer* är ett verktyg som du använder för att skriva text som datorn ignorerar i koden. ActionScript innehåller två olika sorters kommentarer:

- Enkelradskommentar: en enkelradskommentar utmärkts med två snedstreck var som helst på raden. Allt efter snedstrecken till slutet av den aktuella raden ignoreras:

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

- Flerradskommentarer: en flerradskommentar innehåller en startkommentarsmarkering (`/*`), kommentarinnehållet och till sist en slutkommentarsmarkering (`*/`). Allt mellan den inledande och den avslutande markeringen ignoreras, oavsett hur många rader kommentaren sträcker sig över:

  ```
  /*
  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.
  */
  ```

Ett annat vanligt sätt att använda kommentarer är för att tillfälligt ”stänga av” en eller flera kodrader. Du kan till exempel använda kommentarer om du testar ett nytt sätt att göra något. Du kan också använda dem om du behöver försöka lista ut varför ActionScript-kod inte fungerar som du har tänkt dig.

Flödeskontroll

I ett program kan du till exempel behöva upprepa vissa åtgärder, utföra vissa åtgärder men inte andra eller utföra alternativa åtgärder beroende på olika villkor. *Flödeskontroll är en kontroll över vilka åtgärder som ska utföras.* Det finns flera typer av flödeskontrollselement i ActionScript.

- Funktioner: Funktioner är som genvägar. De är ett sätt att gruppera en serie åtgärder under ett enda namn, och de kan användas för att utföra beräkningar. Funktioner behövs för att hantera händelser, men används också som ett allmänt verktyg för att gruppera en serie instruktioner.
- Slingor: Med slingstrukturer kan du definiera en uppsättning instruktioner som datorn utför ett visst antal gånger eller tills vissa villkor ändras. Slingor används ofta för att ändra olika relaterade objekt och då används en variabel vars värde ändras varje gång datorn arbetar igenom slingan.
- Villkorssatser: Villkorssatser är ett sätt att beteckna vissa instruktioner som bara utförs under vissa förhållanden (villkor). De används även för att tillhandahålla alternativa instruktioner för olika villkor. Den vanligaste villkorliga programsatsen är `if`. Med programsatsen `if` kontrolleras ett värde eller uttryck inom parentes. Om värdet är `true` utförs kodraderna inom klammerparenteserna. I annat fall ignoreras de. Till exempel:

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

  Programsatsen `if` har en följeslagare, programsatsen `else`, som du använder för att definiera alternativa instruktioner som ska utföras om villkoret inte är `true`:

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

Exempel: Del i animeringsportfolio (Flash Professional)

I det här exemplet får du en första möjlighet att lära dig hur du sätter samman bitar av ActionScript till ett fullständigt program. Det här kodstycket är ett exempel på hur du kan ta en befintlig linjär animering och lägga till några små interaktiva element. Du kan till exempel bygga in en animering som har skapats för en kund i en onlineportfölj. Det interaktiva beteendet som du lägger till i animeringen inbegriper två knappar som användaren kan klicka på: en för att starta animeringen och en för att navigera till en separat URL (som portföljmenyn eller författarens hemsida).

Processen med att skapa den här lilla biten kan delas upp i följande huvudavsnitt:

1. Förbereda FLA-filen för att lägga till ActionScript och interaktiva element.
2. Skapa och lägga till knapparna.
3. Skriva ActionScript-koden.
4. Testa programmet.

Förbereda för att lägga till interaktivitet

Innan du kan lägga till interaktiva element i animeringen bör du förbereda FLA-filen genom att skapa vissa platser där det nya innehållet kan infogas. I detta ingår att skapa faktiskt utrymme på scenen, där knapparna kan placeras. Det inbegriper även att skapa ”utrymme” i FLA-filen så att olika objekt kan hållas isär.

Så här förbereder du FLA-filen för att lägga till interaktiva element:

1. Skapa en FLA-fil med en enkel animering, till exempel en rörelseinterpolering eller en forminterpolering. Om du redan har en FLA-fil med den animering som du visar i projektet öppnar du den filen och sparar den med ett nytt namn.
2. Bestäm var på skärmen du vill att de två knapparna ska visas. En knapp används för att starta animeringen och den andra för att länka till författarens portfölj eller hemsida. Om det är nödvändigt kan du rensa eller lägga till utrymme på scenen för det nya innehållet. Du kan skapa en startbild i den första bildrutan om animeringen inte innehåller någon sådan. I så fall vill du förmodligen flytta animeringen, så att den börjar vid bildruta 2 eller senare.
3. Lägg till ett nytt lager ovanför de andra lagren i tidslinjen och ge det namnet **buttons**. Det är i det här lagret du ska lägga till knapparna.
4. Lägg till ett nytt lager ovanför buttons-lagret och kalla det **actions**. I det här lagret lägger du till ActionScript-kod i programmet.

Skapa och lägga till knappar

Sedan skapar och placerar du knapparna som utgör själva stommen i det interaktiva programmet.

Så här skapar och lägger du till knappar i FLA-filen:

1. Använd ritverktygen och skapa den första knappens utseende (knappen ”play”) på lagret buttons. Du kan till exempel rita en vågrät oval med text ovanpå.
2. Använd markeringsverktyget för att markera alla grafikdelar i knappen.
3. Välj Ändra > Konvertera till symbol på huvudmenyn.
4. I dialogrutan väljer du Knapp som symboltyp, ge symbolen ett namn och klicka på OK.
5. När knappen är markerad ger du den förekomstnamnet **playButton** i egenskapsinspektören.
6. Upprepa steg 1 till 5 för att skapa knappen som tar användaren till författarens hemsida. Ge knappen namnet **homeButton**.

Skriva koden

ActionScript-koden för det här programmet kan delas upp i tre funktionsuppsättningar, även om all kod införs på samma plats. De tre saker som koden utför är:

- Stoppa spelhuvudet så snart som SWF-filen lästs in (när spelhuvudet kommer till bildruta 1).
- Lyssna efter en händelse som startar uppspelningen av SWF-filen när användaren klickar på knappen Spela upp.
- Lyssna efter en händelse som skickar webbläsaren till rätt URL när användaren klickar på författarens hemsidesknapp.

Så här skapar du kod som stoppar spelhuvudet när det kommer till bildruta 1:

1. Markera nyckelbildrutan i bildruta 1 på åtgärdslagret.
2. Om du vill öppna åtgärdspanelen på huvudmenyn väljer du Fönster > Åtgärder.
3. Ange följande kod i skriptrutan:

```
stop();
```

Så här skriver du kod som startar animeringen när du klickar på Spela upp:

1. I slutet av koden som anges i föregående steg lägger du till två tomma rader.
2. Ange följande kod längst ned i skriptet:

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

   Den här koden definierar funktionen `startMovie()`. När `startMovie()` anropas resulterar det i att huvudtidslinjen börjar uppspelningen.

3. På raden som kommer efter koden som lagts till i föregående steg skriver du in följande kodrad:

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

   Den här kodraden registrerar funktionen `startMovie()` som en avlyssnare för händelsen `click` till `playButton`. Det betyder att när någon klickar på knappen `playButton` anropas funktionen `startMovie()`.

Så här skriver du kod som skickar webbläsaren till en URL när du klickar på hemsidesknappen:

1. I slutet av koden som anges i föregående steg lägger du till två tomma rader.
2. Ange den här koden längst ned i skriptet:

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

Den här koden definierar funktionen `gotoAuthorPage()`. Den här funktionen skapar först en URLRequest-instans, som representerar webbadressen http://example.com/. Sedan skickas webbadressen till funktionen `navigateToURL()`, vilket innebär att användarens webbläsare öppnar adressen.

3 På raden som kommer efter koden som lagts till i föregående steg skriver du in följande kodrad:

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

Den här kodraden registrerar funktionen `gotoAuthorPage()` som en avlyssnare för händelsen `click` till `homeButton`. Det betyder att när någon klickar på knappen `homeButton` anropas funktionen `gotoAuthorPage()`.

Testa programmet

Nu fungerar programmet som det ska. Låt oss testa och se om det fungerar.

Så här testar du programmet:

1 Välj Kontroll > Testa filmen. SWF-filen skapas i Flash Professional och öppnas i ett Flash Player-fönster.

2 Testa att båda knapparna fungerar som du tänkt dig.

3 Om de inte fungerar kan du kontrollera följande:

- Har båda knapparna tydliga förekomstnamn?
- Använder metodanropen för `addEventListener()` samma namn som knapparnas förekomstnamn?
- Används rätt händelsenamn i metodanropen för `addEventListener()`?
- Har rätt parameter angetts för varje funktion? (Båda metoderna måste ha en enda parameter med datatypen MouseEvent.)

Alla dessa fel (och de flesta andra tänkbara fel) ger upphov till ett felmeddelande. Felmeddelandet kan visas antingen när du väljer kommandot Testa filmen eller när du klickar på knappen samtidigt som du testar projektet. Du kan kontrollera kompileringsfel på panelen Kompilatorfel (de som inträffar när du först väljer Testa filmen). Använd utdatapanelen för att kontrollera körningsfel som inträffar när innehållet spelas upp, till exempel när du klickar på en knapp.

Bygga program med ActionScript

Processen med att skriva ActionScript för att bygga ett program innebär mer än att känna till syntax och namn på de klasser som ska användas. Merparten av dokumentationen för Flash Platform behandlar de områdena (syntax och användning av ActionScript-klasser). Om du vill bygga ett ActionScript-program behöver du emellertid också känna till lite om bland annat följande:

- Vilka program kan användas för att skriva ActionScript?
- Hur ordnar jag ActionScript-koden?
- Hur inkluderar jag ActionScript-kod i ett program?
- Vilka steg ska jag utföra när jag utvecklar ett ActionScript-program?

Alternativ för att organisera koden

Du kan använda ActionScript 3.0-kod för att driva allt från enkla grafikanimeringar till komplicerade transaktionsbearbetningssystem för klient-server. Beroende på vilken typ av program du bygger kan du använda dig av ett eller flera av dessa sätt för att inkludera ActionScript i projektet.

Lagra kod i bildrutor i en Flash Professional-tidslinje

I Flash Professional kan du lägga till ActionScript-kod i vilken bildruta som helst i tidslinjen. Koden körs när filmen spelas upp, när spelhuvudet kommer till den aktuella bildrutan.

Ett enkelt sätt att lägga till beteende i program som byggs i Flash Professional är att placera ActionScript-kod i bildrutor. Du kan lägga till kod i vilken bildruta som helst i huvudtidslinjen eller i en bildruta i tidslinjen för en MovieClip-symbol. Den här flexibiliteten kostar emellertid något. När du bygger stora program glömmer du lätt bort vilka skript som ligger i vilka bildrutor. Den här komplicerade strukturen kan medföra att programmet efter ett tag blir svårt att underhålla.

Många utvecklare förenklar organisationen av sin ActionScript-kod i Flash Professional genom att placera kod bara i den första bildrutan i en tidslinje eller på ett speciellt lager i Flash-dokumentet. Om du skiljer ut koden blir det lättare att leta upp och underhålla koden i Flash FLA-filerna. Du kan däremot inte använda samma kod i ett annat Flash Professional-projekt utan att kopiera och klistra in koden i en ny fil.

Om du vill göra det enklare att använda ActionScript-koden i andra Flash Professional-projekt i framtiden sparar du koden i externa ActionScript-filer (textfiler med filtillägget .as).

Bädda in kod i Flex MXML-filer

I en Flex-utvecklingsmiljö som Flash Builder kan du inkludera ActionScript-kod inuti en `<fx:Script>`-tagg i en Flex MXML-fil. Den här tekniken kan dock göra stora projekt mer komplicerade och försvåra användningen av samma kod i ett annat Flex-projekt. Om du vill göra det enklare att använda ActionScript-koden i andra Flex-projekt i framtiden sparar du koden i externa ActionScript-filer.

Obs! *Du kan ange en källparameter för en `<fx:Script>`-tagg. Med hjälp av en källparameter kan du ”importera” ActionScript-kod från en befintlig fil som om den har skrivits direkt i `<fx:Script>`-taggen. Källfilen som du använder kan emellertid inte definiera sin egen klass, vilket begränsar återanvändbarheten.*

Lagra kod i ActionScript-filer

Om ditt projekt innehåller en mängd ActionScript-kod bör du organisera koden i ActionScript-källfiler (textfiler med filtillägget .as). En ActionScript-fil kan struktureras på ett av två sätt, beroende på hur du tänker använda den i programmet.

- Ostrukturerad ActionScript-kod: Rader med ActionScript-kod, däribland programsatser och funktionsdefinitioner, skrivna som om de hade skrivits direkt i ett tidslinjeskript eller en MXML-fil.

  Du kommer åt ActionScript som skrivits på det här sättet med hjälp av programsatsen `include` i ActionScript eller med taggen `<fx:Script>` i Flex MXML. Programsatsen `include` i ActionScript talar om för kompilatorn att innehållet i en extern ActionScript-fil, på en viss plats och inom ett givet intervall, ska inkluderas i ett skript. Resultatet är samma som om koden skrivits där direkt. När du använder en `<fx:Script>`-tagg med ett källattribut i MXML-språk identifierar detta ett externt ActionScript, som kompilatorn läser in vid den tidpunkten i programmet. Med följande tagg läses till exempel den externa ActionScript-filen Box.as in:

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

- ActionScript:definition av klass En definition av en ActionScript-klass, inklusive dess metod- och egenskapsdefinitioner.

När du definierar en klass kan du komma åt ActionScript-koden i klassen genom att skapa en instans av klassen och använda dess egenskaper, metoder och händelser. Det är ingen skillnad på att använda egna klasser eller inbyggda ActionScript-klasser, båda kräver två saker:

- Använd programsatsen `import` för att ange klassens fullständiga namn, så att ActionScript-kompilatorn vet var den finns. Om du till exempel vill använda klassen MovieClip i ActionScript importerar du klassen med dess fullständiga namn, inklusive paket och klass:

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

  Du kan också importera paketet som innehåller klassen MovieClip, vilket är detsamma som att skriva separata `import`-programsatser för varje klass i paketet:

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

  Det enda undantaget för regeln att en klass måste importeras för att du ska kunna använda den klassen i koden är klasser på den översta nivån. De klasserna definieras inte i ett paket.

- Skriv kod som uttryckligen använder klassnamnet. Deklarera till exempel en variabel med den klassen som datatyp och skapa en instans av klassen att spara i variabeln. Om du använder en klass i ActionScript-kod instruerar du kompilatorn att läsa in definitionen av den klassen. Om det till exempel finns en extern klass som heter Box skapar den här programsatsen en instans av klassen Box:

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

  Första gången kompilatorn träffar på referensen till klassen Box söker den i den tillgängliga källkoden efter definitionen för klassen Box.

Välja rätt verktyg

Du kan använda ett av flera verktyg (eller flera verktyg tillsammans) för att skriva och redigera ActionScript-koden.

Flash Builder

Adobe Flash Builder är det bästa verktyget för att skapa projekt med Flex-ramverket eller projekt som huvudsakligen består av ActionScript-kod. Flash Builder innehåller också en komplett ActionScript-redigerare samt funktioner för visuell layout och MXML-redigering. Det kan användas för att skapa Flex-projekt eller projekt som enbart innehåller ActionScript. Flex har flera fördelar, däribland en mängd förbyggda användargränssnittskontroller, flexibla dynamiska layoutkontroller och inbyggda mekanismer för arbete med fjärrdatakällor och länkning av externa data till användargränssnittselement. På grund av den extra kod som krävs för de här funktionerna kan Flex-projekt få större SWF-filer än motsvarande projekt som inte görs i Flex.

Använd Flash Builder om du skapar omfattande datadrivna RIA-program (Rich Internet Applications) med Flex. Använd det när du vill redigera ActionScript-kod, redigera MXML-kod och skapa programmets visuella layout – allt i ett och samma verktyg.

Många Flash Professional-användare som bygger projekt med mycket ActionScript använder Flash Professional för att skapa visuella resurser och Flash Builder som en redigerare för ActionScript-koden.

Flash Professional

Flash Professional innehåller, förutom funktioner för att skapa grafik och animeringar, även verktyg för att arbeta med ActionScript-kod. Koden kan antingen bifogas element i en FLA-fil eller i externa filer som bara innehåller ActionScript. Flash Professional är perfekt för projekt med mycket animeringar eller video. Det är praktiskt när du vill skapa de flesta grafiska resurserna själv. Ett annat skäl att använda Flash Professional för att utveckla ActionScript-projekt är att du kan skapa visuella resurser och skriva kod i ett och samma program. Flash Professional innehåller även förbyggda användargränssnittskomponenter. Du kan använda de komponenterna för att minska storleken på SWF-filen och använda visuella verktyg för att skapa skal för dem i projektet.

Flash Professional innehåller två verktyg för ActionScript-kod:

- Åtgärdspanelen: Den här panelen är tillgänglig när du arbetar i en FLA-fil och du kan skriva ActionScript-kod som bifogas till bildrutor i en tidslinje.
- Skriptfönster: Skriptfönstret är en anpassad textredigerare för ActionScript-kodfiler (.as).

Tredjeparts ActionScript-redigerare

Eftersom ActionScript-filer (.as) lagras som rena textfiler kan alla program som hanterar rena textfiler användas för att skriva ActionScript-filer. Förutom Adobes ActionScript-produkter finns det åtskilliga tredjeparts textredigeringsprogram med ActionScript-specifik kapacitet. Du kan skriva en MXML-fil och ActionScript-klasser med vilket textredigerare som helst. Du kan sedan skapa ett program från de filerna med Flex SDK. Projektet kan använda Flex eller vara ett program som bara består av ActionScript. En del utvecklare använder Flash Builder, eller ActionScript-redigerare från andra tillverkare, för att skriva ActionScript-klasser och Flash Professional för att skapa grafiskt innehåll.

Skälen till att välja en ActionScript-redigerare från en annan tillverkare kan vara:

- Du föredrar att skriva ActionScript-kod i ett separat program och utforma visuella element i Flash Professional.
- Du använder ett program för icke-ActionScript-programmering (till exempel när du skapar HTML-sidor eller bygger program i ett annat programmeringsspråk), och du vill använda samma program för ActionScript-kodningen.
- Du vill skapa ActionScript- eller Flex-projekt med Flex SDK utan Flash Professional or Flash Builder.

Kodredigerare som stöder ActionScript är:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (med ActionScript- och Flex-paket)

ActionScript-utvecklingsprocessen

Oavsett om ActionScript-projektet är stort eller litet blir arbetet med att utforma och utveckla programmet mer effektivt om du använder en process. I följande avsnitt beskrivs den grundläggande utvecklingsprocessen för att skapa ett program som använder ActionScript 3.0:

1 Utforma programmet.

Beskriv programmet på något sätt innan du börjar bygga det.

2 Skapa ActionScript 3.0-koden.

   Du kan skapa ActionScript-kod med hjälp av Flash Professional, Flash Builder, Dreamweaver eller en textredigerare.

3 Skapa ett Flash- eller Flex-projekt för att köra koden.

   I Flash Professional skapar du en FLA-fil, anger publiceringsinställningar, lägger till användargränssnittskomponenter i programmet och refererar till ActionScript-koden. I Flex definierar du programmet, lägger till användargränssnittskomponenter med MXML och refererar till ActionScript-koden.

4 Publicera och testa ActionScript-programmet.

   När du testar programmet kör du det inifrån utvecklingsmiljön och kontrollerar att det utför allt som det ska.

Du behöver inte nödvändigtvis följa anvisningarna i turordning eller slutföra ett steg innan du börjar på nästa. Du kan till exempel utforma en skärm i programmet (steg 1) och sedan skapa grafik, knappar o.s.v. (steg 3) innan du skriver ActionScript-koden (steg 2) och testar (steg 4). Du kan också utforma en del av programmet och sedan lägga till en knapp eller ett gränssnittselement i taget, skriva ActionScript för en del i taget och testa dem vartefter de byggs. Det är en god idé att ha dessa fyra delar av utvecklingsprocess i åtanke. I verkligheten är det däremot mer effektivt att gå fram och tillbaka mellan de olika delarna efter behov.

Skapa egna klasser

Att skapa klasser för ett projekt kan verka skrämmande. Den svåraste delen med att skapa en klass är att utforma dess metoder, egenskaper och händelser.

Strategier för att utforma en klass

Avsnittet objektorienterad design är mycket komplext. Det finns en hel akademisk kurs och professionell träning i det här ämnet. Här föreslår vi i alla fall några alternativ som hjälper dig att komma igång.

1 Fundera på vilken roll den här klassens instanser ska ha i programmet. Vanligtvis spelar objekten en av dessa roller:

   - Värdeobjekt: De här objekten fungerar i huvudsak som databehållare. De har förmodligen flera egenskaper och färre metoder (och ibland inga metoder alls). De utgör vanligtvis kodrepresentationer av tydligt definierade objekt. Ett musikuppspelningsprogram kan till exempel innehålla klassen Song, som representerar en faktisk sång, och klassen Playlist, som representerar en begreppsmässig grupp med sånger.
   - Visningsobjekt: Dessa objekt visas på skärmen. Dessa kan vara användargränssnittselement, till exempel listrutor och statusavläsningar eller grafiska element, som djur i ett videospel.
   - Programstruktur: Dessa objekt spelar stödjande roller i den logik eller process som utförs av program. Du kan till exempel skapa ett objekt som ska utföra vissa beräkningar i en biologisk simulering. Du kan skapa ett som sköter synkroniseringen av värden mellan en kontroll och en volymavläsning i ett musikprogram. Ett annat kan hantera reglerna i ett videospel. Eller så kan du skapa en klass för att läsa in en sparad bild i ett ritprogram.

2 Bestäm dig för vilka funktioner som klassen behöver. De olika funktionerna blir ofta klassens metoder.

3 Om klassen ska fungera som ett värdeobjekt fastställer du vilka data som instanserna ska innehålla. Dessa element kan få bli egenskaper.

4 Eftersom din klass utformas speciellt för ditt projekt är det viktigt att du anger de funktioner som programmet behöver. Försök att besvara följande frågor:

   - Vilken storts information ska programmet spara, spåra och hantera? Genom att besvara den här frågan kan du lättare identifiera de värdeobjekt och egenskaper som behövs.

- Vilka åtgärder ska programmet utföra? Vad händer till exempel när programmet först läses in, när någon klickar på en viss knapp eller när en filmuppspelning tar slut? Dessa saker kan få bli metoder. De kan också vara egenskaper om ”åtgärderna” medför att enskilda värden ändras.
- Vilken information krävs för att utföra en viss åtgärd? Denna information blir metodens parametrar.
- Efterhand som programmet utför sina funktioner: Vilka saker ändras i klassen som andra delar av programmet måste känna till? Dessa saker kan få bli händelser.

5 Finns det ett befintligt objekt som liknar det objekt du behöver, men som kanske saknar någon extrafunktion som du vill lägga till? Fundera på om du ska skapa en underklass. (En *underklass* är en klass som bygger på funktionerna i en befintlig klass, i stället för att ha egna definierade funktioner.) Om du till exempel vill skapa en klass som är ett visuellt objekt på skärmen använder du beteendet hos ett befintligt visningsobjekt som utgångspunkt för klassen. I så fall blir visningsobjektet (t.ex. MovieClip eller Sprite) *basklassen*, och din klass utökar den klassen.

Skriva kod för en klass

När du väl har en design för klassen, eller åtminstone en uppfattning om vilken information som ska sparas i den och vilka åtgärder den ska utföra, är arbetet med att skriva klassen relativt enkelt.

Detta är vad som minst krävs när du vill skapa en egen ActionScript-klass:

1 Öppna ett nytt textdokument i ActionScript-textredigeraren.

2 Ange en `class`-programsats för att definiera namnet på klassen. Om du vill lägga till en `class`-programsats skriver du orden `public class` och sedan klassens namn. Lägg till inledande och avslutande klammerparenteser för att omge innehållet i klassen (metoden och egenskapsdefinitionerna). Till exempel:

```
public class MyClass
{
}
```

Ordet `public` anger att klassen kan nås från vilken kod som helst. Andra alternativ beskrivs i Namnutrymmesattribut för åtkomstkontroll.

3 Skriv en `package`-programsats för att ange namnet på det paket som innehåller klassen. Syntaxen är ordet `package`, följt av det fullständiga paketnamnet, följt av inledande och avslutande klammerparentes runt `class`-programsatsblocket. Ändra till exempel koden i föregående steg till följande:

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

4 Definiera varje egenskap i klassen med programsatsen `var` inuti klassbrödtexten. Syntaxen är densamma som när du deklarerar variabler (plus modifieraren `public`). Om du till exempel lägger till dessa rader mellan den inledande och avslutande klammerparentesen för klassdefinitionen skapas egenskaperna `textProperty`, `numericProperty` och `dateProperty`:

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

5 Definiera varje metod i klassen med samma syntax som används för att definiera en funktion. Till exempel:

- Om du vill skapa metoden `myMethod()` anger du:

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

- Om du vill skapa en konstruktor (den speciella metod som anropas som en del av processen med att skapa en förekomst av en klass) skapar du en metod vars namn ska matcha namnet på klassen:

  ```
  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);
  }
  ```

  Om du inte inkluderar någon konstruktormetod i klassen skapas en tom konstruktor i klassen automatiskt vid kompileringen. (Alltså en konstruktor utan parametrar och utan programsatser.)

Det finns några fler klasselement som du kan definiera. De här elementen är mer komplicerade.

- *Åtkomstmetoden är en korsning mellan en metod och en egenskap.* När du skriver koden för att definiera klassen skriver du åtkomstmetoden som en metod. Du kan utföra flera åtgärder, i stället för att bara läsa eller tilldela ett värde, vilket är det enda du kan göra när du definierar en egenskap. När du skapar en instans av klassen hanterar du åtkomstmetoden som en egenskap och använder namnet för att läsa eller tilldela värdet.
- Händelser i ActionScript definieras inte med hjälp av en särskild syntax. I stället definierar du händelser i klassen med hjälp av funktionerna i klassen EventDispatcher.

Fler hjälpavsnitt

Hantera händelser

Exempel: Skapa ett basprogram

Du kan använda ActionScript 3.0 i flera olika programutvecklingsmiljöer, bland annat Flash Professional, Flash Builder och alla textredigeringsprogram.

I det här exemplet går vi igenom hur du skapar och förbättrar ett enkelt ActionScript 3.0-program med hjälp av Flash Professional eller Flash Builder. Programmet du skapar visar ett enkelt mönster för hur du använder externa ActionScript 3.0-klassfiler i Flash Professional och Flex.

Utforma egna ActionScript-program

Det här exemplet på ActionScript-program är ett vanligt ”Hello World”-program med enkel design:

- Programmet heter HelloWorld.
- Det visar ett enda textfält, som innehåller orden ”Hello World!”.
- Programmet använder en objektorienterad klass, som heter Greeter. Den här utformningen tillåter att klassen används inifrån ett Flash Professional- eller Flex-projekt.
- I det här exemplet skapar du först en enkel version av programmet. Sedan lägger du till funktioner så att användaren kan ange ett användarnamn, som programmet kontrollerar mot en lista över kända användare.

När den här definitionen är på plats kan du börja skapa själva programmet.

Skapa projektet HelloWorld och klassen Greeter

I utformningsbeskrivningen för programmet Hello World anges att koden är lätt att återanvända. För att uppnå det använder programmet en enda objektorienterad klass, som heter Greeter. Du använder den klassen inifrån ett program som du skapar i Flash Builder eller Flash Professional.

Så här skapar du projektet HelloWorld och klassen Greeter i Flex:

1. Välj File > New > Flex Project i Flash Builder.
2. Skriv HelloWorld som projektnamn. Kontrollera att Application type är inställt på ”Web (runs in Adobe Flash Player)” och klicka sedan på Finish.

   Projektet skapas i Flash Builder och visas i Package Explorer. Projektet innehåller som standard redan en fil med namnet HelloWorld.mxml, och den filen öppnas i redigeraren.
3. Skapa nu en egen ActionScript-klassfil i Flash Builder genom att välja Arkiv> Ny > ActionScript-klass.
4. I dialogrutan Ny ActionScript-klass skriver du **Greeter** som klassnamn i fältet Namn. Klicka sedan på Slutför.

   Ett nytt redigeringsfönster visas.

   Fortsätt med Lägga till koder i klassen Greeter.

Så här skapar du klassen Greeter i Flash Professional:

1. Välj Arkiv > Nytt i Flash Professional.
2. I dialogrutan Nytt dokument väljer du ActionScript-fil och klickar på OK.

   Ett nytt redigeringsfönster visas.
3. Välj Arkiv > Spara. Välj en mapp som ska innehålla programmet, ge ActionScript-filen namnet **Greeter.as** och klicka på OK.

   Fortsätt med Lägga till koder i klassen Greeter.

Lägga till koder i klassen Greeter

Med klassen Greeter definieras objektet `Greeter`, som du använder i programmet HelloWorld.

Så här lägger du till kod i klassen Greeter:

1. Skriv den här koden i den nya filen (viss kod kan ha lagts till):

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

   Klassen Greeter innehåller en enda `sayHello()`-metod, som returnerar strängen ”Hello World!”.
2. Välj Arkiv > Spara och spara ActionScript-filen.

Klassen Greeter är nu färdig att användas i ett program.

Skapa ett program som använder din ActionScript-kod

Klassen Greeter som du har skapat definierar en fristående uppsättning programfunktioner, men den representerar inte ett fullständigt program. För att använda klassen skapar du ett Flash Professional-dokument eller ett Flex-projekt.

Koden måste innehålla en instans av klassen Greeter. Nedan beskrivs hur du använder klassen Greeter i programmet.

Så här skapar du ett ActionScript-program med Flash Professional:

1. Välj Arkiv > Nytt.
2. I dialogrutan Nytt dokument väljer du Flash-fil (ActionScript 3.0) och klickar på OK.

   Ett nytt dokumentfönster visas.
3. Välj Arkiv > Spara. Välj samma mapp som innehåller klassfilen Greeter.as, ge Flash-dokumentet namnet **HelloWorld.fla** och klicka på OK.
4. Välj textverktyget i verktygspaletten i Flash Professional. Dra över scenen för att definiera ett nytt textfält, ungefär 300 pixlar brett och 100 pixlar högt.
5. Gå till egenskapspanelen när textfältet fortfarande är markerat på scenen, ställ in texttypen på ”Dynamisk text” och skriv **mainText** som instansnamn för textfältet.
6. Klicka på den första bildrutan i huvudtidslinjen. Öppna panelen Åtgärder genom att välja Fönster > Åtgärder.
7. Skriv följande skript i åtgärdspanelen:

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

8. Spara filen.

Fortsätt med Publicera och testa ActionScript-programmet.

Så här skapar du ett ActionScript-program med Flash Builder:

1. Öppna filen HelloWorld.mxml och lägg till kod enligt följande:

```
<?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>
```

Det här Flex-projektet innehåller fyra MXML-taggar:

- En `<s:Application>`-tagg som definierar Application-behållaren
- En `<s:layout>`-tagg, som definierar layouten (vertikal layout) för Application-taggen
- En `<fx:Script>`-tagg som innehåller ActionScript-kod
- En `<s:TextArea>`-tagg som definierar ett fält som visar textmeddelanden för användaren

Koden i `<fx:Script>`-taggen definierar en `initApp()`-metod som anropas när programmet läses in. Metoden `initApp()` ställer in textvärdet för textområdet `mainTxt` till strängen ”Hello World!” som returneras av metoden `sayHello()` för den egna klassen Greeter, som du precis har skrivit.

2 Välj Arkiv > Spara för att spara programmet.

Fortsätt med Publicera och testa ActionScript-programmet.

Publicera och testa ActionScript-programmet

Programutveckling är en repetitiv process. Du skriver litet kod, försöker kompilera den och redigerar koden tills den kompileras utan problem. Du kör det kompilerade programmet och testar om det fungerar som det är tänkt. Om inte, redigerar du koden igen tills det fungerar. I utvecklingsmiljöerna Flash Professional och Flash Builder finns det olika sätt att publicera, testa och felsöka program.

Här anges de grundläggande stegen för att testa programmet HelloWorld i respektive miljö.

Så här publicerar du och testar ett ActionScript-program med Flash Professional:

1 Publicera programmet och se upp för kompileringsfel. Välj Kontroll > Testa filmen i Flash Professional för att kompilera ActionScript-koden och köra programmet.

2 Om det visas några fel eller varningar i utdatafönstret när du testar programmet korrigerar du felen i HelloWorld.fla- eller HelloWorld.as-filerna. Sedan testar du programmet igen.

3 Om det nu inte blir några kompileringsfel visas ett Flash Player-fönster med programmet Hello World.

Du har skapat ett enkelt men fullständigt objektorienterat program som använder ActionScript 3.0. Fortsätt med Förbättra programmet HelloWorld.

Så här publicerar du och testar ett ActionScript-program med Flash Builder:

1 Välj Kör > Kör HelloWorld.

2 Programmet HelloWorld startar.

- Om det visas några fel eller varningar i utdatafönstret när du testar programmet korrigerar du felen i HelloWorld.mxml- eller Greeter.as-filerna. Sedan testar du programmet igen.
- Om det inte finns några kompileringsfel öppnas ett webbläsarfönster där programmet Hello World visas. Texten ”Hello World!” visas.

Du har skapat ett enkelt men fullständigt objektorienterat program som använder ActionScript 3.0. Fortsätt med Förbättra programmet HelloWorld.

Förbättra programmet HelloWorld

Om du vill göra programmet lite mer intressant kan du göra så att det frågar efter och validerar ett användarnamn mot en fördefinierad lista med namn.

Först uppdaterar du klassen Greeter och lägger till nya funktioner. Därefter uppdaterar du programmet så att det använder de nya funktionerna.

Så här uppdaterar du filen Greeter.as:

1 Öppna filen Greeter.as.

2 Ändra så att innehållet i filen blir följande (nya och ändrade rader visas med fetstil):

```
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;
            }
        }
    }
}
```

Klassen Greeter har nu fått ett antal nya funktioner:

- I arrayen `validNames` visas giltiga användarnamn. Arrayen initieras till en lista med tre namn när klassen Greeter läses in.
- Metoden `sayHello()` godkänner nu ett användarnamn och ändrar hälsningen baserat på vissa villkor. Om `userName` är en tom sträng (`""`) ställs egenskapen `greeting` in på att uppmana användaren att ange ett namn. Om användarnamnet är giltigt blir hälsningen `"Hello, `*userName*`"`. Om något av villkoren inte möts ställs variabeln `greeting` in på `"Sorry `*userName*`, you are not on the list."`
- Metoden `validName()` returnerar `true` om `inputName` finns i arrayen `validNames` och `false` om det inte finns. Programsatsen `validNames.indexOf(inputName)` kontrollerar alla strängar i arrayen `validNames` mot strängen `inputName`. Metoden `Array.indexOf()` returnerar indexpositionen för den första instansen av ett objekt i en array. Om objektet inte hittas i arrayen returneras värdet -1.

Därefter redigerar du den programfil som refererar till den här ActionScript-klassen.

Så här ändrar du programmet med Flash Professional:

1. Öppna filen HelloWorld.fla.
2. Ändra skriptet i bildruta 1 så att en tom sträng (`""`) skickas till metoden `sayHello()` för klassen Greeter:

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

3. Välj textverktyget i verktygspaletten. Skapa två nya textfält på scenen. Placera dem bredvid varandra direkt under det befintliga textfältet `mainText`.
4. I det första nya textfältet, som är etiketten, skriver du texten **User Name:**.
5. Markera det andra nya textfältet och välj Inmatningstext som textfältstyp i egenskapsinspektören. Välj Enkelrad som radtyp. Skriv **textIn** som förekomstnamn.
6. Klicka på den första bildrutan i huvudtidslinjen.

7 I åtgärdspanelen lägger du till följande rader i slutet av det befintliga skriptet:

```
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);
    }
}
```

Med den nya koden infogas följande funktion:

- De första två raderna definierar kanterna för de två textfälten.
- Ett inmatningstextfält, som fältet `textIn`, har en serie händelser som det kan skicka. Med metoden `addEventListener()` kan du definiera en funktion som körs när en typ av händelse inträffar. I det här fallet är denna händelse att någon trycker på en tangent på tangentbordet.
- Den egna funktionen `keyPressed()` kontrollerar om tangenten som tryckte var Retur-tangenten. Om så är fallet anropas metoden `sayHello()` för objektet `myGreeter` och texten skickas från textfältet `textIn` som en parameter. Den här metoden returnerar en stränghälsning baserat på det värde som skickats. Den returnerade strängen kopplas därefter till egenskapen `text` för textfältet `mainText`.

Det fullständiga skriptet för bildruta 1 blir:

```
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 Spara filen.

9 Välj Kontroll > Testa filmen för att köra programmet.

När programmet körs uppmanas du att ange ett användarnamn. Om det är giltigt (Sammy, Frank eller Dean) visas meddelandet ”hello”.

Så här ändrar du programmet med Flash Builder:

1 Öppna filen HelloWorld.mxml.

2 Sedan ändrar du taggen `<mx:TextArea>` för att tala om för användaren att den bara är för visning. Ändra bakgrundsfärgen till en ljusgrå nyans och ställ in attributet `editable` på `false`:

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

3 Lägg nu till följande rader efter den avslutande `<s:TextArea>`-taggen. Dessa rader skapar en TextInput-komponent, där användaren kan ange ett användarnamn:

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

Attributet `enter` definierar vad som händer när användaren trycker på returtangenten i fältet `userNameTxt`. I det här exemplet skickar koden texten i fältet till metoden `Greeter.sayHello()`. Hälsningen i fältet `mainTxt` ändras på motsvarande sätt.

Filen HelloWorld.mxml ser ut så här:

```
<?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 Spara den redigerade HelloWorld.mxml-filen. Välj Kör > Kör HelloWorld för att köra programmet.

När programmet körs uppmanas du att ange ett användarnamn. Om det är giltigt (Sammy, Frank eller Dean) visas bekräftelsemeddelandet ”`Hello, ` *userName*”.

Kapitel 3: ActionScript-språk och -syntax

ActionScript 3.0 innehåller både huvudspråket ActionScript och API:t för Adobe Flash Platform. Kärnspråket är den del av ActionScript som definierar syntaxen för språket samt datatyperna på den högsta nivån. Via ActionScript 3.0 får du programmeringstillgång till körningsmiljöerna för Adobe Flash Platform: Adobe Flash Player och Adobe AIR.

Språköversikt

Objekten är kärnan i ActionScript 3.0-språket och de är grundstenarna i språket. Varje variabel som du deklarerar, varje funktion du skriver och varje klassinstans du skapar är ett objekt. Du kan tänka på ett ActionScript 3.0-program som en grupp objekt som utför åtgärder, svarar på händelser och kommunicerar med varandra.

Programmerare som är duktiga på objektorienterad programmering (OOP) i Java eller C++ tänker kanske på objekt som moduler som innehåller två sorters medlemmar: data som lagrats i medlemsvariabler eller egenskaper, och beteende som kan nås via metoder. ActionScript 3.0 definierar objekt på nästan samma sätt. I ActionScript 3.0 är objekt helt enkelt samlingar av egenskaper. Dessa egenskaper är behållare som kan innehålla inte bara data utan också funktioner eller andra objekt. Om en funktion bifogas till ett objekt på det här sättet, kallas den en metod.

ActionScript 3.0-definitionen kan verka lite egendomlig för programmerare som tidigare arbetat med Java eller C++, men i själva verket definierar man objekttyper med ActionScript 3.0-klasser på ungefär samma sätt som man definierar klasser i Java eller i C++. Skillnaden mellan de två definitionerna av objekt är viktig när man beskriver ActionScript-objektmodellen och andra avancerade ämnen, men i de flesta fall betyder termen *egenskaper* klassmedlemsvariabler i motsats till metoder. I Referenshandbok för ActionScript 3.0 i Adobe Flash-plattformen används till exempel termen *egenskaper* om variabler eller get-/set-egenskaper. Här används termen *metoder* för att ange funktioner som är en del av en klass.

En liten skillnad mellan klasser i ActionScript och klasser i Java eller C++ är att i ActionScript är klasser inte bara abstrakta enheter. ActionScript-klasser representeras av *klassobjekt* som lagrar klassens egenskaper och metoder. Detta gör att man kan använda teknik som är helt främmande för Java- och C++-programmerare, till exempel att ta med programsatser eller körbar kod på den översta nivån i en klass eller ett paket.

En annan skillnad mellan klasserna ActionScript och Java eller C++ är att varje ActionScript-klass har något som kallas *prototyp-objekt*. I föregående versioner av ActionScript fungerade prototypobjekten, som kopplats ihop till *prototypkedjor*, kollektivt som grunden för hela klassarvshierarkin. I ActionScript 3.0 spelar prototypobjekten bara en liten roll i arvssystemet. Prototypobjektet kan fortfarande vara användbart som ett alternativ till statiska egenskaper och metoder om du vill dela ut en egenskap och dess värde till alla instanser av en klass.

Tidigare kunde avancerade ActionScript-programmerare direkt ändra prototypkedjan med speciella inbyggda språkelement. Nu när språket tillhandahåller en mera utvecklad implementering av ett klassbaserat programmeringsgränssnitt ingår många av dessa speciella språkelement, som `__proto__` och `__resolve`, inte längre i språket. Optimeringar av den interna arvsmekanismen, som ger betydande prestandaförbättringar, förhindrar dessutom direktåtkomst till arvsmekanismen.

Objekt och klasser

I ActionScript 3.0 definieras alla objekt av en klass. En klass kan betraktas som en mall eller en plan för en typ av objekt. Klassdefinitioner kan innehålla variabler och konstanter som innehåller datavärden, samt metoder, som är funktioner som sammanfattar beteenden som kopplas till klassen. De värden som lagras i egenskaper kan vara *primitiva värden* eller andra objekt. Primitiva värden är nummer, strängar eller booleska värden.

ActionScript innehåller ett antal inbyggda klasser som är en del av huvudspråket. Vissa av dessa inbyggda klasser, till exempel Number, Boolean och String, representerar primitiva värden som finns i ActionScript. Andra, som klasserna Array, Math och XML, definierar mer komplexa objekt.

Alla klasser, vare sig de är inbyggda eller användardefinierade, härstammar från klassen Object. Programmerare med tidigare erfarenhet av ActionScript bör känna till att datatypen Object inte längre är standardtyp, även om alla andra klasser fortfarande härstammar från den. I ActionScript 2.0 var följande två kodrader likvärdiga eftersom saknaden av typanteckning betydde att en variabel kunde vara av typen Object:

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

ActionScript 3.0 introducerar emellertid begreppet typlösa variabler, som kan anges på följande två sätt:

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

En typlös variabel är inte detsamma som en variabel av typen Object. Huvudskillnaden är att typlösa variabler kan innehålla det speciella värdet `undefined`, medan en variabel av typen Objekt inte kan innehålla detta värde.

Du kan definiera egna klasser med hjälp av nyckelordet `class`. Du kan deklarera klassegenskaper på tre olika sätt: konstanter kan definieras med nyckelordet `const`, variabler definieras med nyckelordet `var` och get- och set-egenskaper definieras med attributen `get` och `set` i en metoddeklaration. Du kan deklarera metoder med nyckelordet `function`.

Du kan skapa en instans av en klass med operatorn `new`. I följande exempel skapas en instans av klassen Date som får namnet `myBirthday`.

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

Paket och namnutrymmen

Paket och namnutrymmen är relaterade begrepp. Med paket kan du samla ihop klassdefinitioner på ett sätt som underlättar koddelning och som minimerar namnkonflikter. Med namnutrymmen kan du styra visningen av identifierare, till exempel egenskaps- och metodnamn, och de kan användas i kod vare sig de finns inuti eller utanför ett paket. Med paket kan du organisera klassfiler och med namnutrymmen kan du hantera visningen av individuella egenskaper och metoder.

Paket

Paket i ActionScript 3.0 implementeras med namnutrymmen, men är inte synonymer med dem. När du deklarerar ett paket skapar du en speciell typ av namnutrymme som garanterat identifieras vid kompileringen. När namnutrymmen skapas explicit identifieras de inte nödvändigtvis vid kompileringen.

I följande exempel används direktivet `package` för att skapa ett enkelt paket som innehåller en klass:

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

Namnet på klassen i det här exemplet är SampleCode. Eftersom klassen finns inuti exempelpaketet kvalificerar kompilatorn automatiskt klassnamnet vid kompileringen till det fullständiga, kvalificerade namnet samples.SampleCode. Kompilatorn kvalificerar också namnet på egenskaper eller metoder så att `sampleGreeting` och `sampleFunction()` blir `samples.SampleCode.sampleGreeting` respektive `samples.SampleCode.sampleFunction()`.

Många utvecklare, särskilt de som har använt Java, väljer att placera bara klasser på den översta paketnivån. I ActionScript 3.0 kan du emellertid använda variabler, funktioner och även programsatser på den översta paketnivån och inte bara klasser. Ett avancerat sätt att använda den här funktionen är att definiera ett namnutrymme på den översta paketnivån, så att det blir tillgängligt för alla klasser i det paketet. Observera att bara två åtkomstspecifikationer, `public` och `internal`, tillåts på den översta paketnivån. I motsats till Java, där du kan deklarera kapslade klasser som privata, stöder ActionScript 3.0 varken kapslade eller privata klasser.

ActionScript 3.0-paket liknar på många sätt paket i programmeringsspråket Java. Som du ser i föregående exempel uttrycks fullständiga, kvalificerade paketreferenser med hjälp av punktoperatorn (.), på samma sätt som i Java. Du kan använda paket för att organisera din kod i en intuitiv hierarkisk struktur som kan användas av andra programmerare. Detta underlättar koddelning eftersom du kan skapa ett eget paket som du delar med andra, och du kan använda paket som skapats av andra i din kod.

När du använder paket blir de identifierarnamn du använder unika och hamnar inte i konflikt med andra identifierare. Somliga tycker att det här är den viktigaste fördelen med paket. Två programmerare som vill dela kod med varandra kan till exempel båda ha skapat en klass som heter SampleCode. Utan paket skulle detta skapa en namnkonflikt och den enda lösningen skulle vara att namnändra en av klasserna. Med paket undviks namnkonflikten genom att man placerar en, eller företrädesvis båda, klasserna i paket med unika namn.

Du kan också infoga inbäddade punkter i paketnamnet för att skapa kapslade paket. Då kan du skapa en hierarkisk paketorganisation. Ett bra exempel på detta är flash.display-paketet som ingår i ActionScript 3.0. Detta paket är kapslat inuti flash-paketet.

Det mesta i ActionScript 3.0 struktureras under flash-paketet. Paketet flash-display innehåller till exempel visningslist-API:n och paketet flash.events innehåller den nya händelsemodellen.

Skapa paket

ActionScript 3.0 är mycket flexibelt eftersom du kan organisera dina paket, klasser och källfiler. I tidigare versioner av ActionScript kunde du bara använda en klass per källfil och dessutom krävdes att namnet på källfilen matchade namnet på klassen. Med ActionScript 3.0 kan du ha flera klasser i samma källfil, men bara en klass i varje fil kan göras åtkomligt för kod som är extern för filen. Endast en klass i varje fil kan m.a.o. deklareras inuti en paketdeklaration. Du måste deklarera eventuella ytterligare klasser utanför paketdefinitionen, vilket gör att dessa klasser inte syns för kod som är utanför källfilen. Namnet på den klass som deklarerats inuti paketdefinitionen måste matcha namnet på källfilen.

ActionScript 3.0 är också mera flexibelt när det gäller att deklarera paket. I tidigare versioner av ActionScript representerade paket snarare kataloger i vilka du placerade källfiler, och du deklarerade inte paket med programsatsen `package` utan infogade paketnamnet som en del av det fullständiga, kvalificerade klassnamnet i klassdeklarationen. Även om paket fortfarande representerar kataloger i ActionScript 3.0 kan paket innehålla mer än just klasser. I ActionScript 3.0 använder du programsatsen `package` för att deklarera ett paket, vilket betyder att du också kan deklarera variabler, funktioner och namnutrymmen på den översta paketnivån. Du kan till och med infoga körbara programsatser på den översta paketnivån. Om du deklarerar variabler, funktioner eller namnutrymmen på den översta paketnivån är attributen `public` och `internal` de enda tillgängliga på den nivån, och attributet `public` kan bara användas av en enda paketnivådeklaration per fil, vare sig deklarationen är klass, variabel, funktion eller namnutrymme.

Du kan använda paket för att organisera kod och för att förhindra namnkonflikter. Du får inte blanda ihop begreppet paket med det orelaterade begreppet klassarv. Två klasser som finns i samma paket har ett gemensamt namnutrymme, men behöver inte vara relaterade till varandra på något annat sätt. På samma sätt har ett kapslat paket inte någon semantisk relation till sitt överordnade paket.

Importera paket

Om du vill använda en klass som finns inuti ett paket, måste du importera antingen paketet eller klassen. I ActionScript 2.0 behöver du inte importera klasser.

Vi kan använda klassen SampleCode som nämndes tidigare som exempel. Om klassen finns i ett paket som heter samples måste du använda någon av följande importprogramsatser innan du kan använda klassen SampleCode:

```
import samples.*;
```

eller

```
import samples.SampleCode;
```

Vanligtvis måste `import`-programsatser vara så specifika som möjligt. Om du bara vill använda klassen SampleCode från exempelpaketet, får du bara importera klassen SampleCode och inte hela paketet som klassen hör till. Om du importerar hela paket kan namnkonflikter uppstå.

Du måste också placera källkoden som definierar paketet eller klassen inuti din *klassökväg*. Klassökvägen är en användardefinierad lista med lokala katalogsökvägar, som anger var kompilatorn ska söka efter importerade paket och klasser. Klassökvägen kallas ibland *byggsökväg* eller *källsökväg*.

När du har importerat klassen eller paketet kan du använda antingen det fullständiga, kvalificerade namnet på klassen (samples.SampleCode) eller bara själva klassnamnet (SampleCode).

Du kan använda fullständiga, kvalificerade namn när klasser, metoder eller egenskaper med samma namn resulterar i tvetydig kod, men de kan vara svåra att hantera om de används för alla identifierare. Om du till exempel använder fullständiga, kvalificerade namn blir koden alltför detaljerad när du initierar klassinstansen SampleCode:

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

Vartefter nivån av kapslade paket ökar, minskar kodens läsbarhet. Om du är säker på att tvetydiga identifierare inte är något problem kan du använda enkla identifierare så att koden blir lättare att läsa. När du instansierar en ny instans av klassen SampleCode blir koden inte så detaljerad om du bara använder klassidentifieraren:

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

Om du försöker använda identifierarnamn utan att först importera lämpligt paket eller lämplig klass kan klassdefinitionerna inte hittas av kompilatorn. Om du å andra sidan importerar ett paket eller en klass genereras ett fel om du försöker definiera ett namn som är i konflikt med ett importerat namn.

När du har skapat ett paket blir standardåtkomstspecifikationen för alla medlemmar i detta paket `internal`, vilket betyder att paketmedlemmarna bara visas för andra medlemmar i detta paket. Om du vill att en klass ska vara tillgänglig för kod utanför paketet, måste du deklarera klassen som `public`. Följande paket innehåller till exempel två klasser, SampleCode och CodeFormatter:

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

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

Klassen SampleCode visas utanför paketet eftersom det har deklarerats som en `public`-klass. Klassen CodeFormatter visas bara inuti själva exempelpaketet. Om du försöker öppna klassen CodeFormatter utanför exempelpaketet genereras ett felmeddelande, vilket visas i följande exempel:

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

Om du vill att båda klasserna ska vara tillgängliga utanför paketet måste du deklarera båda klasserna som `public`. Du kan inte använda attributet `public` på paketdeklarationen.

Du kan använda fullständiga, kvalificerade namn för att lösa de namnkonflikter som inträffar när du använder paket. Ett sådant scenario kan inträffa om du importerar två paket som definierar klasser med samma identifierare. Tänk dig följande paket som också har en klass som heter SampleCode:

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

Om du importerar båda klasserna uppstår en namnkonflikt när du använder klassen SampleCode:

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

Kompilatorn vet inte vilken SampleCode-klass som ska användas. För att lösa den här konflikten måste du använda det fullständiga, kvalificerade namnet på båda klasserna:

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

Obs! *Programmerare som använt C++ använder ofta programsatsen `import` med `#include`. Direktivet `#include` är obligatoriskt i C++ eftersom C++-kompilatorn bearbetar en fil i taget och bara letar efter klassdefinitioner i andra filer om en rubrikfil ingår. ActionScript 3.0 har ett `include`-direktiv, men det importerar inte klasser och paket. Om du vill importera klasser och paket i ActionScript 3.0 måste du använda programsatsen `import` och placera källfilen som innehåller paketet i klassökvägen.*

Namnutrymmen

Med namnutrymmen får du kontroll över hur egenskaper och metoder du skapar visas. Tänk på åtkomstkontrollsspecifikationerna `public`, `private`, `protected` och `internal` som inbyggda namnutrymmen. Om dessa fördefinierade specifikationer inte passar kan du skapa egna namnutrymmen.

Om du känner till XML-namnutrymmen är mycket av den här beskrivningen inte ny för dig, även om syntax och detaljer i ActionScript-implementeringen är litet annorlunda än i XML. Även om du inte har arbetat med namnutrymmen tidigare förstår du ändå själva begreppet, men implementeringen har en speciell terminologi som du måste lära dig.

Om du vill förstå hur namnutrymmen fungerar kan det vara bra att veta att namnet på en egenskap eller en metod består av två delar: en identifierare och ett namnutrymme. Identifieraren är det som du vanligtvis tänker på som ett namn. Identifieraren i följande klassdefinition är `sampleGreeting` och `sampleFunction()`:

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

När definitioner föregås av ett namnutrymmesattribut är deras namn kvalificerat av standardnamnutrymmet `internal`, vilket betyder att de bara visas för anropare i samma paket. Om kompilatorn är inställd på strikt läge utfärdas en varning om att namnutrymmet `internal` tillämpas på alla identifierare som saknar ett namnutrymmesattribut. Om du vill vara säker på att en identifierare ska bli tillgänglig överallt, måste du lägga in attributet `public` före identifierarnamnet. I föregående exempelkod har både `sampleGreeting` och `sampleFunction()` namnutrymmesvärdet `internal`.

Du följer tre grundprinciper när du använder namnutrymmen. För det första måste du definiera namnutrymmet med nyckelordet `namespace`. I följande kod definieras till exempel namnutrymmet `version1`:

```
namespace version1;
```

För det andra använder du namnutrymmet i stället för en åtkomstkontrollsspecifikation i en egenskaps- eller metoddeklaration. I följande exempel placeras funktionen `myFunction()` i namnutrymmet `version1`:

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

För det tredje kan du, när du har tillämpat namnutrymmet, referera det med direktivet `use` eller genom att kvalificera namnet på en identifierare med ett namnutrymme. I följande exempel refereras funktionen `myFunction()` via direktivet `use`:

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

Du kan också använda ett kvalificerat namn för att referera funktionen `myFunction()`, vilket visas i följande exempel:

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

Definiera namnutrymmen

Namnutrymmen innehåller ett värde, URI (Uniform Resource Identifier), som ibland kallas *namnutrymmesnamn*. Med en URI kan du kontrollera att namnutrymmesdefinitionen är unik.

Du skapar ett namnutrymme genom att deklarera en namnutrymmesdefinition på ett av två sätt. Du kan antingen definiera namnutrymmet med en angiven URI, på samma sätt som du definierar ett XML-namnutrymme, eller också kan du utelämna detta URI. I följande exempel visas hur du definierar ett namnutrymme med en URI:

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

URI:n fungerar som en unik identifikationssträng för namnutrymmet. Om du, som i följande exempel, utelämnar URI:n skapar kompilatorn en unik intern identifikationssträng i stället för URI:n. Du har inte åtkomst till den här interna identifikationsfilen.

```
namespace flash_proxy;
```

När du har definierat ett namnutrymme, med eller utan en URI, kan namnutrymmet inte definieras om i samma omfång. Om du försöker definiera ett namnutrymme som tidigare har definierats i samma omfång genereras ett kompilatorfel.

Om ett namnutrymme definieras inuti ett paket eller en klass, visas det bara för kod utanför paketet eller klassen om lämplig åtkomstkontrollsspecifikation används. Följande kod visar namnutrymmet `flash_proxy` som definierats inuti paketet flash.utils. I följande exempel betyder avsaknaden av en åtkomstkontrollsspecifikation att namnutrymmet `flash_proxy` bara visas för kod inuti paketet flash.utils och inte för kod utanför paketet:

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

I följande kod används attributet `public` för att visa namnutrymmet `flash_proxy` för kod utanför paketet:

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

Använda namnutrymmen

Att använda ett namnutrymme innebär att placera en definition i ett namnutrymme. Definitioner, som funktioner, variabler och konstanter, kan placeras i namnutrymmen (du kan inte placera en klass i ett anpassat namnutrymme).

En funktion kan till exempel deklareras med hjälp av åtkomstkontrollsspecifikationen `public`. Om du använder attributet `public` i en funktionsdefinition placeras funktionen i det allmänna namnutrymmet, vilket gör att funktionen blir tillgänglig för all kod. När du har definierat ett namnutrymme kan du använda namnutrymmet på samma sätt som du använder attributet `public`, och definitionen blir tillgänglig för kod som kan referera till ditt anpassade namnutrymme. Om du till exempel definierar namnutrymmet `example1` kan du lägga till metoden `myFunction()` och använda `example1` som ett attribut, vilket visas i följande exempel:

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

Att deklarera metoden `myFunction()` och använda namnutrymmet `example1` som ett attribut betyder att metoden hör till namnutrymmet `example1`.

Du måste tänka på följande när du använder namnutrymmen:

- Du kan bara använda ett namnutrymme för varje deklaration.
- Du kan inte använda ett namnutrymmesattribut för mer än en definition i taget. Om du vill använda namnutrymmet på tio olika funktioner måste du alltså lägga till namnutrymmet som ett attribut i alla tio funktionsdefinitionerna.

- Om du använder ett namnutrymme kan du inte ange en åtkomstkontrollsspecifikation samtidigt, eftersom namnutrymmen och åtkomstkontrollsspecifikationer inte kan kombineras. Du kan alltså inte deklarera en funktion eller en egenskap som `public`, `private`, `protected` eller `internal` när du samtidigt använder namnutrymmet.

Referera namnutrymmen

Du behöver inte explicit referera ett namnutrymme när du använder en metod eller en egenskap som deklarerats med någon av åtkomstkontrollsnamnutrymmena, till exempel `public`, `private`, `protected` och `internal`. Detta beror på att åtkomsten till dessa speciella namnutrymmen styrs av sammanhanget. Definitioner som placerats i namnutrymmet `private` blir till exempel automatiskt tillgängliga för kod i samma klass. Det finns däremot ingen sådan sammanhangskänslighet för de namnutrymmen du definierar. Om du vill använda en metod eller egenskap som du har placerat i ett anpassat namnutrymme, måste du referera till namnutrymmet.

Du kan referera namnutrymmen med direktivet `use namespace` och du kan kvalificera namnet med namnutrymmet med hjälp av namnkvalificeringstecknet (`::`). Om du refererar ett namnutrymme med direktivet `use namespace` ”öppnas” namnutrymmet, så att det kan användas på alla identifierare som inte är kvalificerade. Om du har definierat namnutrymmet `example1` kan du öppna namnet i namnutrymmet genom att använda `use namespace example1`:

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

Du kan öppna flera namnutrymmen samtidigt. När du har öppnat ett namnutrymme med `use namespace` förblir det öppet i hela kodblocket som det öppnats i. Det finns inget sätt att explicit stänga ett namnutrymme.

Om du har flera öppna namnutrymmen ökar emellertid risken för namnkonflikter. Om du föredrar att inte öppna ett namnutrymme, kan du undvika direktivet `use namespace` genom att kvalificera metod- eller egenskapsnamnet med namnutrymmet och namnkvalificeringstecknet. I följande kod visas hur du kan kvalificera namnet `myFunction()` med namnutrymmet `example1`:

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

Använda namnutrymmen

Du hittar ett verkligt exempel på ett namnutrymme som används för att förhindra namnkonflikter i klassen flash.utils.Proxy som ingår i ActionScript 3.0. Klassen Proxy ersätter egenskapen `Object.__resolve` från ActionScript 2.0, gör att du kan avbryta referenser till odefinierade egenskaper eller metoder innan ett fel inträffar. Alla metoder för klassen Proxy finns i namnutrymmet `flash_proxy` för att förhindra namnkonflikter.

För att du bättre ska kunna förstå hur namnutrymmet `flash_proxy` används, måste du förstå hur klassen Proxy ska användas. Klassen Proxys funktioner är bara tillgängliga för klasser som ärver från denna klass. Om du vill använda metoderna i klassen Proxy på ett objekt, måste objektets klassdefinition utöka klassen Proxy. Om du till exempel vill avbryta försök till anrop av en odefinierad metod, utökar du klassen Proxy och sedan åsidosätter du metoden `callProperty()` i klassen Proxy.

Du kanske kommer ihåg att implementering av namnutrymmen vanligtvis är en trestegsprocess som består av att definiera, använda och referera ett namnutrymme. Eftersom du aldrig explicit anropar någon av klassen Proxys metoder, kommer namnutrymmet `flash_proxy` att definieras och användas, men inte att refereras. ActionScript 3.0 definierar namnutrymmet `flash_proxy` och använder det i klassen Proxy. Din kod behöver bara använda namnutrymmet `flash_proxy` på klasser som utökar klassen Proxy.

Namnutrymmet `flash_proxy` definieras i paketet flash.utils på ett sätt som liknar detta:

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

Namnutrymmet används på metoderna i klassen Proxy enligt följande utdrag ur klassen Proxy:

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

Som följande kod visar måste du först importera både klassen Proxy och namnutrymmet `flash_proxy`. Du måste deklarera klassen så att den utökar klassen Proxy (du måste alltså lägga till attributet `dynamic` om du kompilerar i strikt läge). När du åsidosätter metoden `callProperty()` måste du använda namnutrymmet `flash_property`.

```
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);
        }
    }
}
```

Om du skapar en instans av klassen MyProxy och anropar en odefinierad metod, till exempel metoden `testing()`, som anropas i följande exempel, kommer objektet Proxy att avbryta metodanropet och köra programsatserna inuti den åsidosatta metoden `callProperty()` (i det här fallet en enkel `trace()`-programsats).

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

Det finns två fördelar med att ha metoder av klassen Proxy inuti namnutrymmet `flash_proxy`. Om du för det första har ett separat namnutrymme minskas skräpet i det gemensamma gränssnittet för en klass som utökar klassen Proxy. (Det finns ungefär ett dussin metoder i klassen Proxy som du kan åsidosätta, och ingen av dem kan anropas direkt. Om du placerar dem i det allmänna namnutrymmet blir det konstigt.) Om du för det andra använder namnutrymmet `flash_proxy` undviker du namnkonflikter om underklassen till Proxy innehåller instansmetoder med samma namn som någon av metoderna i klassen Proxy. Du kan till exempel vilja kalla en av dina metoder `callProperty()`. Följande kod godkänns eftersom din version av metoden `callProperty()` finns i ett annat namnutrymme:

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

Det kan också vara bra att använda namnutrymmen när du vill ha åtkomst till metoder och egenskaper på ett sätt som inte kan uppnås med de fyra åtkomstkontrollsspecifikationerna (`public`, `private`, `internal` och `protected`). Det finns till exempel några verktygsfunktioner som är spridda över flera paket. Du vill att alla dessa metoder ska vara tillgängliga för alla dina paket, men du vill inte att metoderna ska vara allmänna. För att uppnå detta kan du skapa ett namnutrymme och använda det som en egen speciell åtkomstkontrollsspecifikation.

I följande exempel används ett användardefinierat namnutrymme för att gruppera två funktioner som finns i olika paket. Genom att gruppera dem i samma namnutrymme, kan du göra båda funktionerna synliga för en klass eller ett paket via en enda `use namespace`-programsats.

I det här exemplet används fyra filer för att visa den här tekniken. Alla filerna måste ligga i din klassökväg. Den första filen, myInternal.as, används för att definiera namnutrymmet `myInternal`. Eftersom filen finns i ett paket som heter example, måste du placera filen i en mapp som heter example. Namnutrymmet markeras som `public` så att det kan importeras till andra paket.

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

Den andra och tredje filen, Utility.as och Helper.as, definierar de klasser som innehåller metoder som ska vara tillgängliga för andra paket. Klassen Utility finns i paketet example.alpha, vilket betyder att filen ska placeras i en mapp som heter alpha som är en undermapp till mappen example. Klassen Helper finns i paketet example.beta, vilket betyder att filen ska placeras i en mapp som heter beta som också är en undermapp till mappen example. Båda dessa paket, example.alpha och example.beta, måste importera namnutrymmet innan de kan använda det.

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

Den fjärde filen, NamespaceUseCase.as, är huvudprogramklassen och ska ligga på samma nivå som mappen example. I Flash Professional används den här klassen som dokumentklass för FLA. Klassen NamespaceUseCase importerar också namnutrymmet `myInternal` och använder det för att anropa de två statiska metoderna som finns i andra paket. I exemplet används statiska metoder bara för att förenkla koden. Både statiska metoder och instansmetoder kan placeras i namnutrymmet `myInternal`.

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

Variabler

Med variabler kan du lagra värden som du använder i programmet. Om du vill deklarera en variabel måste du använda programsatsen `var` med variabelnamnet. I ActionScript 3.0, krävs alltid programsatsen `var`. I följande ActionScript-rad deklareras en variabel som heter `i`:

```
var i;
```

Om du utesluter programsatsen `var` när du deklarerar en variabel visas ett kompileringsfel i strikt läge och ett körningsfel i standardläge. Följande kodrad resulterar i ett fel om variabeln `i` inte har definierats tidigare:

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

Om du vill koppla en variabel till en datatyp måste du göra det när du deklarerar variabeln. Du kan deklarera en variabel utan att ange variabelns typ, men det genererar en kompileringsvarning i strikt läge. Du anger en variabels typ genom att avsluta variabelnamnet med ett kolon (:) följt av variabelns typ. I följande kod deklareras variabeln `i` som är av typen int:

```
var i:int;
```

Du kan koppla ett värde till en variabel med tilldelningsoperatorn (`=`). I följande kod deklareras variabeln `i` och värdet 20 tilldelas variabeln:

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

Du kanske tycker att det är enklare att tilldela en variabel ett värde samtidigt som du deklarerar variabeln, som i följande exempel:

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

Tekniken med att tilldela en variabel ett värde samtidigt som variabeln deklareras är vanlig inte bara när man tilldelar primitiva värden som heltal och strängar, utan också när man skapar en array eller instansierar en instans av en klass. I följande exempel visas en array som deklarerats och tilldelats ett värde med en kodrad.

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

Du kan skapa en instans av en klass med operatorn `new`. I följande exempel skapas en instans av en namngiven `CustomClass` och en referens till den nyligen skapade klassinstansen kopplas till variabeln `customItem`:

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

Om du har flera variabler som ska deklareras kan du deklarera alla på samma kodrad med hjälp av kommaoperatorn (`,`) som avgränsar variablerna. I följande kod deklareras tre variabler på samma kodrad:

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

Du kan också tilldela varje variabel värden på samma kodrad. I följande kod deklareras tre variabler (`a`, `b` och `c`) och varje variabel tilldelas ett värde:

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

Du kan använda kommaoperatorn för att gruppera variabeldeklarationer i samma programsats, men detta minskar kodens läsbarhet.

Variabelomfång

En variabels *omfång* är det område i koden där variabeln kan nås av en lexikalreferens. En *global* variabel har definierats i alla kodens områden, medan en *lokal* variabel bara har definierats i en viss del av koden. I ActionScript 3.0 tilldelas variabler alltid det omfång av den funktion eller den klass som de deklarerats i. En global variabel definierar du utanför en funktions- eller klassdefinition. I följande kod skapas till exempel den globala variabeln `strGlobal` eftersom den deklareras utanför en funktion. I exemplet visas att en global variabel är tillgänglig både inuti och utanför funktionsdefinitionen.

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

Du deklarerar en lokal variabel inuti en funktionsdefinition. Det minsta område i en kod för vilket du kan definiera en lokal variabel är en funktionsdefinition. En lokal variabel som deklareras inuti en funktion finns bara i den funktionen. Om du exempelvis deklarerar en variabel med namnet `str2` i en funktion som heter `localScope()` är den variabeln inte tillgänglig utanför funktionen.

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

Om det variabelnamn du använder för den lokala variabeln redan har deklarerats som en global variabel, är det den lokala definitionen som döljer (eller skuggar) den globala definitionen när den lokala variabeln finns i omfånget. Utanför funktionen kommer den globala variabeln fortfarande att gälla. I följande kod skapas en global variabel med namnet `str1` och därefter kommer en lokal variabel med samma namn att skapas i funktionen `scopeTest()`. Programsatsen `trace` inuti funktionen matar ut variabelns lokala värde, och programsatsen `trace` utanför funktionen matar ut variabelns globala värde.

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

ActionScript-variabler har inte blocknivåomfång, vilket variabler i C++ och Java har. Ett kodblock är en grupp programsatser mellan en inledande klammerparentes ({) och en avslutande klammerparentes (}). I vissa programmeringsspråk, till exempel C++ och Java, är variabler som deklarerats inuti ett kodblock inte tillgängliga utanför kodblocket. Denna begränsning av omfånget kallas blocknivåomfång och finns inte i ActionScript. Om du deklarerar en variabel inuti ett kodblock är denna variabel tillgänglig inte bara i det kodblocket, utan även i andra delar av funktionen som kodblocket hör till. Följande funktion innehåller variabler som definierats i olika blockomfång. Alla variablerna är tillgängliga i hela funktionen.

```
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"]);
```

En intressant konsekvens av att det inte finns något blocknivåomfång är att du kan läsa eller skriva till en variabel innan den är deklarerad, förutsatt att den deklareras innan funktionen avslutas. Detta tack vare en teknik som kallas *hoisting*, som innebär att kompilatorn flyttar alla variabeldeklarationer till funktionens översta nivå. Följande kod kompileras till exempel även om den initiala funktionen `trace()` för variabeln `num` inträffar innan variabeln `num` har deklarerats:

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

Kompilatorn lyfter inte några tilldelningsprogramsatser. Detta förklarar varför den initiala funktionen `trace()` för `num` resulterar i `NaN` (not a number), vilket är standardvärdet för variabler för datatypen Number. Detta betyder att du kan tilldela variabler värden även innan de har deklarerats, vilket visas i följande exempel:

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

Standardvärden

Ett *standardvärde* är variabelns värde innan du anger ett värde för den. Du *initierar* en variabel när du bestämmer värdet första gången. Om du deklarerar en variabel, men inte bestämmer något värde, kommer variabeln att vara *oinitierad*. Värdet på en oinitierad variabel beror på variabelns datatyp. I följande tabell beskrivs standardvärden för variabler, ordnade efter datatyp:

Datatyp	**Standardvärde**
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`

Datatyp	Standardvärde
uint	0
Ej deklarerad (motsvarar en typanteckning `*`)	`undefined`
Alla andra klasser, däribland användardefinierade klasser.	`null`

För variabler av typen Number är standardvärdet `NaN` (not a number) som är ett speciellt värde som definieras av IEEE-754-standarden och som betyder att det är ett värde som inte representerar ett tal.

Om du deklarerar en variabel, men inte deklarerar dess datatyp, används standarddatatypen `*`, vilket betyder att variabeln är typlös. Om du heller inte initierar en typlös variabel med ett värde, blir standardvärdet `undefined`.

För andra datatyper än Boolean, Number, int och uint är standardvärdet för alla oinitierade variabler `null`. Detta gäller alla klasser som har definierats av ActionScript 3.0, men även alla de klasser som du själv skapar.

Värdet `null` är inte ett giltigt värde för variabler av typen Boolean, Number, int och uint. Om du försöker tilldela en sådan variabel värdet `null` konverteras det till standardvärdet för den datatypen. Du kan däremot tilldela värdet `null` för variabler av typen Object. Om du försöker tilldela en variabel av typen Object värdet `undefined`, konverteras värdet till `null`.

För variabler av typen Number finns det en speciell funktion på översta nivån som heter `isNaN()` som returnerar det booleska värdet `true` om variabeln inte är ett tal och i annat fall `false`.

Datatyper

En *datatyp* definierar en serie värden. Datatypen Boolean är en serie av exakt två värden: `true` och `false`. Förutom datatypen Boolean definieras flera vanligare datatyper i ActionScript 3.0, däribland String, Number och Array. Du kan definiera egna datatyper genom att använda klasser eller gränssnitt för att definiera en egen uppsättning värden. Alla värden i ActionScript 3.0 är objekt, vare sig de är primitiva eller komplexa,.

Ett *primitivt värde* är ett värde som hör till någon av följande datatyper: Boolean, int, Number, String och uint. Det går oftast snabbare att arbeta med primitiva värden än att arbeta med komplexa värden, eftersom primitiva värden lagras i ActionScript på ett speciellt sätt som gör minnes- och hastighetsoptimering möjlig.

Obs! *I ActionScript lagras primitiva värden internt som oföränderliga objekt, vilket kan vara intressant att veta. Det faktum att de lagras som oföränderliga objekt betyder att överföring med referens är detsamma som överföring med värde. Detta minskar minnesanvändningen och ökar körningshastigheten eftersom referenserna vanligtvis är mycket mindre än själva värdena.*

Ett *komplext värde* är ett värde som inte är primitivt. Datatyper som definierar uppsättningar av komplexa värden är bland annat Array, Date, Error, Function, RegExp, XML och XMLList.

I många programmeringsspråk skiljer man mellan primitiva värden och deras wrapper-objekt. Java har till exempel det primitiva värdet int och klassen java.lang.Integer som omsluter värdet. Primitiva värden i Java är inte objekt, men deras wrapprar är det, vilket gör att primitiva värden är bra för vissa åtgärder och wrapper-objekt är bättre för andra åtgärder. I ActionScript 3.0 är primitiva värden och deras wrapper-objekt omöjliga att särskilja. Alla värden, även primitiva, är objekt. I körningsmiljön behandlas dessa primitiva typer som specialfall som beter sig som objekt, men som inte kräver den kapacitet som normalt associeras med att skapa objekt. Det betyder att följande två kodrader är likvärdiga:

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

Alla ovanstående primitiva och komplexa datatyper definieras av huvudklasserna i ActionScript 3.0. Med huvudklasserna kan du skapa objekt med hjälp av litterala värden i stället för att använda operatorn `new`. Du kan till exempel skapa en array med ett litteralt värde eller klasskonstruktorn Array så här:

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

Typkontroll

Typkontroll utförs antingen vid kompilering eller vid körning. Språk med statiska typer, som C++ och Java, utför typkontroll vid kompileringen. Språk med dynamiska typer, som Smalltalk och Python, hanterar typkontroll vid körningen. ActionScript 3.0, som är ett språk med dynamiska typer, har en typkontroll som utförs vid körningen, men kan även använda en typkontroll som utförs vid kompileringen tack vare ett speciellt kompileringsläge som kallas *strikt läge*. I strikt läge utförs typkontrollen både vid kompilering och vid körning, men i standardläge utförs typkontroll bara vid körning.

Språk som har dynamiska typer är otroligt flexibla när du strukturerar kod men genererar typfel vid körningen. Språk med statiska typer rapporterar typfel vid kompileringen och typinformation krävs vid kompileringen.

Typkontroll vid kompilering

Typkontroll vid kompilering föredras ofta i större projekt. Detta beror på, att när man arbetar med stora projekt blir det viktigare att hitta typfel så tidigt som möjligt än att man använder en flexibel datatyp. Det är därför som ActionScript-kompilatorn i Flash Professional och Flash Builder som standard är inställd på att köras i strikt läge.

Adobe Flash Builder

Du kan inaktivera det strikta läget i Flash Builder via inställningarna för ActionScript-kompilatorn i dialogrutan Project Properties.

För att kompilatorn ska kunna utföra typkontroll vid kompilering måste koden innehålla datatypsinformation om variabler och uttryck. Om du vill deklarera en datatyp för en variabel lägger du till kolonoperatorn (`:`) följt av datatypen som suffix i variabelnamnet. Om du vill associera en datatyp med en parameter använder du kolonoperatorn följt av datatypen. I följande kod läggs datatypsinformation till i parametern `xParam` och deklareras variabeln `myParam` med en explicit datatyp:

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

I strikt läge rapporterar ActionScript-kompilatorn felaktig matchning av typer som kompileringsfel. I följande kod deklareras funktionsparametern `xParam`, för typen Object, men försök görs senare att tilldela värden av typen String och Number till den parametern. Detta skapar ett kompileringsfel i strikt läge.

```
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);
    }
}
```

Du kan emellertid även i strikt läge selektivt välja bort typkontroll vid kompilering genom att låta höger sida av en tilldelningsprogramsats vara typlös. Du kan markera en variabel eller ett uttryck som typlöst genom att antingen utesluta en typanteckning eller använda den speciella asterisktypanteckningen (`*`). Om parametern `xParam` i föregående exempel ändras så att den inte längre har en typanteckning kompileras koden i strikt läge:

```
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");
```

Typkontroll vid körning

Typkontroll vid körning utförs i ActionScript 3.0 vare sig du kompilerar i strikt läge eller i standardläge. Anta att värdet 3 skickas som ett argument till en funktion som förväntar sig en array. I strikt läge genereras ett fel eftersom värdet 3 inte är kompatibelt med datatypen Array. Om du inaktiverar strikt läge och kör i standardläge reagerar kompilatorn inte över typmatchningsfelet, men typkontrollen vid körning resulterar i ett körningsfel.

I följande exempel visas funktionen `typeTest()` som förväntar sig ett Array-argument men som tar emot värdet 3. Detta förorsakar ett körningsfel i standardläge eftersom värdet 3 inte ingår i parameterns deklarerade datatyp (Array).

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

Du kan också i vissa situationer få ett körningsfel även när du arbetar i strikt läge. Detta kan hända om du använder strikt läge men väljer bort typkontroll vid kompilering genom att använda en typlös variabel. När du använder en typlös variabel väljer du inte bort typkontrollen, utan fördröjer den till körningen. Om till exempel variabeln `myNum` i föregående exempel inte har en deklarerad datatyp kan kompilatorn inte identifiera typmatchningsfelet. Koden genererar då ett körningsfel, eftersom körningsvärdet för `myNum`, som har värdet 3 som ett resultat av tilldelningsprogramsatsen, jämförs med typen för `xParam`, som är inställd på datatypen Array.

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

Typkontroll vid körning tillåter flexiblare användning av arv än typkontroll vid kompilering. Genom att fördröja typkontrollen till körningen kan du i standardläge referera egenskaper av en underklass även om du använder *upcast*. En upcast inträffar när du använder en basklass för att deklarera typ av klassinstans men en underklass för att instansiera den. Du kan alltså skapa en klass med namnet ClassBase som kan utökas (klasser med attributet `final` kan inte utökas):

```
class ClassBase
{
}
```

Du kan därefter skapa en underklass för ClassBase som heter ClassExtender som har en egenskap som heter `someString` enligt följande:

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

Om du använder båda klasserna kan du skapa en klassinstans som deklareras med datatypen ClassBase men som instansieras med konstruktorn ClassExtender. En upcast är en säker åtgärd eftersom basklassen bara innehåller egenskaper och metoder som finns i underklassen.

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

En underklass innehåller däremot egenskaper och metoder som basklassen inte innehåller. Klassen ClassExtender innehåller till exempel egenskapen `someString` som inte finns i klassen ClassBase. I standardläge i ActionScript 3.0 kan du referera den här egenskapen med instansen `myClass` utan att generera ett kompileringsfel, vilket visas i följande exempel:

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

Operatorn is

Med operatorn `is` kan du testa om en variabel eller ett uttryck är medlem i en viss datatyp. I tidigare versioner av ActionScript hade operatorn `instanceof` den här funktionaliteten, men i ActionScript 3.0 ska operatorn `instanceof` inte användas för att testa datatypsmedlemskap. Använd operatorn `is` i stället för operatorn `instanceof` för manuell typkontroll eftersom uttrycket `x instanceof y` snarare kontrollerar om prototypkedjan för `x` innehåller `y` (och i ActionScript 3.0 ger prototypkedjan inte en fullständig bild av arvshierarkin).

Operatorn `is` kontrollerar arvshierarkin och kan användas för att kontrollera inte bara om ett objekt är en instans av en viss klass, utan även om objektet är en instans av en klass som implementerar ett visst gränssnitt. I följande exempel skapas en instans av klassen Sprite som heter `mySprite` och använder operatorn `is` för att testa om `mySprite` är en instans av klasserna Sprite och DisplayObject, och huruvida den implementerar gränssnittet IEventDispatcher:

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

Operatorn `is` kontrollerar arvshierarkin och rapporterar att `mySprite` är kompatibel med klasserna Sprite och DisplayObject (klassen Sprite är en underklass till klassen DisplayObject). Operatorn `is` kontrollerar också om `mySprite` ärver från några klasser som implementerar gränssnittet IEventDispatcher. Eftersom klassen Sprite ärver från klassen EventDispatcher, som implementerar gränssnittet IEventDispatcher, rapporterar operatorn `is` att `mySprite` implementerar samma gränssnitt.

I följande exempel visas samma tester från föregående exempel, men med `instanceof` i stället för operatorn `is`. Operatorn `instanceof` identifierar att `mySprite` är en instans av Sprite eller DisplayObject, men den returnerar `false` när den används för att testa om `mySprite` implementerar gränssnittet IEventDispatcher.

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

Operatorn as

Med operatorn `as` kan du testa om ett uttryck är medlem i en viss datatyp. l motsats till operatorn `is` returnerar operatorn `as` inte ett booleskt värde. I stället returnerar operatorn `as` värdet för uttrycket i stället för `true`, och `null` i stället för `false`. I följande exempel visas resultatet av att använda operatorn `as` i stället för operatorn `is` vid en enkel kontroll av om instansen Sprite är medlem i datatypen DisplayObject, IEventDispatcher och 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
```

När du använder operatorn `as` måste operanden till höger vara en datatyp. Om du försöker använda ett uttryck som inte är en datatyp som operand till höger genereras ett fel.

Dynamiska klasser

Med en *dynamisk* klass definieras ett objekt som du kan ändra vid körning genom att lägga till eller ändra egenskaper och metoder. En klass som inte är dynamisk, till exempel klassen String, är en *fast* klass. Du kan inte lägga till egenskaper och metoder i en fast klass vid körning.

Du kan skapa dynamiska klasser med attributet `dynamic` när du deklarerar en klass. Med följande kod skapas en dynamisk klass som heter `Protean`:

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

Om du senare instansierar en instans av klassen `Protean` kan du lägga till egenskaper och metoder i den utanför klassdefinitionen. Med följande kod skapas till exempel en instans av klassen `Protean` och läggs egenskapen `aString` och egenskapen `aNumber` till i instansen.

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

Egenskaper som du lägger till i en instans av en dynamisk klass är körningsenheter, och all typkontroll utförs vid körning. Du kan inte lägga till en typanteckning i en egenskap som du lägger till på det här sättet.

Du kan också lägga till en metod i instansen `myProtean` genom att definiera en funktion och bifoga den till en egenskap för instansen `myProtean`. Följande kod flyttar programsatsen trace till metoden `traceProtean()`:

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

Metoder som skapats på det här sättet kommer inte åt privata egenskaper eller metoder av klassen Protean. Referenser till allmänna egenskaper eller metoder för klassen `Protean` måste dessutom kvalificeras med antingen nyckelordet `this` eller med klassnamnet. I följande exempel visas hur metoden `traceProtean()` försöker komma åt privata och allmänna variabler för klassen `Protean`.

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

Beskrivning av datatyper

Primitiva datatyper kan vara Boolean, int, Null, Number, String, uint och void. Huvudklasserna i ActionScript definierar också följande komplexa datatyper: Object, Array, Date, Error, Function, RegExp, XML och XMLList.

Boolean, datatyp

Datatypen Boolean kan ha två värden: `true` och `false`. Inga andra värden är giltiga för den här datatypen. Standard för en boolesk variabel som är deklarerad, men inte initierad, är `false`.

int, datatyp

Datatypen int lagras internt som 32-bitars heltal och har en uppsättning heltal från

-2 147 483 648 (-2^{31}) till och med 2 147 483 647 ($2^{31} - 1$). Tidigare versioner av ActionScript hade bara datatypen Number som användes för både heltal och flyttal. I ActionScript 3.0 kan du använda lågnivåmaskintyper för 32-bitars heltal med och utan tecken. Om din variabel inte använder flyttal går det snabbare och är effektivare om du använder datatypen int i stället för datatypen Number.

För heltalsvärden som ligger utanför minimi- och maximivärdena för int använder du datatypen Number, som kan hantera värden mellan positiva och negativa 9 007 199 254 740 992 (53-bitars heltalsvärden). Standardvärdet för variabler som är av datatypen int är 0.

Null, datatyp

Datatypen Null innehåller bara ett värde, `null`. Detta är standardvärdet för datatypen String och alla klasser som definierar komplexa datatyper, däribland klassen Object. Ingen av de primitiva datatyperna, till exempel Boolean, Number, int eller uint, innehåller värdet `null`. Vid körning konverteras värdet `null` till lämpligt standardvärde om du försöker tilldela `null` till variabler av typen Boolean, Number, int eller uint. Du kan inte använda den här datatypen som en typanteckning.

Number, datatyp

I ActionScript 3.0 kan den här datatypen representera heltal, heltal utan tecken och flyttal. Om du vill maximera prestanda bör du använda datatypen Number bara för heltal som är större än vad 32-bitarstyperna `int` och `uint` kan lagra eller för flyttal. Om du vill lagra flyttal måste du infoga ett decimalkomma i talet. Om du utesluter decimalkommat sparas talet som ett heltal.

Datatypen Number använder 64-bitars dubbelprecisionsformat som anges av IEEE:s standard för binär flyttalsaritmetik (IEEE-754). Denna standard anger hur flyttal lagras med 64 tillgängliga bitar. En bit används för att ange om talet är positivt eller negativt. Elva bitar används för exponenten som lagras som bas 2. De återstående 52 bitarna används för att lagra *signifikanden* (kallas också *mantissa*), som är det tal som upphöjs till det som anges av exponenten.

Genom att använda vissa bitar för att lagra en exponent kan datatypen Number lagra flyttal som är mycket större än om den använde alla bitarna för signifikanden. Om datatypen Number till exempel använder alla 64 bitar för att lagra signifikanden, bör den lagra ett tal som är så stort som $2^{65} - 1$. Genom att använda elva bitar för att lagra en exponent kan datatypen Number upphöja signifikanden till 2^{1023}.

Maximi- och minimivärdena som typen Number kan representera lagras i statiska egenskaper för klassen Number som kallas `Number.MAX_VALUE` respektive `Number.MIN_VALUE`.

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

Även om det här talintervallet är enormt är priset för intervallet precision. Datatypen Number använder 52 bitar för att lagra signifikanden, vilket resulterar i att tal som kräver mer än 52 bitar för att representeras exakt, till exempel bråket 1/3, bara blir approximationer. Om ditt program kräver absolut precision med decimaltal måste du använda program som implementerar decimalflyttalsaritmetik i motsats till binär flyttalsaritmetik.

När du lagrar heltalsvärden med datatypen Number används bara de 52 bitarna av signifikanden. Datatypen Number använder dessa 52 bitar och en speciell dold bit för att representera heltal från -9 007 199 254 740 992 (-2^{53}) till 9 007 199 254 740 992 (2^{53}).

Värdet `NaN` används inte bara som standardvärde för variabler av typen `Number`, utan också som resultatet av en beräkning som ska returnera ett tal men som inte gör det. Om du till exempel försöker beräkna kvadratroten av ett negativt tal blir resultatet `NaN`. Andra speciella Number-värden är *positiv oändlighet* och *negativ oändlighet*.

***Obs!** Resultatet av division med `0` är bara `NaN` om divisorn också är `0`. Division med `0` skapar `infinity` när dividenden är positiv eller `-infinity` när dividenden är negativ.*

String, datatyp

Datatypen String representerar en sekvens av 16-bitarstecken. Strängar sparas internt som Unicode-tecken med UTF-16-formatet. Strängar är oföränderliga värden, på samma sätt som i programmeringsspråket Java. En operation på ett String-värde resulterar i en ny instans av strängen. Standardvärdet för en variabel som har deklarerats med datatypen String är `null`. Värdet `null` är inte samma sak som den tomma strängen (`""`). Värdet `null` betyder att variabeln saknar ett sparat värde, medans en tom sträng betyder att variabeln har ett värde som är en sträng utan några tecken.

uint, datatyp

Datatypen uint lagras internt som ett 32-bitars heltal utan tecken och innehåller en serie heltal från 0 till och med 4 294 967 295 (2^{32} - 1). Använd datatypen uint vid speciella tillfällen som anropar icke-negativa heltal. Du måste till exempel använda datatypen uint för att representera pixelvärden i färg eftersom datatypen int har en intern bit med tecken som inte är lämplig för hantering av färgvärden. Om det gäller heltal som är större än maximivärdet för uint använder du datatypen Number som kan hantera 53-bitars heltal. Standardvärdet för variabler som är av datatypen uint är 0.

void, datatyp

Datatypen void innehåller bara ett värde, `undefined`. I tidigare versioner av ActionScript var `undefined` standardvärde för instanser av klassen Object. I ActionScript 3.0 är standardvärdet för Object-instanser `null`. Om du försöker tilldela en instans av klassen Object värdet `undefined` konverteras värdet till `null`. Du kan bara tilldela variabler som är typlösa värdet `undefined`. Typlösa variabler är variabler som antingen saknar typanteckning eller som använder symbolen asterisk (`*`) för typanteckning. Du kan använda `void` bara som returtypsanteckning.

Object, datatyp

Datatypen Object definieras av klassen Object. Klassen Object fungerar som basklass för alla klassdefinitioner i ActionScript. ActionScript 3.0-versionen av datatypen Object skiljer sig på tre sätt från datatypen i tidigare versioner. För det första är datatypen Object inte längre standarddatatyp som tilldelas variabler utan typanteckning. För det andra innehåller datatypen Object inte längre värdet `undefined`, som används som standardvärde för Object-instanser. För det tredje är standardvärdet för klassen Object `null` i ActionScript 3.0.

I tidigare versioner av ActionScript har en variabel utan typanteckning automatiskt tilldelats datatypen Object. Detta gäller inte längre i ActionScript 3.0 som nu innehåller en helt typlös variabel. Variabler utan typanteckning betraktas nu som typlösa. Om du vill visa läsarna av din kod att du avsiktligen har lämnat en variabel typlös kan du använda en asterisk (`*`) för typanteckningen, vilket motsvarar att utesluta en typanteckning. I följande exempel visas två likvärdiga programsatser som båda deklarerar den typlösa variabeln `x`:

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

Endast typlösa variabler kan innehålla värdet `undefined`. Om du försöker tilldela en variabel som har en datatyp värdet `undefined` konverteras värdet `undefined` till standardvärdet för den datatypen vid körning. För instanser av datatypen Object är standardvärdet `null`, vilket betyder att om du försöker tilldela värdet `undefined` till en Object-instans konverteras värdet till `null`.

Typkonverteringar

En typkonvertering inträffar när ett värde transformeras till ett värde av en annan datatyp. Typkonverteringar kan antingen vara *implicita* eller *explicita*. Implicit konvertering, som även kallas *tvång*, utförs ibland vid körning. Om till exempel värdet 2 tilldelas en variabel med datatypen Boolean konverteras värdet 2 till det booleska värdet `true` innan variabeln tilldelas värdet. Explicit konvertering, som också kallas *datatypsbyte*, inträffar när koden instruerar kompilatorn att hantera en variabel av en viss datatyp som om den hör till en annan datatyp. När primitiva värden ingår konverteras värden från en datatyp till en annan datatyp med datatypsbytet. När du vill byta datatyp för ett objekt omsluter du objektnamnet med parentes och skriver namnet på den nya datatypen framför. I följande kod byts en boolesk variabel ut mot en heltalsvariabel:

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

Implicita konverteringar

Implicita konverteringar inträffar vid körning i olika sammanhang:

- I tilldelningsprogramsatser
- När värden skickas som funktionsargument
- När värden returneras från funktioner
- I uttryck som använder vissa operatorer, till exempel additionsoperatorn (+)

  För användardefinierade typer lyckas implicita konverteringar när värdet som ska konverteras är en instans av destinationsklassen eller en klass som härstammar från destinationsklassen. Om en implicit konvertering misslyckas uppstår ett fel. Följande kod innehåller en lyckad och en misslyckad implicit konvertering:

  ```
  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.
  ```

  Om det gäller primitiva typer hanteras implicita konverteringar genom anrop till samma interna konverteringsalgoritmer som anropas av de explicita konverteringsfunktionerna.

Explicita konverteringar

Du bör använda explicita konverteringar, eller datatypsbyte, när du kompilerar i strikt läge eftersom du kanske inte vill att ett typmatchningsfel ska generera ett kompileringsfel. Detta kan inträffa när du vet att tvång kommer att konvertera dina värden korrekt vid körning. När du till exempel arbetar med data som du fått från ett formulär, kanske du vill förlita dig på tvång för att konvertera vissa strängvärden till numeriska värden. I följande kod genereras ett kompileringsfel även om koden kan köras korrekt i standardläge:

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

Om du vill fortsätta i strikt läge, men vill att strängen ska konverteras till ett heltal, kan du använda explicit konvertering enligt följande:

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

Datatypsbyte till int, uint och Number

Du kan byta ut en datatyp till någon av de tre taltyperna int, uint och Number. Om det av någon anledning inte går att konvertera talet tilldelas datatyperna int och uint standardvärdet 0, medan datatypen Number tilldelas standardvärdet `NaN`. Om du konverterar ett booleskt värde till ett tal blir `true` värdet 1 och `false` blir värdet 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
```

Strängvärden som innehåller bara siffror kan konverteras till någon av taltyperna. Taltyperna kan också konvertera strängar som ser ut som negativa tal eller strängar som representerar ett hexadecimalt värde (till exempel `0x1A`). Vid konverteringen ignoreras inledande och avslutande tomrum i strängvärdet. Du kan också byta strängar som ser ut som flyttal med `Number()`. Inkluderingen av ett decimaltecken gör att `uint()` och `int()` returnerar ett heltal och decimaltecknet och de tecken som kommer efter trunkeras. Följande strängvärden kan till exempel bytas till tal:

```
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
```

Strängvärden som innehåller icke-numeriska tecken returnerar 0 när de byts med `int()` eller `uint()` och `NaN` när de byts med `Number()`. Vid konverteringen ignoreras inledande och avslutande tomrum, men 0 eller `NaN` returneras om en sträng har ett tomrum som avgränsar två tal.

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

I ActionScript 3.0 stöder funktionen `Number()` inte längre oktala (eller bas-8) tal. Om du använder en sträng med en inledande nolla i funktionen `Number()` i ActionScript 2.0 tolkas talet som ett oktalt tal och konverteras till ett decimaltal. Detta gäller inte med funktionen `Number()` i ActionScript 3.0 där nollan ignoreras. Med följande kod genereras olika utdata vid kompilering med olika versioner av ActionScript:

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

Datatypsbyte är nödvändigt när ett värde av en viss numerisk typ tilldelas en variabel av en annan numerisk typ. Även i strikt läge konverteras de numeriska typerna implicit till andra numeriska typer. Detta betyder att man i vissa fall får oväntat resultat när intervallet för en typ överskrids. Följande exempel kompileras alla i strikt läge, men vissa av dem genererar oväntade värden:

```
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
```

I följande tabell sammanfattas resultatet av datatypsbyte till Number, int eller uint från andra datatyper.

Datatyp eller värde	**Resultat av konvertering till Number, int eller uint**
Boolean	Om värdet är `true`: 1, annars 0.
Date	Den interna beteckningen för objektet Date, som är antalet millisekunder som förflutit sedan midnatt den 1 januari 1970, universaltid.
`null`	0
Object	Om instansen är `null` och konverterat till Number `NaN`; annars 0.
String	Ett tal om det går att konvertera strängen till ett tal; i annat fall `NaN` om det konverteras till Number eller 0 om det konverteras till int eller uint.
`undefined`	`NaN` om den konverteras till Number. 0 om den konverteras till int eller uint.

Datatypsbyte till Boolean

Datatypsbyte till Boolean från någon av de numeriska datatyperna (uint, int eller Number) resulterar i `false` om det numeriska värdet är 0, och i annat fall i `true`. Om du använder datatypen Number kan värdet `NaN` också resultera i `false`. I följande exempel visas resultatet av datatypsbyte av talen -1, 0 och 1:

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

Utdata från exemplet visar att värdet `false` bara returneras av talet 0:

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

Databyte till Boolean från ett strängvärde returnerar `false` om strängen är `null` eller en tom sträng (`""`). I annat fall returneras `true`.

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

Databyte till Boolean från en instans av klassen Object returnerar `false` om instansen är `null` och returnerar annars `true`:

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

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

Booleska variabler får specialbehandling i strikt läge på det sättet att du kan tilldela ett booleskt värde värden av valfri datatyp utan att använda databyte. Implicit tvång från alla datatyper till datatypen Boolean inträffar även i strikt läge. Databyte till Boolean är inte nödvändigt för att undvika fel i strikt läge i motsats till byte till andra datatyper. I följande exempel kompileras allt i strikt läge och beteendet blir väntat vid körning:

```
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
```

I följande tabell sammanfattas resultatet av datatypsbyte till Boolean från andra datatyper:

Datatyp eller värde	**Resultat av konvertering till Boolean**
String	`false` om värdet är `null` eller tom sträng (`""`). Annars `true`.
`null`	`false`
Number, int eller uint	`false` om värdet är `NaN` eller 0; annars `true`.
Object	`false` om instansen är `null`; annars `true`.

Databyte till String

Vid databyte till datatypen String från någon av de numeriska datatyperna returneras en strängbeteckning för talet. Vid databyte till datatypen String från ett booleskt värde returneras strängen `"true"`om värdet är `true` och strängen `"false"` om värdet är `false`.

Vid databyte till datatypen String från en instans av klassen Objekt returneras strängen`"null"` om instansen är `null`. I annat fall vid databyte till typen Sträng från klassen Object returneras strängen `"[object Object]"`.

Vid databyte till String från en instans av klassen Array returneras en sträng som innehåller en semikolonavgränsad lista på alla array-element. I följande databyte till datatypen String returneras en sträng som innehåller alla tre elementen av arrayen:

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

Vid databyte till String från en instans av klassen Date returneras en strängbeteckning för det datum som instansen innehåller. I följande exempel returneras en strängbeteckning för klassinstansen Date (utdata ger resultatet för Pacific Daylight Time):

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

I följande tabell sammanfattas resultatet av datatypsbyte till datatypen String från andra datatyper.

Datatyp eller värde	Resultat av konvertering till String
Array	En sträng som består av alla array-element.
Boolean	"`true` eller `false`"
Date	En strängbeteckning för objektet Date.
`null`	`"null"`
Number, int eller uint	En strängbeteckning för talet.
Object	Om instansen är null `"null"`. Annars `"[object Object]"`.

Syntax

Med ett språks syntax definieras en uppsättning regler som måste följas när man skriver körbar kod.

Skiftlägeskänslighet

ActionScript 3.0 är ett skiftlägeskänsligt språk. Identifierare som bara är olika när det gäller skiftläge betraktas som olika identifierare. I följande kod skapas två olika variabler:

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

Punktsyntax

Med punktoperatorn (.) kommer du åt ett objekts egenskaper och metoder. Om du använder punktsyntax kan du referera till en klassegenskap eller metod med hjälp av ett instansnamn följt av punktoperatorn och namnet på egenskapen eller metoden. Du kan t.ex. använda följande klassdefinition:

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

Om du använder punktsyntax kommer du åt egenskapen `prop1` och metoden `method1()` med instansnamnet som skapats i följande kod:

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

Du kan använda punktsyntax när du definierar paket. Du kan använda punktoperatorn för att referera till kapslade paket. Exempel: Klassen EventDispatcher finns i paketet som heter events som är kapslat i paketet som heter flash. Du kan referera till paketet events med följande uttryck:

```
flash.events
```

Du kan också referera till klassen EventDispatcher med det här uttrycket:

```
flash.events.EventDispatcher
```

Snedstreckssyntax

Snedstreckssyntax kan inte användas i ActionScript 3.0. Snedstreckssyntax användes i tidigare versioner av ActionScript för att ange sökvägen för ett filmklipp eller en variabel.

Litteraler

En *litteral* är ett värde som visas direkt i koden. Följande exempel är också litteraler:

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

Litteraler kan även placeras i grupper och bildar då sammansatta litteraler. Arraylitteraler placeras inom hakparentes (`[]`) och använder kommatecken för att avskilja arrayelementen.

En arraylitteral kan användas till att initiera en array. I följande exempel visas två arrayer som initierats med arraylitteraler. Du kan använda programsatsen `new` och skicka den sammansatta litteralen som en parameter till array-klasskonstruktorn, men du kan också tilldela litteralvärden direkt när du instansierar instanser av följande ActionScript-huvudklasser: Object, Array, String, Number, int, uint, XML, XMLList och 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];
```

Du kan använda litteraler till att initiera ett generiskt objekt. Ett generiskt objekt är en instans av objektklassen. Objektlitteraler placeras inom klammerparentes (`{}`) och du använder komma för att skilja objektegenskaper åt. Varje egenskap deklareras med kolon (`:`) som skiljer namnet på egenskapen från egenskapens värde.

Du kan skapa ett generiskt objekt med hjälp av programsatsen `new` och skickar objektlitteralen som en parameter till objektklasskonstruktorn eller också kan du tilldela objektlitteralen direkt till instansen som du deklarerar. I följande exempel visas två olika sätt att skapa ett nytt generiskt objekt och initiera objektet med tre egenskaper (`propA`, `propB` och `propC`) som har värdena 1, 2 och 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};
```

Fler hjälpavsnitt

Arbeta med strängar

Använda reguljära uttryck

Initiera XML-variabler

Semikolon

Du kan använda semikolon (`;`) för att avsluta en programsats. Om du däremot utelämnar semikolonet förutsätter ActionScript-kompilatorn att varje rad med kod representerar en programsats. Eftersom många programmerare brukar använda semikolon för att ange slutet på en programsats, blir din kod lättare att läsa om du använder semikolon.

Om du använder semikolon för att avsluta en programsats kan du placera mer än en programsats på samma rad, men då blir koden vanligtvis svårare att läsa.

Parentes

Du kan använda parentes (`()`) på tre olika sätt i ActionScript 3.0. För det första kan du använda parentes för att ändra ordningen på åtgärderna i ett uttryck. Åtgärder som grupperats inom parentesen körs alltid först. Parentes används till exempel för att ändra ordningen på åtgärderna i följande kod:

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

För det andra kan du använda parentes med kommaoperatorn (`,`) för att utvärdera en serie uttryck och returnera resultatet av det slutliga uttrycket, vilket visas i följande exempel:

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

För det tredje kan du använda parentes för att skicka en eller flera parametrar till funktioner och metoder, vilket visas i följande exempel, som skickar ett strängvärde till funktionen `trace()`:

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

Kommentarer

Du kan använda två olika sorters kommentarer i ActionScript 3.0-kod: enkelradskommentarer och flerradskommentarer. Kommentarsfunktionen liknar samma funktion i C++ och Java. Vid kompileringen ignoreras text som markerats som en kommentar.

Enkelradskommentarer inleds med två snedstreck (`//`) och fortsätter till radens slut. I följande kod visas en enkelradskommentar:

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

Flerradskommentarer inleds med ett snedstreck och en asterisk (`/*`) och avslutas med en asterisk och ett snedstreck (`*/`).

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

Nyckelord och reserverade ord

Reserverade ord är ord som du inte kan använda som identifierare i din kod eftersom orden är reserverade för ActionScript. Reserverade ord innehåller *lexikala nyckelord* som tas bort från programmets namnutrymme av kompilatorn. Ett fel rapporteras vid kompilering om du använder ett lexikalt nyckelord som identifierare. I följande tabell visas lexikala nyckelord i ActionScript 3.0.

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

Det finns några nyckelord som kallas *syntaktiska nyckelord* som kan användas som identifierare, och som har speciell betydelse i vissa sammanhang. I följande tabell visas syntaktiska nyckelord i ActionScript 3.0.

each	get	set	namespace
include	dynamic	final	native
override	static		

Det finns också olika identifierare som ibland kallas *framtida reserverade ord*. Dessa identifierare har inte reserverats för ActionScript 3.0 även om vissa av dem kanske hanteras som nyckelord av programvara som använder ActionScript 3.0. Du kan använda en hel del av dessa identifierare i din kod, men Adobe rekommenderar inte detta eftersom de kan visas som nyckelord i en kommande version av språket.

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

Konstanter

ActionScript 3.0 har stöd för programsatsen `const` som du använder för att skapa konstanter. Konstanter är egenskaper med ett fast värde som inte kan ändras. Du kan bara tilldela en konstant ett värde en gång, och tilldelningen måste ske i omedelbar närhet till konstantens deklaration. Om en konstant deklareras som en medlem av en klass kan du tilldela konstanten ett värde som en del av deklarationen eller inuti klasskonstruktorn.

I följande kod deklareras två konstanter. Den första konstanten `MINIMUM` har tilldelats ett värde som en del av deklarationsprogramsatsen. Den andra konstanten `MAXIMUM` har tilldelats ett värde i konstruktorn. Observera att det här exemplet bara kompileras i standardläge eftersom en konstants värde bara tilldelas vid initieringen i strikt läge.

```
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
```

Ett fel uppstår om du försöker tilldela en konstant ett initialt värde på något annat sätt. Om du till exempel försöker ange startvärdet för `MAXIMUM` utanför klassen inträffar ett körningsfel.

```
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 definierar en mängd konstanter som du kan använda. Konstanter i ActionScript skrivs med versaler och två ord avgränsas med understreck (_). Exempel: I klassdefinitionen för MouseEvent används den här namnkonventionen för konstanterna, som representerar en händelse som relateras till musindata:

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

Operatorer

Operatorer är speciella funktioner som använder en eller flera operander och returnerar ett värde. En *operand* är ett värde, vanligtvis en litteral, en variabel eller ett uttryck, som används som indata av en operator. I följande kod används additions- (`+`) och multiplikationsoperatorn (`*`) med tre litterala operander (2, 3 och 4) för att returnera ett värde. Värdet används därefter av tilldelningsoperatorn (`=`) för att tilldela variabeln `sumNumber` det returnerade värdet 14.

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

Operatorer kan vara enställiga, binära eller ternära. En *unär operator* tar en operand. Ökningsoperatorn (`++`) är till exempel en unär operator eftersom den bara tar en operand. En *binär* operator tar två operander. Divisionsoperatorn (`/`) tar till exempel två operander. En *ternär* operator tar tre operander. Den villkorliga operatorn (`?:`) tar till exempel tre operander.

Vissa operatorer är *överlastade*, vilket betyder att de beter sig på olika sätt beroende på vilken typ eller kvantitet av operander som skickas till dem. Additionsoperatorn (`+`) är ett exempel på en överlastad operator som beter sig på olika sätt beroende på operandernas datatyp. Om båda operanderna är tal returneras summan av värdena. Om båda operanderna är strängar returneras sammanfogningen av de två operanderna. I följande exempelkod visas hur operatorn beter sig på olika sätt beroende på operanderna:

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

Operatorer kan också bete sig på olika sätt beroende på hur många operander det finns. Subtraktionsoperatorn (`-`) är både unär och binär operator. När det bara finns en operand negeras operanden av subtraktionsoperatorn och resultatet returneras. När det finns två operander returneras skillnaden mellan operanderna. I följande exempel visas hur subtraktionsoperatorn först används som unär operator och sedan som binär operator.

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

Operatorprioritet och associativitet

Med operatorprioritet och associativitet avgörs i vilken ordning som operatorerna behandlas. Även om det kan verka naturligt för dem som är bekanta med aritmetik att kompilatorn behandlar multiplikationsoperatorn (`*`) före additionsoperatorn (`+`), så behöver kompilatorn uttryckliga instruktioner om vilka operatorer den ska behandla först. Sådana instruktioner kallas för *operatorprioritet*. I ActionScript definieras en standardoperatorprioritet som du kan ändra genom att använda parentesoperatorn (`()`). I följande kod ändras standardprioriteten från föregående exempel så att kompilatorn tvingas bearbeta additionsoperatorn före multiplikationsoperatorn:

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

Du kan komma att stöta på situationer i vilka två eller flera operatorer med samma prioritet förekommer i samma uttryck. I dessa fall använder kompilatorn reglerna för *associativitet* för att avgöra vilken operator som ska behandlas först. Samtliga binära operatorer, förutom tilldelningsoperatorerna, är *vänster-associerande*, vilket innebär att operatorer till vänster behandlas före operatorer till höger. Tilldelningsoperatorer och villkorliga operatorer (`?:`) är *höger-associativa*, vilket innebär att operatorer till höger behandlas före operatorer till vänster.

Låt oss ta operatorerna för mindre än (`<`) och större än (`>`) som har samma prioritet. Om båda operatorerna används i samma uttryck behandlas operatorn till vänster först eftersom båda operatorerna har vänster-associativitet. Detta innebär att de två följande programsatserna ger samma utdata:

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

Operatorn för större än behandlas först, vilket ger värdet `true` eftersom operanden 3 är större än operanden 2. Värdet `true` skickas sedan till operatorn för mindre än tillsammans med operanden 1. Följande kod anger det här mellanläget:

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

Operatorn mindre än konverterar värdet `true` till det numeriska värdet 1 och jämför det numeriska värdet med den andra operanden 1 för att returnera värdet `false` (värdet 1 är inte mindre än 1).

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

Du kan ändra standardvärdet för vänster-associativitet med parentesoperatorn. Du kan instruera kompilatorn att behandla operatorn mindre än först genom att sätta denna operator och dess operander inom parentes. I följande exempel används parentesoperatorn för att skapa andra utdata med samma tal som i föregående exempel.

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

Operatorn för mindre än behandlas först, vilket ger värdet `false` eftersom operanden 2 inte är mindre än operanden 1. Värdet `false` skickas sedan till operatorn för större än tillsammans med operanden 3. Följande kod anger det här mellanläget:

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

Operatorn större än konverterar värdet `false` till det numeriska värdet 0 och jämför det numeriska värdet med den andra operanden 3 för att returnera värdet `true` (värdet 3 är större än 0).

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

I följande tabell visas operatorerna för ActionScript 3.0 i minskande prioritetsordning. Varje rad i tabellen innehåller operatorer med samma prioritet. En rad med operatorer i tabellen har högre prioritet än raden under.

Grupp	**Operatorer**
Primär	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Postfix	`x++ x--`
Unär	`++x --x + - ~ ! delete typeof void`
Multiplicerande	`* / %`
Adderande	`+ -`
Bitvisa skift	`<< >> >>>`
Relation	`< > <= >= as in instanceof is`
Likhet	`== != === !==`
Bitvis AND	`&`
Bitvis XOR	`^`
Bitvis OR	`|`
Logiskt AND	`&&`
Logiskt OR	`||`
Villkorligt	`?:`
Tilldelning	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Komma	`,`

Primära operatorer

Primära operatorer används för att skapa array- och objektlitteraler, gruppera uttryck, anropa funktioner, instansiera klassinstanser och komma åt egenskaper.

Alla primära operatorer som visas i följande tabell har samma prioritet. Operatorer som är en del av E4X-specifikationen anges med (E4X)-notationen.

Operator	Utförd operation
`[]`	Initierar en array
`{x:y}`	Initierar ett objekt
`()`	Grupperar uttryck
`f(x)`	Anropar en funktion
`new`	Anropar en konstruktor
`x.y x[y]`	Kommer åt en egenskap
`<></>`	Initierar ett XMLList-objekt (E4X)
`@`	Kommer åt ett attribut (E4X)
`::`	Kvalificerar ett namn (E4X)
`..`	Kommer åt en underordnat XML-element (E4X)

Postfix-operatorer

Postfix-operatorer använder en operator och antingen ökar eller minskar värdet. Även om dessa operatorer är unära operatorer klassificeras de avgränsat från resten av de unära operatorerna genom deras högre prioritet och speciella beteende. När du använder en postfix-operator som en del i ett större uttryck, returneras uttryckets värde innan postfix-operatorn behandlas. Följande kod visar hur värdet för uttrycket `xNum++` returneras innan värdet ökas:

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

Alla postfix-operatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`++`	Ökningar (postfix)
`--`	Minskningar (postfix)

Unära operatorer

Unära operatorer använder en operand. Operatorer för ökning (`++`) och minskning (`--`) i denna grupp är *prefix-operatorer*, vilket betyder att de förekommer före operanden i ett uttryck. Prefix-operatorer skiljer sig från postfix-motsvarigheten eftersom öknings- och minskningsoperationen utförs innan värdet för det övergripande uttrycket returneras. I följande kod visas hur värdet för uttrycket `++xNum` returneras efter att värdet ökats.

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

Alla unära operatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`++`	Ökningar (prefix)
`--`	Minskningar (prefix)
`+`	Unär +
`-`	Unär – (negation)
`!`	Logiskt NOT
`~`	Bitvis NOT
`delete`	Tar bort en egenskap
`typeof`	Returnerar typinformation
`void`	Returnerar odefinierat värde

Multiplicerande operatorer

Multiplicerande operatorer tar två operander och utför multiplikation, division eller modulo-beräkningar.

Alla multiplicerande operatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`*`	Multiplikation
`/`	Division
`%`	Modulo

Adderande operatorer

Adderande operatorer tar två operander och utför addition eller subtraktion. Alla adderande operatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`+`	Addition
`-`	Subtraktion

Bitvisa skiftoperatorer

De bitvisa skiftoperatorerna tar två operatorer och skiftar bitarna i den första operanden i den utsträckning som anges av den andra operanden. Alla bitvisa skiftoperatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`<<`	Bitvis vänsterskift
`>>`	Bitvis högerskift
`>>>`	Bitvis osignerad högerskift

Relationsoperatorer

Relationsoperatorer tar två operander, jämför deras värden och returnerar ett booleskt värde. Alla relationsoperatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`<`	Mindre än
`>`	Större än
`<=`	Mindre än eller lika med
`>=`	Större än eller lika med
`as`	Kontrollerar datatyp
`in`	Kontrollerar objektegenskaper
`instanceof`	Kontrollerar prototypkedja
`is`	Kontrollerar datatyp

Likhetsoperatorer

Likhetsoperatorer tar två operander, jämför deras värden, och returnerar ett booleskt värde. Alla likhetsoperatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`==`	Likhet
`!=`	Olikhet
`===`	Strikt likhet
`!==`	Strikt olikhet

Bitvisa logiska operatorer

Bitvisa logiska operatorer tar två operander och utför logiska operationer på bitnivå. De bitvisa logiska operatorerna har inte samma prioritet och anges i tabellen efter minskande prioritet:

Operator	Utförd operation
`&`	Bitvis AND
`^`	Bitvis XOR
`|`	Bitvis OR

Logiska operatorer

Logiska operatorer tar två operander och returnerar ett booleskt värde. De logiska operatorerna har inte samma prioritet och anges i tabellen efter minskande prioritet:

Operator	Utförd operation
`&&`	Logiskt AND
`||`	Logiskt OR

Villkorsoperator

Villkorsoperatorn är en ternär operator vilket innebär att den tar tre operander. Villkorsoperatorn är en snabbare metod att använda den villkorliga programsatsen `if..else` .

Operator	Utförd operation
`?:`	Villkorligt

Tilldelningsoperatorer

Tilldelningsoperatorer tar två operander och tilldelar en operand ett värde baserat på värdet för den andra operanden. Alla tilldelningsoperatorer i följande tabell har samma prioritet:

Operator	Utförd operation
`=`	Tilldelning
`*=`	Multiplikationstilldelning
`/=`	Divisionstilldelning
`%=`	Modulotilldelning
`+=`	Additionstilldelning
`-=`	Subtraktionstilldelning
`<<=`	Bitvis vänsterskifttilldelning
`>>=`	Bitvis högerskifttilldelning
`>>>=`	Bitvis osignerad högerskifttilldelning
`&=`	Bitvis AND-tilldelning
`^=`	Bitvis XOR-tilldelning
`|=`	Bitvis OR-tilldelning

Villkorliga satser

ActionScript 3.0 innehåller tre grundläggande villkorliga satser som du använder för att styra programflödet.

if..else

Med den villkorliga programsatsen `if..else` kan du testa ett villkor och sedan köra ett kodblock om villkoret är uppfyllt eller köra ett annat kodblock om villkoret inte är uppfyllt. Exempel: I följande kod testas om värdet för `x` överstiger 20, skapas en `trace()`-funktion om så är fallet eller skapas en annan `trace()`-funktion om så inte är fallet:

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

Om du inte vill köra ett alternativt kodblock kan du använda programsatsen `if` utan programsatsen `else`.

if..else if

Du kan testa mer än ett villkor med hjälp av den villkorliga programsatsen `if..else if`. I följande kod testas inte enbart om värdet för `x` överstiger 20 utan även om värdet för `x` är negativt:

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

Om en `if`- eller `else`-programsats följs av endast en programsats behöver den inte sättas inom klammerparentes. I koden nedan används till exempel inga klammerparenteser:

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

Du bör emellertid alltid använda klammerparenteser, eftersom ett oväntat beteende kan uppstå om programsatser senare läggs till i en villkorlig programsats som saknar klammerparenteser. I följande kod ökas värdet för `positiveNums` med 1 vare sig villkoret utvärderas till `true` eller inte:

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

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

trace(positiveNums); // 1
```

switch

Använd programsatsen `switch` om du har flera körningssökvägar som är beroende av samma villkorliga uttryck. Programsatsen fungerar på ungefär samma sätt som en lång serie av `if..else if`-element, men den är lite lättare att läsa. I stället för att testa ett villkor för ett booleskt värde utvärderar programsatsen `switch` ett uttryck och använder resultatet för att bestämma vilket kodblock som ska köras. Kodblock börjar med en `case`-programsats och slutar med en `break`-programsats. Exempel: Följande `switch`-programsats skriver ut veckodag baserat på dagnumret som returnerats av metoden `Date.getDay()`:

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

Looping

Med slingprogramsatser kör du ett speciellt kodblock upprepade gånger med hjälp av en serie värden eller variabler. Du bör alltid omsluta kodblocket med klammerparenteser ({}). Även om du kan utesluta klammerparenteser om kodblocket bara innehåller en programsats bör du inte göra det av samma orsak som du inte gör det i villkorliga programsatser: det ökar risken för att programsatser som läggs till senare av misstag utesluts från kodblocket. Om du senare lägger till en programsats i kodblocket, men glömmer att lägga till de nödvändiga klammerparenteserna, körs programsatsen inte som en del av slingan.

for

Med slingan `for` kan du iterera genom en variabel efter ett visst värdeintervall. Du måste ange tre uttryck för en `for`-programsats: en variabel som är inställd på ett initialvärde, en villkorlig programsats som bestämmer när repeteringen avslutas och ett uttryck som ändrar värdet för variabeln med varje repetition. Följande kod repeteras fem gånger. Värdet för variabeln `i` startar vid 0 och slutar vid 4, och utdata blir siffrorna 0 till 4 på fyra olika rader.

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

for..in

Slingan `for..in` itereras från ett objekts egenskaper eller en arrays element. Du kan till exempel använda en `for..in`-slinga för att iterera genom egenskaperna för ett generiskt objekt (objektegenskaper ligger inte i någon särskilt ordning, vilket gör att de visas i slumpvis ordning):

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

Du kan även iterera igenom elementen i en array:

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

Du kan dock inte iterera genom egenskaperna för ett objekt om det är en instans av en fast klass (inklusive inbyggda klasser och användardefinierade klasser). Du kan bara iterera genom egenskaperna för en dynamisk klass. Och även för instanser av dynamiska klasser kan du bara iterera genom egenskaper som läggs till dynamiskt.

for each..in

Slingan `for each..in` itererar genom en samlings objekt som kan vara taggar i ett XML- eller XMLList-objekt, värden för objektegenskaper eller element i en array. Som visas i följande utdrag kan du använda en `for each..in`-slinga för att iterera genom egenskaper för ett generiskt objekt, men i motsats till slingan `for..in` innehåller iteratorvariabeln i en `for each..in`-slinga värdet för egenskapen i stället för namnet på egenskapen:

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

Du kan iterera genom ett XML- eller XMLList-objekt, vilket visas i följande exempel:

```
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
*/
```

Du kan även iterera genom elementen i en array, vilket visas i följande exempel:

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

Du kan inte iterera genom egenskaperna för ett objekt om objektet är en instans av en fast klass. Inte ens för instanser av dynamiska klasser kan du iterera genom fasta egenskaper som har definierats som en del av klassdefinitionen.

while

Slingan `while` är som en `if`-programsats som upprepas så länge som villkoret är `true`. Följande kod skapar samma utdata som exemplet med slingan `for`:

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

En nackdel med att använda en `while`-slinga i stället för en `for`-slinga är att det är lättare att råka skriva oändliga slingor med `while`-slingor. Exempelkoden för `for`-slingan kompileras inte om du utelämnar uttrycket som ökar räknarvariabeln, men exemplet med `while`-slingan kompilerar om du utelämnar det steget. Utan uttrycket som ökar `i` blir slingan oändlig.

do..while

Slingan `do..while` är en `while`-slinga som garanterar att kodblocket körs minst en gång, eftersom villkoret kontrolleras när kodblocket har körts. Följande kod visar ett enkelt exempel på en `do..while`-slinga som ger utdata även om villkoret inte uppfylls:

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

Funktioner

Funktioner är kodblock som utför vissa uppgifter och som kan återanvändas i programmet. Det finns två typer av funktioner i ActionScript 3.0: *metoder* och *funktionsslut*. Huruvida en funktion kallas metod eller funktionsslut beror på det sammanhang som funktionen definierats i. En funktion kallas metod om du definierar den som en del av en klassdefinition eller bifogar den till en instans av ett objekt. En funktion kallas funktionsslut om den har definierats på något annat sätt.

Funktioner har alltid varit oerhört viktiga i ActionScript. I ActionScript 1.0 fanns inte nyckelordet `class`, och ”klasser” definierades av konstruktorfunktioner. Även om nyckelordet `class` sedan dess har lagts till i språket är det fortfarande viktigt att du känner till hur funktioner fungerar om du vill få ut det mesta av språket. Detta kan vara en utmaning för programmerare som förväntar sig att ActionScript-funktioner ska bete sig på samma sätt som funktioner i C++ eller Java. Även om grundläggande funktionsdefinition och funktionsanrop inte är någon utmaning för erfarna programmerare, kräver ändå vissa av de avancerade funktionerna i ActionScript en viss förklaring.

Grundläggande funktionsbegrepp

Anropsfunktioner

Du kan anropa en funktion med dess identifierare följt av parentesoperatorn (`()`). Använd parentesoperatorn för att omsluta de funktionsparametrar som ska skickas till funktionen. Funktionen `trace()` är till exempel en funktion på översta nivån i ActionScript 3.0:

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

Om du anropar en funktion som saknar parametrar måste du använda en tom parentes. Du kan till exempel använda metoden `Math.random()`, som inte tar någon parameter, för att generera ett slumpmässigt tal:

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

Definiera egna funktioner

Du kan definiera en funktion på två sätt i ActionScript 3.0: du kan använda en funktionssats eller ett funktionsuttryck. Vilken teknik du använder beror på om du vill ha en statisk eller dynamisk programmeringsstil. Om du föredrar statisk programmering, s.k. strikt läge, definierar du funktioner med funktionssatser. Om du har ett särskilt behov av att använda funktionsuttryck definierar du funktioner med dessa. Funktionsuttryck används oftare i dynamisk programmering, s.k. standardläge.

Funktionssatser

Funktionssatser används mest vid definition av funktioner i strikt läge. En funktionssats börjar med nyckelordet `function` följt av:

- Funktionens namn
- De parametrar i en kommaavgränsad lista som omsluts av parentes

- Funktionstexten, d.v.s. den ActionScript-kod som ska köras när funktionen anropas, omsluten av klammerparenteser

Följande kod skapar en funktion som definierar en parameter och sedan anropar funktionen med strängen ”`hello`” som parametervärde:

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

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

Funktionsuttryck

Det andra sättet att deklarera en funktion är att använda en tilldelningsprogramsats med ett funktionsuttryck, vilken ibland kallas funktionslitteral eller anonym funktion. Detta är en mer detaljerad metod som används mycket i tidigare versioner av ActionScript.

En tilldelningssats med ett funktionsuttryck börjar med nyckelordet `var` följt av:

- Funktionens namn
- Kolonoperatorn (`:`)
- Klassen `Function` som anger datatypen
- Tilldelningsoperatorn (`=`)
- Nyckelordet `function`
- De parametrar i en kommaavgränsad lista som omsluts av parentes
- Funktionstexten, d.v.s. den ActionScript-kod som ska köras när funktionen anropas, omsluten av klammerparenteser

  I följande kod deklareras funktionen `traceParameter` med ett funktionsuttryck:

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

  Observera att du inte anger något funktionsnamn, vilket du gör i en funktionssats. En annan viktig skillnad mellan funktionsuttryck och funktionssatser är att ett funktionsuttryck är ett uttryck och inte en programsats. Detta betyder att ett funktionsuttryck inte kan stå för sig självt, vilket en funktionssats kan. Ett funktionsuttryck kan användas som en del av en sats, vanligen en tilldelningssats. I följande exempel visas ett array-element som tilldelats ett funktionsuttryck:

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

Välj mellan programsatser och uttryck

Som allmän regel gäller att du använder en funktionssats om du inte särskilt måste använda ett uttryck. Funktionssatser är mindre detaljerade och de är mera konsekventa vad gäller strikt läge och standardläge än funktionsuttryck.

Funktionssatser är lättare att läsa än tilldelningssatser som innehåller funktionsuttryck. Funktionssatser gör koden mera koncis. De är mindre krångliga än funktionsuttryck, som kräver att du använder både nyckelordet `var` och `function`.

Funktionssatser är mera konsekventa vad beträffar de två kompilatorlägena, eftersom du kan använda punktsyntax i både strikt läge och standardläge för att anropa en metod som deklarerats med en funktionssats. Detta är inte alltid säkert när det gäller metoder som deklarerats med ett funktionsuttryck. I följande kod definieras klassen Example med två metoder: `methodExpression()` som deklareras med ett funktionsuttryck samt `methodStatement()` som deklareras med en funktionssats. I strikt läge kan du inte använda punktsyntax för att anropa metoden `methodExpression()`.

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

Funktionsuttryck passar bättre för programmering som fokuseras på körningsbeteende, s.k. dynamiskt beteende. Om du föredrar strikt läge men ändå måste anropa en metod som deklarerats med ett funktionsuttryck, kan du använda vilken teknik du vill. Du kan anropa metoden med hjälp av hakparentes (`[]`) i stället för punktoperatorn (`.`). Följande metodanrop lyckas både i strikt läge och i standardläge:

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

Du kan också deklarera hela klassen som en dynamisk klass. Även om detta gör att du kan anropa metoden med punktoperatorn, är nackdelen att du offrar vissa funktioner i strikt läge för alla instanser av den här klassen. Kompilatorn genererar inte något fel om du försöker komma åt en odefinierad egenskap för en instans av en dynamisk klass.

I vissa lägen är funktionsuttryck mycket användbara. Ofta används funktionsuttryck för funktioner som används bara en gång och sedan tas bort. Ett ovanligare användning är att bifoga en funktion till en prototypegenskap. Mer information finns i Objektet prototype.

Det finns två små skillnader mellan funktionssatser och funktionsuttryck som du måste tänka på när du väljer vilken teknik du ska använda. Den första skillnaden är att funktionsuttryck inte existerar oberoende som objekt med hänsyn till minneshantering och skräpsamling. När du tilldelar ett annat objekt, t.ex. ett array-element eller en objektegenskap, ett funktionsuttryck skapar du bara en referens till funktionsuttrycket i koden. Om den array eller det objekt som funktionsuttrycket är kopplat till hamnar utanför omfånget eller på annat sätt blir otillgängligt, kan du inte komma åt funktionsuttrycket. Om arrayen eller objektet tas bort blir det minne som funktionsuttrycket använder godkänt för skräpsamling, vilket betyder att minnet kan återvinnas och återanvändas för andra ändamål.

I följande exempel visas, att för ett funktionsuttryck är funktionen inte längre tillgänglig när egenskapen som uttrycket är kopplat till tas bort. Klassen Test är dynamisk, vilket betyder att du kan lägga till egenskapen `functionExp` som innehåller ett funktionsuttryck. Funktionen `functionExp()` kan anropas med punktoperatorn, men när egenskapen `functionExp` tas bort är funktionen inte längre tillgänglig.

```
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
```

Om å andra sidan funktionen först definieras med en funktionssats existerar den som ett eget objekt och fortsätter att existera även när du tar bort egenskapen som funktionen är kopplad till. Operatorn `delete` fungerar bara på egenskaper för objekt, och inte ens ett anrop för att ta bort själva funktionen `stateFunc()` fungerar.

```
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
```

Den andra skillnaden mellan funktionssatser och funktionsuttryck är att funktionssatser existerar i hela omfånget som de definierats i, däribland programsatser som visas före funktionssatsen. Funktionsuttryck definieras däremot bara för efterföljande programsatser. I följande kod anropas funktionen `scopeTest()` innan den definieras:

```
statementTest(); // statementTest

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

Funktionsuttryck är inte tillgängliga innan de definieras, och följande kod genererar ett körningsfel:

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

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

Returnera värden från funktioner

Om du vill returnera ett värde från funktionen använder du programsatsen `return` följt av det uttryck eller litterala värde du vill returnera. Följande kod returnerar ett uttryck som representerar parametern:

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

Observera att programsatsen `return` avbryter funktionen, vilket innebär att programsatser under en `return`-programsats inte körs, enligt följande:

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

I strikt läge måste du returnera ett värde för en lämplig typ om du väljer att ange en returtyp. I följande kod genereras ett fel i strikt läge eftersom ett giltigt värde inte returneras:

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

Kapslade funktioner

Du kan kapsla funktioner, vilket betyder att funktioner kan deklareras inuti andra funktioner. En kapslad funktion är bara tillgänglig inuti dess överordnade funktion, såvida inte en referens till funktionen skickas till extern kod. I följande kod deklareras två kapslade funktioner inuti funktionen `getNameAndVersion()`:

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

När kapslade funktioner skickas till extern kod skickas de som funktionsslut, vilket betyder att funktionen sparar alla definitioner som finns i omfånget när funktionen definieras. Mer information finns i Funktionsomfång.

Funktionsparametrar

I ActionScript 3.0 finns vissa funktioner för funktionsparametrar som kan verka vara nyheter för programmerare som är nybörjare i språket. Även om begreppet att skicka parametrar med värde eller referens är känt för de flesta programmerare, är kanske `argument`objektet och... (rest)-parametern ny för många.

Överföra argument med värde eller referens

I många programmeringsspråk är det viktigt att förstå skillnaden mellan att skicka argument med värde eller med referens. Skillnaden kan påverka sättet som koden utformas på.

Att skickas med värde betyder att värdet för argumentet kopieras till en lokal variabel som ska användas i funktionen. Att skickas med referens betyder att bara en referens till argumentet skickas, i stället för det verkliga värdet. Ingen kopia av argumentet görs. I stället skapas en referens till den variabel som skickas som ett argument och den kopplas till en lokal variabel som ska användas i funktionen. Med den lokala variabeln, som är en referens till en variabel utanför funktionen, kan du ändra värdet på ursprungsvariabeln.

I ActionScript 3.0 skickas alla argument med referens, eftersom alla värden lagras som objekt. Objekt som tillhör primitiva datatyper, däribland Boolean, Number, int, uint och String, har speciella operatorer som gör att de beter sig som om de skickats med värde. Med följande kod skapas till exempel funktionen `passPrimitives()` som definierar två parametrar, `xParam` och `yParam`, som båda är av typen int. Dessa parametrar liknar lokala variabler som deklarerats i texten för funktionen `passPrimitives()`. När funktionen anropas med argumenten `xValue` och `yValue` initieras parametrarna `xParam` och `yParam` med referenser till int-objekt som representeras av `xValue` och `yValue`. Eftersom argumenten är primitiva beter de sig som om de skickats med värde. Även om `xParam` och `yParam` initialt bara innehåller referenser till objekten `xValue` och `yValue`, genererar alla ändringar i variablerna i funktionstexten nya kopior av värdena i minnet.

```
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
```

I funktionen `passPrimitives()` ökas värdena för `xParam` och `yParam`, men detta påverkar inte värdena för `xValue` och `yValue`, vilket visas i den senaste `trace`-programsatsen. Detta skulle vara fallet även om parametrarna hade exakt samma namn som variablerna, `xValue` och `yValue`, eftersom `xValue` och `yValue` inuti funktionen pekar på nya platser i minnet som ligger åtskilda från variablerna med samma namn utanför funktionen.

Alla andra objekt, d.v.s. objekt som inte hör till de primitiva datatyperna, skickas alltid med referens, vilket gör att du kan ändra värdet på ursprungsvariabeln. I följande kod skapas objektet `objVar` med de två egenskaperna `x` och `y`. Objektet skickas som ett argument till funktionen `passByRef()`. Eftersom objektet inte är av primitiv typ skickas det med referens och stannar som referens. Detta betyder att ändringar av parametrarna i funktionen påverkar objektets egenskaper utanför funktionen.

```
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
```

Parametern `objParam` refererar samma objekt som den globala variabeln `objVar`. Som du kan se av programsatsen `trace` i exemplet återspeglas ändringar i egenskaperna `x` och `y` för objektet `objParam` i objektet `objVar`.

Standardvärden för parametrar

I ActionScript 3.0 kan du deklarera *standardparametervärden* för en funktion. Om ett anrop till en funktion med standardvärden för parametrar utesluter en parameter med standardvärden, används värdet som anges i funktionsdefinitionen för denna parameter. Alla parametrar med standardvärden måste placeras i slutet av parameterlistan. De värden som angetts som standardvärden måste vara kompileringskonstanter. Om det finns ett standardvärde för en parameter blir denna parameter en *valfri parameter*. En parameter utan standardvärde betraktas som en *obligatorisk parameter*.

I följande kod skapas en funktion med tre parametrar, varav två har standardvärden. När funktionen anropas med bara en parameter, används standardvärdena för denna parameter.

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

Objektet Arguments

När parametrar skickas till en funktion, kan du använda objektet `arguments` för att komma åt information om de parametrar som skickats till funktionen. Vissa aspekter om `argument` är viktiga:

- Objektet `arguments` är en array som innehåller alla parametrar som skickats till funktionen.
- Egenskapen `arguments.length` rapporterar antalet parametrar som skickats till funktionen.
- Egenskapen `arguments.callee` ger en referens till själva funktionen, och den är användbar för rekursiva anrop till funktionsuttryck.

  Obs! *Objektet `arguments` är inte tillgängligt om någon parameter heter `arguments` eller om du använder parametern ... (rest).*

  Om funktionstexten refererar till `arguments`-objektet kan funktionsanrop i ActionScript 3.0 innehålla fler parametrar än de som definierats i funktionsdefinitionen, men ett kompileringsfel genereras i strikt läge om antalet parametrar inte matchar antalet obligatoriska parametrar (och eventuellt valfria parametrar). Du kan använda arrayaspekten för objektet `arguments` för att komma åt alla parametrar som skickats till funktionen, vare sig de är definierade i funktionsdefinitionen eller inte. I följande exempel, som bara kompileras i standardläge, används arrayen `arguments` tillsammans med egenskapen `arguments.length` för att kalkera alla parametrar som skickats till funktionen `traceArgArray()`:

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

  traceArgArray(1, 2, 3);

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

  Egenskapen `arguments.callee` används ofta i anonyma funktioner för att skapa rekursion. Du kan använda den för att göra koden flexiblare. Om namnet på en rekursiv funktion ändras under utvecklingen måste du kanske ändra det rekursiva anropet i funktionstexten om du använder `arguments.callee` i stället för funktionsnamnet. Egenskapen `arguments.callee` används i följande funktionsuttryck för att skapa rekursion:

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

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

Om du använder parametern ... (rest) i funktionsdeklarationen är objektet `arguments` inte tillgängligt. Du måste i stället komma åt parametrarna med hjälp av de parameternamn du deklarerat för dem.

Du bör inte använda strängen `"arguments"` som parameternamn, eftersom den skuggar objektet `arguments`. Om till exempel funktionen `traceArgArray()` skrivs om så att parametern `arguments` läggs till, pekar referenserna på `arguments` i funktionstexten på parametern och inte på objektet `arguments`. Följande kod ger inga utdata:

```
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
```

Objektet `arguments` i tidigare versioner av ActionScript innehåller också egenskapen `caller` som är en referens till den funktion som anropas i aktuell funktion. Egenskapen `caller` finns inte i ActionScript 3.0, men om du behöver en referens till anropsfunktionen, kan du ändra den så att den skickar en extra parameter som är en referens till sig själv.

Parametern... (rest)

I ActionScript 3.0 introduceras en ny parameterdeklaration som kallas parametern... (rest). Med den här parametern kan du ange en arrayparameter som godkänner valfritt antal kommaavgränsade parametrar. Parametern kan heta vad som helst bara det inte är ett reserverat ord. Parameterdeklarationen måste vara den sist angivna parametern. Om du använder den här parametern kan du inte använda objektet `arguments`. Även om parametern ... (rest) har samma funktioner som arrayen `arguments` och egenskapen `arguments.length` har den inte samma funktioner som `arguments.callee`. Du måste kontrollera att du inte behöver använda `argument.callee` innan du använder parametern ... (rest).

I följande exempel skrivs funktionen `traceArgArray()` om med hjälp av ... (rest)-parametern i stället för objektet `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
```

Parametern... (rest) kan också användas med andra parametrar så länge som den är den sista parametern. I följande exempel ändras funktionen `traceArgArray()` så att dess första parameter, `x`, är av typen int och den andra parametern använder parametern ... (rest). Det första värdet finns inte med i utdata eftersom den första parametern inte längre ingår i arrayen som skapats av parametern ... (rest).

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

traceArgArray(1, 2, 3);

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

Funktioner som objekt

Funktioner i ActionScript 3.0 är objekt. När du skapar en funktion skapar du ett objekt, som inte bara kan skickas som parameter till en annan funktion, utan även få egenskaper och metoder kopplade till sig.

Funktioner som skickas som argument till andra funktioner skickas med referens och inte med värde. När du skickar en funktion som ett argument använder du bara identifieraren och inte parentesoperatorn som du använder för att anropa metoden. Med följande kod skickas funktionen `clickListener()` som ett argument till metoden `addEventListener()`:

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

Även om du som inte använt ActionScript tidigare tycker att det är märkligt, kan funktioner ha egenskaper och metoder som vilka andra objekt som helst. Alla funktioner har i själva verket en skrivskyddad egenskap som heter `length` som lagrar antalet parametrar som definierats för funktionen. Egenskapen `arguments.length` rapporterar i stället antalet argument som skickas till funktionen. Kom ihåg, att i ActionScript kan antalet argument som skickas till en funktion överstiga antalet parametrar som definierats för samma funktion. I följande exempel, som bara kompileras i standardläge eftersom strikt läge kräver en exakt överensstämmelse mellan antalet skickade argument och antalet definierade parametrar, visas skillnaden mellan de två egenskaperna:

```
// 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 */
```

I standardläget kan du definiera egna funktionsegenskaper genom att definiera dem utanför funktionstexten. Funktionsegenskaperna kan fungera som kvasistatiska egenskaper som du använder för att spara en variabel som relateras till funktionen. Du kanske vill spåra hur många gånger en viss funktion anropas. Du kan använda den här funktionen om du skriver ett spel och vill spåra hur många gånger en användare använder ett visst kommando, men du kan också använda en statisk klassegenskap för detta. I följande exempel, som bara kompileras i standardläge eftersom du inte kan lägga till dynamiska egenskaper i funktioner i strikt läge, skapas en funktionsegenskap utanför funktionsdeklarationen och egenskapen ökar stegvis varje gång som funktionen anropas:

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

someFunction.counter = 0;

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

Funktionsomfång

Med en funktions omfång anges var i programmet funktionen kan anropas men också vilka funktionsdefinitioner som funktioner har åtkomst till. Samma omfångsregler som gäller för variabelidentifierare gäller också för funktionsidentifierare. En funktion som deklareras i det globala omfånget kan användas i hela koden. ActionScript 3.0 innehåller globala funktioner, till exempel `isNaN()` och `parseInt()`, som är tillgängliga överallt i koden. En kapslad funktion, d.v.s. en funktion som har deklarerats inuti en annan funktion, kan användas var som helst i den funktion som den har deklarerats i.

Omfångskedjan

Varje gång en funktion körs skapas ett antal objekt och egenskaper. Först skapas ett speciellt objekt som kallas *aktiveringsobjekt* som lagrar parametern och alla lokala variabler eller funktioner som deklarerats i funktionstexten. Du kan inte komma åt aktiveringsobjektet direkt eftersom det är en intern mekanism. Därefter skapas en *omfångskedja*, som innehåller en ordnad lista på objekt som vid körning genomsöks efter identifierardeklarationer. Varje funktion som körs har en omfångskedja som lagras i en intern egenskap. Om det är en kapslad funktion startar omfångskedjan med det egna aktiveringsobjektet, följt av aktiveringsobjektet i den överordnade funktionen. Kedjan fortsätter på det här sättet tills den når det globala objektet. Det globala objektet skapas när ett ActionScript-program startar och objektet innehåller alla globala variabler och funktioner.

Funktionsslut

Ett *funktionsslut* är ett objekt som innehåller en ögonblicksbild av en funktion och dess *lexikala miljö*. En funktions lexikala miljö innehåller alla variabler, egenskaper, metoder och objekt i funktionens omfångskedja, tillsammans med deras värden. Funktionsslut skapas varje gång en funktion körs skilt från ett objekt eller en klass. Det faktum att funktionsslutet sparar omfånget som det definierats i, skapar intressanta resultat när en funktion skickas som ett argument eller ett returvärde till ett annat omfång.

I följande kod skapas två olika funktioner: `foo()`, som returnerar den kapslade funktionen `rectArea()` som beräknar arean av en rektangel, samt `bar()` som anropar `foo()` och lagrar det returnerade funktionsslutet i variabeln `myProduct`. Även om funktionen `bar()` definierar sin egen lokala variabel `x` (med värdet 2) när funktionsslutet `myProduct()` anropas, sparas variabeln `x` (med värdet 40) som definierats i funktionen `foo()`. Funktionen `bar()` returnerar därför värdet `160` i stället för `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
```

Metoder beter sig på ungefär samma sätt eftersom de också sparar information om den lexikala miljö som de skapats i. Dessa egenskaper märks mest när en metod extraheras från sin instans, vilket skapar en bunden metod. Den största skillnaden mellan ett funktionsslut och en bunden metod är att värdet för nyckelordet `this` i en bunden metod alltid refererar till instansen som den ursprungligen var kopplad till, medan värdet för nyckelordet `this` kan ändras i ett funktionsslut.

Kapitel 4: Objektorienterad programmering i ActionScript

Objektorienterad programmering

Objektorienterad programmering (OOP) är ett sätt att ordna koden i ett program genom att gruppera den i objekt. Termen *objekt* betyder i det här sammanhanget ett enskilt element, som innehåller information (datavärden) och funktioner. Vid objektorienterad programmering grupperar du information tillsammans med funktioner eller åtgärder, som är kopplade till informationen i fråga. Du kan till exempel gruppera musikinformation, som album-, spår- eller artistnamn, med funktioner som ”lägg till spår i spellista” eller ”spela upp alla spår med den här artisten”. De här delarna kombineras till ett enda objekt (till exempel ett ”album” eller ”musikspår”). Det finns flera fördelar med att sammankoppla värden och funktioner. En viktig fördel är att du bara behöver använda en enda variabel, i stället för flera. Dessutom håller det ihop relaterade funktioner. Slutligen kan du genom att kombinera information och funktioner strukturera program på sätt som bättre speglar verkligheten.

Klasser

En klass är en abstrakt beteckning för ett objekt. I en klass lagras information om de typer av data som ett objekt kan innehålla och det beteende som objektet kan ärva. Fördelen med en sådan abstraktion kanske inte syns när du skriver små skript, som bara innehåller några få objekt som samverkar med varandra. Men efterhand som ett program utvecklas och växer ökar också antalet objekt som måste hanteras. I sådana fall blir det enklare att med hjälp av klasser styra hur objekt skapas och samverkar med varandra.

Så långt tillbaka som i ActionScript 1.0 kunde programmerarna använda Function-objekt för att skapa konstruktioner som liknade klasser. ActionScript 2.0 hade formellt stöd för klasser med nyckelord som `class` och `extends`. ActionScript 3.0 har fortsatt stöd för de nyckelord som introducerades i ActionScript 2.0, men innehåller också nya funktioner. ActionScript 3.0 innehåller till exempel förbättrad åtkomstkontroll via attributen `protected` och `internal`. Det ger också bättre kontroll över arv via nyckelorden `final` och `override`.

För utvecklare som har skapat klasser i programmeringsspråk som Java, C++ eller C# kommer ActionScript att kännas bekant. ActionScript delar många nyckelord och attributnamn, som `class`, `extends` och `public`, med de språken.

Obs! *I dokumentationen för Adobe ActionScript avser termen ”egenskap” alla medlemmar av ett objekt eller en klass, däribland variabler, konstanter och metoder. Termerna klass och statisk kan ofta användas som synonymer, men här är de olika. Termen klassegenskaper avser till exempel alla medlemmar i en klass, inte bara de statiska medlemmarna.*

Klassdefinitioner

I ActionScript 3.0 används en syntax för klassdefinitioner som liknar den som används för klassdefinitioner i ActionScript 2.0. Korrekt syntax för en klassdefinition anropar nyckelordet `class` följt av klassnamnet. Klassbrödtexten, som omsluts av klammerparenteser ({}), följer efter klassnamnet. I följande kod skapas klassen Shape som innehåller en variabel som heter `visible`:

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

En viktig syntaxändring gäller klassdefinitioner som finns i ett paket. Om en klass finns i ett paket måste paketnamnet i ActionScript 2.0 inkluderas i klassdeklarationen. I ActionScript 3.0, som innehåller programsatsen `package`, måste paketnamnet inkluderas i paketdeklarationen i stället för i klassdeklarationen. I följande klassdeklarationer visas hur klassen BitmapData, som är en del av paketet flash.display, definieras i ActionScript 2.0 och i ActionScript 3.0:

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

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

Klassattribut

I ActionScript 3.0 kan du ändra klassdefinitionerna med något av dessa fyra attribut:

Attribut	Definition
`dynamic`	Egenskaper kan läggas till i förekomster vid körningen.
`final`	Får inte utökas av en annan klass.
`internal` (standard)	Synligt för referenser inuti aktuellt paket.
`public`	Synligt för referenser överallt.

För alla dessa attribut, med undantag av `internal`, inkluderar du attributet explicit för att få det associerade beteendet. Om du till exempel inte inkluderar attributet `dynamic` när du definierar en klass kan du inte lägga till egenskaper i en klassinstans vid körning. Du tilldelar ett attribut genom att placera det i början av klassdefinitionen, vilket visas i följande kod:

```
dynamic class Shape {}
```

Observera att listan inte innehåller attributet `abstract`. Abstrakta klasser stöds inte i ActionScript 3.0. Observera också att listan inte innehåller attributen `private` och `protected`. Dessa attribut är bara betydelsefulla inuti en klassdefinition och kan inte tillämpas på själva klasserna. Om du vill att en klass inte ska vara synlig utanför ett paket placerar du klassen inuti paketet och markerar klassen med attributet `internal`. Du kan också utesluta attributen `internal` och `public` och då läggs attributet `internal` automatiskt till av kompilatorn. Du kan också definiera en klass som bara är synlig inuti den källfil i vilken den har definierats. Placera klassen längst ned i källfilen, nedanför paketdefinitionens avslutande klammerparentes.

Klassbrödtext

Klassbrödtexten omsluts av klammerparenteser. Den definierar klassens variabler, konstanter och metoder. I följande exempel visas deklarationen för klassen Accessibility i ActionScript 3.0:

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

Du kan också definiera ett namnutrymme i en klassbrödtext. I följande exempel visas hur ett namnutrymme kan definieras i en klassbrödtext och användas som attribut för en metod i klassen:

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

Med ActionScript 3.0 kan du inkludera definitioner men också programsatser i en klassbrödtext. Programsatser som finns inuti en klassbrödtext, men utanför en metoddefinition, körs en gång. Den här körningen sker när klassdefinitionen påträffas första gången och det associerade klassobjektet skapas. I följande exempel visas ett anrop till den externa funktionen `hello()` och en `trace`-programsats som matar ut ett bekräftelsemeddelande när klassen har definierats:

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

I ActionScript 3.0 kan du definiera en statisk egenskap och en instansegenskap med samma namn i samma klassbrödtext. I följande kod deklareras den statiska variabeln `message` och en förekomstvariabel med samma namn:

```
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
```

Klassegenskapsattribut

Vid beskrivning av ActionScript-objektmodellen betyder termen *egenskap* allt som kan vara medlem i en klass, däribland variabler, konstanter och metoder. I Referenshandbok för ActionScript 3.0 i Adobe Flash-plattformen används termen dock i en snävare betydelse. I det sammanhanget omfattar termen egenskap bara klassmedlemmar som är variabler eller som har definierats med en get- eller set-metod. I ActionScript 3.0 finns en serie attribut som kan användas med valfri klassegenskap. I följande tabell visas den här attributuppsättningen.

Attribut	**Definition**
`internal` (standard)	Synligt för referenser inuti samma paket.
`private`	Synligt för referenser inuti samma klass.
`protected`	Synligt för referenser i samma klass och i härledda klasser.

Attribut	Definition
`public`	Synligt för referenser överallt.
`statiska`	Anger att en egenskap tillhör klassen och inte förekomster av klassen.
`UserDefinedNamespace`	Anpassat namnutrymme som definierats av användaren.

Namnutrymmesattribut för åtkomstkontroll

I ActionScript 3.0 finns det fyra speciella attribut som styr åtkomsten till egenskaper som definierats inuti en klass: `public`, `private`, `protected` och `internal`.

Med attributet `public` blir en egenskap synlig överallt i skriptet. Om du vill att en metod ska vara tillgänglig för kod utanför paketet, måste du deklarera metoden med attributet `public`. Detta gäller för alla egenskaper, vare sig de deklarerats med nyckelordet `var`, `const` eller `function`.

Med attributet `private` blir en egenskap synlig bara för anropare i egenskapens definitionsklass. Detta skiljer sig från attributet `private` i ActionScript 2.0, som ger en underklass åtkomst till en privat egenskap i en superklass. En annan viktig skillnad gäller körningsåtkomst. I ActionScript 2.0 förhindrar nyckelordet `private` åtkomst bara vid kompileringen och kan lätt kringgås vid körningen. I ActionScript 3.0 gäller detta inte längre. Egenskaper som har markerats som `privata` är otillgängliga både vid kompilering och vid körning.

I följande kod skapas den enkla klassen PrivateExample med en privat variabel och därefter testas åtkomsten till den privata variabeln från klassens utsida.

```
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.
```

I ActionScript 3.0 skapar ett försök att få åtkomst till en privat egenskap med hjälp av punktoperatorn (`myExample.privVar`) ett kompileringsfel om du använder strikt läge. I annat fall rapporteras felet vid körningen, på samma sätt som när du använder egenskapsåtkomstoperatorn (`myExample["privVar"]`).

I följande tabell sammanfattas resultatet av åtkomstförsök till en privat egenskap som hör till en fast (och inte en dynamisk) klass:

	Strikt läge	Standardläge
punktoperator (`.`)	kompileringsfel	körningsfel
hakparentesoperator (`[]`)	körningsfel	körningsfel

I klasser som deklarerats med attributet `dynamic` resulterar försök att komma åt en privat variabel inte i ett körningsfel. I stället är variabeln inte synlig, varför värdet `undefined` returneras. Ett kompileringsfel inträffar emellertid om du använder punktoperatorn i strikt läge. Följande exempel är detsamma som föregående exempel förutom att klassen PrivateExample deklarerats som en dynamisk klass:

```
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
```

Dynamiska klasser returnerar vanligtvis värdet `undefined` i stället för att generera ett fel när extern kod försöker komma åt en privat egenskap. I följande tabell visas att ett fel genereras bara när punktoperatorn används för att komma åt en privat egenskap i strikt läge:

	Strikt läge	**Standardläge**
punktoperator (`.`)	kompileringsfel	`undefined`
hakparentesoperator (`[]`)	`undefined`	`undefined`

Attributet `protected`, som är nytt för ActionScript 3.0, gör att en egenskap blir synlig för anropare i dess egen klass eller i en underklass. En skyddad egenskap är därför tillgänglig i sin egen klass eller i klasser som ligger någonstans under den i arvshierarkin. Detta gäller oavsett om underklassen är i samma paket eller i ett annat paket.

Om du är bekant med ActionScript 2.0 ser du att den här funktionen påminner om attributet `private` i ActionScript 2.0. Attributet `protected` i ActionScript 3.0 liknar även attributet `protected` i Java. Skillnaden är att Java-versionen även tillåter åtkomst till anropare inom samma paket. Använd attributet `protected` när du har en variabel eller metod som underklasserna behöver men som du vill dölja för kod som ligger utanför arvskedjan.

Attributet `internal`, som är nytt för ActionScript 3.0, gör att en egenskap blir synlig för anropare i dess eget paket. Det här är standardattributet för kod inuti ett paket och det tillämpas på alla egenskaper som inte har något av följande attribut:

- `public`
- `private`
- `protected`
- användardefinierat namnutrymme

Attributet `internal` liknar standardåtkomstkontrollen i Java, men i Java finns det inget explicit namn för den här åtkomstnivån och den kan bara uppnås om man utesluter alla andra åtkomstmodifierare. Attributet `internal` är tillgängligt i ActionScript 3.0 för att du ska kunna bekräfta att du vill att en egenskap bara ska vara synlig för anropare i egenskapens eget paket.

static, attribut

Med attributet `static`, som du använder med egenskaper som deklarerats med nyckelorden `var`, `const` eller `function`, kan du bifoga en egenskap till klassen i stället för till förekomster av klassen. Kod som ligger utanför klassen måste anropa statiska egenskaper med hjälp av klassnamnet och inte med ett förekomstnamn.

Statiska egenskaper ärvs inte av underklasser, men egenskaperna är en del av underklassens omfångskedja. Detta betyder att en statisk variabel eller metod, i brödtexten till en underklass, kan användas utan att referera till klassen i vilken den har definierats.

Användardefinierade namnutrymmesattribut

Som ett alternativ till de fördefinierade åtkomstkontrollsattributen kan du skapa ett anpassat namnutrymme och använda det som ett attribut. Du kan bara använda ett namnutrymmesattribut per definition och du kan inte använda det i kombination med något av åtkomstkontrollsattributen (`public`, `private`, `protected`, `internal`).

Variabler

Du kan deklarera variabler med något av nyckelorden `var` eller `const`. Variabler som deklarerats med nyckelordet `var` kan få sina värden ändrade flera gånger under skriptkörningen. Variabler som deklarerats med nyckelordet `const` kallas *konstanter* och kan få värden bara en gång. Ett försök att tilldela en initierad konstant ett nytt värde resulterar i ett fel.

Statiska variabler

Statiska variabler deklareras med en kombination av nyckelordet `static` och någon av programsatserna `var` eller `const`. Använd statiska variabler, som bifogas till en klass i stället för till en förekomst av en klass, för att lagra och dela information som gäller en hel objektklass. En statisk variabel är lämplig om du vill hålla reda på hur många gånger en klass instansieras eller om du vill lagra maximalt tillåtet antal klassförekomster.

I följande exempel skapas variabeln `totalCount` som ska hålla reda på antalet klassinstansieringar och konstanten `MAX_NUM` som ska lagra maximalt tillåtet antal instansieringar. Variablerna `totalCount` och `MAX_NUM` är statiska eftersom de innehåller värden som tillämpas på klassen som helhet och inte på en viss förekomst.

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

Kod som ligger utanför klassen StaticVars och dess underklasser kan referera till egenskaperna `totalCount` och `MAX_NUM` bara genom själva klassen. Följande kod fungerar till exempel:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

Du kan inte komma åt statiska variabler via en förekomst av klassen och följande kod returnerar ett fel:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Variabler som har deklarerats med både nyckelordet `static` och `const` måste initieras samtidigt som du deklarerar konstanten, på samma sätt som klassen StaticVars gör för `MAX_NUM`. Du kan inte tilldela `MAX_NUM` ett värde inuti konstruktorn eller en förekomstmetod. Följande kod genererar ett fel, eftersom det inte är ett giltigt sätt att initiera en statisk konstant:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Förekomstvariabler

Förekomstvariabler innehåller egenskaper som deklarerats med nyckelorden `var` och `const`, men utan nyckelordet `static`. Använd förekomstvariabler, som bifogas till klassförekomster i stället för till en hel klass, när du vill lagra värden som är speciella för en förekomst. Klassen Array har till exempel en förekomstegenskap som heter `length` som lagrar det antal arrayelement som en viss förekomst av klassen Array innehåller.

Förekomstvariabler, vare sig de deklarerats som `var` eller `const`, får inte åsidosättas i en underklass. Du kan emellertid använda en funktion som liknar åsidosättning av variabler genom att åsidosätta get- och set-metoder.

Metoder

En metod är en funktion som är en del av en klassdefinition. När en förekomst av klassen har skapats binds en metod till denna förekomst. I motsats till en funktion som deklarerats utanför en klass, kan en metod inte användas utanför den förekomst som den är bifogad till.

Metoder definieras med hjälp av nyckelordet `function`. Precis som med klassegenskaper kan du också använda klassegenskapsattribut på metoder, bland annat privata, skyddade, offentliga, interna, statiska och anpassade namnutrymmen. Du kan använda en funktionssats som följande:

```
public function sampleFunction():String {}
```

Du kan också använda en variabel som du kopplar ett funktionsuttryck till så här:

```
public var sampleFunction:Function = function () {}
```

I de flesta fall använder du en funktionssats i stället för ett funktionsuttryck av följande skäl:

- Funktionssatser är mer precisa och lättare att läsa.
- Med funktionssatserna kan du använda nyckelorden `override` och `final`.
- Funktionssatser skapar en starkare koppling mellan identifieraren (namnet på funktionen) och koden i metodbrödtexten. Eftersom värdet på en variabel kan ändras med en tilldelningssats kan kopplingen mellan en variabel och dess funktionsuttryck när som helst skadas. Även om du kan gå runt det här problemet genom att deklarera variabeln med `const` i stället för `var` är denna teknik inte att rekommendera. Koden blir svår att läsa och det går inte att använda nyckelorden `override` och `final`.

När du väljer att bifoga en funktion till prototypobjektet måste du använda ett funktionsuttryck.

Konstruktormetoder

Konstruktormetoder, som ibland kallas *konstruktorer*, är funktioner som har samma namn som den klass i vilken de har definierats. All kod som du inkluderar i en konstruktormetod körs när en förekomst av klassen skapas med nyckelordet `new`. I följande kod definieras en enkel klass som heter Example och som innehåller en enda egenskap som heter `status`. Det initiala värdet för variabeln `status` anges inuti konstruktorfunktionen.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Konstruktormetoder kan bara vara allmänna, men användningen av attributet `public` är valfri. Du kan inte använda någon av de andra åtkomstkontrollsspecifikationerna, däribland `private`, `protected` och `internal`, på en konstruktor. Du kan inte heller använda ett användardefinierat namnutrymme med en konstruktormetod.

En konstruktor kan anropa konstruktorn för sin direkta superklass med hjälp av programsatsen `super()`. Om superklasskonstruktorn inte explicit anropas infogar kompilatorn automatiskt ett anrop före den första programsatsen i konstruktorbrödtexten. Du kan också anropa superklassens metoder med hjälp av prefixet `super` som en referens till superklassen. Om du använder både `super()` och `super` i samma konstruktorbrödtext måste du först anropa `super()`. Annars beter referensen `super` sig på ett oväntat sätt. Konstruktorn `super()` måste också anropas före en `throw`- eller `return`-programsats.

I följande exempel visas vad som händer om du försöker använda referensen `super` innan du anropar konstruktorn `super()`. Den nya klassen ExampleEx utökar klassen Example. Konstruktorn ExampleEx försöker komma åt statusvariabeln som definierats i superklassen, men detta sker innan den anropar `super()`. Programsatsen `trace()` i konstruktorn ExampleEx skapar värdet `null` eftersom variabeln `status` inte är tillgänglig förrän konstruktorn `super()` körs.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Även om det är tillåtet att använda programsatsen `return` inuti en konstruktor är det inte tillåtet att returnera ett värde. `return`-programsatser får alltså inte ha kopplade uttryck eller värden. Konstruktormetoder kan följaktligen inte ha returvärden, vilket betyder att ingen returtyp kan anges.

Om du inte definierar en konstruktormetod i klassen skapas en tom konstruktor automatiskt. Om klassen utökar en annan klass infogas ett `super()`-anrop i den konstruktor som genereras.

Statiska metoder

Statiska metoder, som också kallas *klassmetoder*, är metoder som deklareras med nyckelordet `static`. Använd statiska metoder, som bifogas till en klass och inte till en förekomst av en klass, när du kapslar in funktioner som påverkar något annat än en individuell förekomsts läge. Eftersom statiska metoder bifogas till en klass som en helhet, kan de bara nås via en klass och inte via en förekomst av klassen.

Använd statiska metoder för att kapsla in funktioner som inte begränsas till att påverka läget för en klassförekomst. En metod måste alltså vara statisk om den tillhandahåller funktioner som inte direkt påverkar värdet för en klassförekomst. Klassen Date har till exempel en statisk metod som heter `parse()`, som tar en sträng och konverterar den till ett tal. Metoden är statisk eftersom den inte påverkar en individuell förekomst av klassen. I stället tar metoden `parse()` en sträng som representerar ett datumvärde, tolkar strängen och returnerar ett tal i ett format som är kompatibelt med den interna representationen av ett Date-objekt. Metoden är inte en förekomstmetod eftersom det inte är lämpligt att använda metoden på en förekomst av klassen Date.

Jämför metoden `parse()` med en av förekomstmetoderna för klassen Date, till exempel `getMonth()`. Metoden `getMonth()` är en förekomstmetod eftersom den arbetar direkt på värdet för en förekomst genom att hämta en speciell komponent, månaden, för en Date-förekomst.

Eftersom statiska metoder inte är bundna till individuella förekomster, kan du inte använda nyckelorden `this` och `super` i brödtexten för en statisk metod. Referenserna `this` och `super` är bara meningsfulla i samband med en förekomstmetod.

Statiska metoder i ActionScript 3.0 ärvs inte till skillnad från vissa andra klassbaserade programmeringsspråk.

Förekomstmetoder

Förekomstmetoder är metoder som deklareras utan nyckelordet `static`. Du kan använda förekomstmetoder som bifogas till förekomster av en klass i stället för till klassen som helhet för att implementera funktioner som påverkar individuella förekomster av en klass. Klassen Array innehåller till exempel en förekomstmetod som heter `sort()` som arbetar direkt på Array-förekomster.

I en förekomstmetods brödtext finns både statiska variabler och förekomstvariabler i omfång, vilket betyder att variabler som definierats i samma klass kan refereras med en enkel identifierare. Klassen CustomArray utökar till exempel klassen Array. Klassen CustomArray definierar en statisk variabel som heter `arrayCountTotal` som spårar antalet klassförekomster, en förekomstvariabel som heter `arrayNumber` som spårar den ordning i vilken förekomsterna skapats samt en förekomstmetod som heter `getPosition()` som returnerar värdena för dessa variabler.

```
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);
    }
}
```

Även om kod som ligger utanför klassen måste komma åt den statiska variabeln `arrayCountTotal` via klassobjektet med hjälp av `CustomArray.arrayCountTotal`, så kan kod som ligger i brödtexten för metoden `getPosition()` referera direkt till den statiska variabeln `arrayCountTotal`. Detta gäller även för statiska variabler i superklasser. Trots att statiska egenskaper inte ärvs i ActionScript 3.0, placeras statiska egenskaper i superklasser i omfång. Klassen Array har till exempel några få statiska variabler, varav en är en konstant som heter `DESCENDING`. Kod som finns i en Array-underklass kan komma åt den statiska konstanten `DESCENDING` med hjälp av en enkel identifierare:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

Värdet för referensen `this` i brödtexten för en förekomstmetod är en referens till förekomsten som metoden är bifogad till. I följande kod visas att referensen `this` pekar på förekomsten som innehåller metoden:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Arv av förekomstmetoder kan kontrolleras med nyckelorden `override` och `final`. Du kan använda attributet `override` för att omdefiniera en ärvd metod och attributet `final` för att förhindra att underklasser åsidosätter en metod.

Åtkomstmetoder med get och set

Med åtkomstfunktionerna get och set, som också kallas *get*- och *set*-metoder, kan du anpassa dig till programmeringsprinciperna för dold och kapslad information medan du samtidigt tillhandahåller ett enkelt programmeringsgränssnitt för de klasser du skapar. Med get- och set-metoderna blir klassegenskaperna privata för klassen, men användarna av klassen kan komma åt dessa egenskaper på samma sätt som om de kommer åt en klassvariabel i stället för att anropa en klassmetod.

Fördelen med detta är att du slipper de traditionella åtkomstfunktionerna med otympliga namn, till exempel `getPropertyName()` och `setPropertyName()`. En annan fördel med get och set är att du slipper ha två allmänna funktioner för varje egenskap som tillåter både läs- och skrivåtkomst.

Följande exempelklass GetSet innehåller get- och set-åtkomstfunktionerna som heter `publicAccess()` som ger åtkomst till den privata variabeln `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Om du försöker komma åt egenskapen `privateProperty` direkt returneras ett fel:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

I stället får användaren av klassen GetSet använda något som ser ut att vara egenskapen `publicAccess`, men som i själva verket är ett par get- och set-åtkomstfunktioner för den privata egenskapen `privateProperty`. I följande exempel instansieras klassen GetSet och här anges värdet för `privateProperty` med hjälp av den allmänna åtkomstfunktionen `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Med get- och set-funktionerna kan du också åsidosätta egenskaper som ärvs från en superklass, vilket inte är möjligt när du använder vanliga klassmedlemsvariabler. Klassmedlemsvariabler som deklarerats med nyckelordet `var` kan inte åsidosättas i en underklass. Egenskaper som har skapats med get- och set-funktioner har inte den här begränsningen. Du kan använda attributet `override` på get- och set-funktioner som ärvts från en superklass.

Bundna metoder

En bunden metod, som ibland kallas ett *metodslut*, är en metod som extraheras från sin förekomst. Bundna metoder är metoder som skickats som argument till en funktion eller som returnerats som värden från en funktion. Bundna metoder är en nyhet i ActionScript 3.0 och funktionen liknar ett funktionsslut på det sättet att den sparar sin lexikala miljö även när den extraheras från sin förekomst. Den största skillnaden mellan en bunden metod och ett funktionsslut är att referensen `this` för en bunden metod förblir länkad, eller bunden, till förekomsten som implementerar metoden. Referensen `this` i en bunden metod pekar alltså alltid på det ursprungliga objektet som implementerat metoden. När det gäller funktionsslut är referensen `this` generisk, vilket betyder att den pekar på det objekt som funktionen är kopplad till vid den tidpunkt den anropas.

Om du använder nyckelordet `this` måste du veta hur bundna metoder fungerar. Kom ihåg att nyckelordet `this` skapar en referens till en metods överordnade objekt. De flesta ActionScript-programmerare förväntar sig att nyckelordet `this` alltid representerar det objekt eller den klass som innehåller definitionen för en metod. Utan metodbindning stämmer det här inte alltid. I tidigare versioner av ActionScript refererar referensen `this` inte alltid till förekomsten som implementerat metoden. Om metoder extraheras från en förekomst i ActionScript 2.0 är referensen `this` inte bunden till ursprungsförekomsten och medlemsvariablerna och metoderna för förekomstens klass är dessutom inte tillgängliga. Detta är inget problem i ActionScript 3.0 eftersom bundna metoder automatiskt skapas när du skickar en metod som parameter. Med bundna metoder kommer nyckelordet `this` alltid att referera till det objekt eller den klass som en metod är definierad i.

I följande kod definieras klassen ThisTest som innehåller metoden `foo()` som definierar den bundna metoden, och metoden `bar()` som returnerar den bundna metoden. Kod som ligger utanför klassen skapar en förekomst av klassen ThisTest, anropar metoden `bar()` och lagrar returvärdet i variabeln `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 sista två kodraderna visar att referensen `this` i den bundna metoden `foo()` fortfarande pekar på en förekomst av klassen ThisTest, även om referensen `this` på raden före pekar på det globala objektet. Den bundna metoden som lagrats i variabeln `myFunc` har dessutom fortfarande åtkomst till medlemsvariablerna i klassen ThisTest. Om samma kod körs i ActionScript 2.0 kommer referensen `this` att matcha och variabeln `num` kommer att bli `undefined`.

Om du lägger till bundna metoder med händelsehanterare märks det mest eftersom metoden `addEventListener()` kräver att du skickar en funktion eller metod som ett argument.

Uppräkningar med klasser

Uppräkningar är anpassade datatyper som du skapar för att kapsla in en liten uppsättning värden. I ActionScript 3.0 finns inte någon särskild uppräkningsfunktion i motsats till C++ som har nyckelordet `enum` och Java som har ett uppräkningsgränssnitt. Du kan emellertid skapa uppräkningar med hjälp av klasser och statiska konstanter. I klassen PrintJob i ActionScript 3.0 används till exempel en uppräkning som heter PrintJobOrientation för att spara värdena `"landscape"` och `"portrait"`, vilket visas i följande kod:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

En uppräkningsklass deklareras med attributet `final` eftersom klassen inte behöver utökas. Klassen innehåller bara statiska medlemmar, vilket betyder att du inte skapar några instanser av klassen. I stället kommer du åt uppräkningsvärdena direkt via objektet class, vilket visas i följande kodutdrag:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Uppräkningsklasserna i ActionScript 3.0 innehåller bara variabler av typen String, int och uint. Fördelen med att använda uppräkningar i stället för litterala strängvärden eller talvärden är att det är lättare att hitta typografiska misstag med uppräkningar. Om du skriver namnet på en uppräkning fel genereras ett fel i ActionScript-kompilatorn. Om du använder litterala värden reagerar kompilatorn inte om du stavar ett ord fel eller använder fel tal. I föregående exempel genereras ett fel om namnet på uppräkningskonstanten är felaktigt, vilken visas i följande utdrag:

```
if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Inget fel genereras emellertid om du stavar fel på ett stränglitteralvärde:

```
if (pj.orientation == "portrai") // no compiler error
```

Ett annat sätt att skapa uppräkningar är att skapa en separat klass med statiska egenskaper för uppräkningen. Detta sätt är annorlunda på så sätt att varje statisk egenskap innehåller en förekomst av klassen i stället för en sträng eller ett heltal. Med följande kod skapas en uppräkningsklass för veckodagarna:

```
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();
}
```

Denna metod används inte i ActionScript 3.0 men används av många utvecklare som föredrar den förbättrade typkontroll som den här metoden ger. En metod som till exempel returnerar ett uppräkningsvärde kan begränsa returvärdet till uppräkningens datatyp. Med följande kod visas en funktion som returnerar en veckodag men också ett funktionsanrop som använder uppräkningstypen som en typanteckning.

```
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();
```

Du kan också utöka klassen Day så att den kopplar ett heltal till varje veckodag och som tillhandahåller en `toString()`-metod som returnerar ett strängbegrepp för dagen.

Inbäddade resursklasser

I ActionScript 3.0 används specialklasser, som även kallas *inbäddade resursklasser*, till att representera inbäddade resurser. En *inbäddad resurs* är en resurs, till exempel ett ljud, en bild eller ett teckensnitt, som ingår i en SWF-fil vid kompileringen. Om du bäddar in en resurs i stället för att läsa in den dynamiskt blir den tillgänglig vid körning, men SWF-filens storlek ökar då också.

Använda inbäddade resursklasser i Flash Professional

Om du vill bädda in en resurs placerar du först resursen i en FLA-fils bibliotek. Därefter använder du resursens länkningsegenskap för att ge resursens inbäddade klass ett namn. Om det inte finns någon klass med detta namn i klassökvägen genereras en klass automatiskt. Därefter kan du skapa en förekomst av den inbäddade resursklassen och använda alla egenskaperna och metoderna som definierats eller ärvts av denna klass. Följande kod kan till exempel användas för att spela upp ett inbäddat ljud som är länkat till en inbäddad resursklass som heter PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Du kan också använda metadatataggen `[Embed]` för att bädda in resurser i ett Flash Professional-projekt, vilket beskrivs nedan. Om du använder metadatataggen `[Embed]` i koden används Flex-kompilatorn för att kompilera projektet i stället för Flash Professional-kompilatorn.

Använda inbäddade resursklasser med Flex-kompilatorn

Om du kompilerar koden med Flex-kompilatorn och vill bädda in en resurs i ActionScript-koden använder du metadatataggen `[Embed]`. Placera resursen i huvudkällmappen eller i en annan mapp som finns i projektets byggsökväg. När Flex-kompilatorn påträffar en Embed-metadatatagg skapas den inbäddade resursklassen. Du får tillgång till klassen via en variabel av datatypen Class som du deklarerar omedelbart efter `[Embed]`-metadatataggen. I följande kod bäddas ett ljud med namnet sound1.mp3 in och en variabel med namnet `soundCls` används för att lagra en referens till den inbäddade resursklassen som hör till ljudet. I exemplet skapas sedan en förekomst av den inbäddade resursklassen och metoden `play()` anropas för den förekomsten:

```
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

Om du vill använda metadatataggen `[Embed]` i ett Flash Builder ActionScript-projekt importerar du de klasser som behövs från Flex-ramverket. Om du till exempel vill bädda in ljud importerar du klassen mx.core.SoundAsset. Om du vill använda Flex-ramverket inkluderar du filen framework.swc i ActionScript-byggsökvägen. Detta ökar storleken på SWF-filen.

Adobe Flex

I Flex kan du också bädda in en resurs med direktivet `@Embed()` i en MXML-taggdefinition.

Gränssnitt

Ett gränssnitt är en samling metoddeklarationer som tillåter att objekt som inte har med varandra att göra ändå kommunicerar med varandra. Gränssnittet IEventdispatcher definieras i ActionScript 3.0 och gränssnittet innehåller metoddeklarationer som kan användas av en klass för att hantera händelseobjekt. Med gränssnittet IEventDispatcher kan objekt skicka händelseobjekt mellan varandra på ett standardmässigt sätt. I följande kod visas definitionen av gränssnittet 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;
}
```

Gränssnitt baseras på skillnaden mellan en metods gränssnitt och dess implementering. En metods gränssnitt innehåller all information som krävs för att anropa den metoden, däribland metodens namn, parametrar och returtyp. En metods implementering innehåller gränssnittsinformation men också de körbara programsatser som utför metodens beteende. En gränssnittsdefinition innehåller bara metodgränssnitt och alla klasser som implementerar gränssnittet ansvarar för att definiera metodimplementeringarna.

I ActionScript 3.0 implementeras gränssnittet IEventDispatcher av klassen EventDispatcher genom att alla IEventDispatcher-gränssnittsmetoder definieras och metodbrödtexter läggs till i alla metoder. Följande kod är ett utdrag av definitionen för klassen EventDispatcher:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

IEventDispatcher-gränssnittet fungerar som ett protokoll som används av EventDispatcher-förekomster för att bearbeta händelseobjekt och skicka dem till andra objekt som också har implementerat IEventDispatcher-gränssnittet.

Man kan också säga att ett gränssnitt definierar en datatyp på samma sätt som en klass gör. Därför kan ett gränssnitt, på samma sätt som en klass, användas som en typanteckning. Som en datatyp kan gränssnittet också användas med operatorer, till exempel `is` och `as`, som kräver en datatyp. I motsats till en klass kan ett gränssnitt inte instansieras. Den här skillnaden har gjort att många programmerare tänker sig gränssnitt som abstrakta datatyper och klasser som konkreta datatyper.

Definiera ett gränssnitt

En gränssnittsdefinitions struktur liknar en klassdefinitions struktur, förutom att gränssnittet bara kan innehålla metoder som saknar metodbrödtext. Gränssnitt kan inte innehålla variabler och konstanter, men kan innehålla get- och set-funktioner. Använd nyckelordet `interface` när du definierar ett gränssnitt. Gränssnittet IExternalizable är en del av paketet flash.utils i ActionScript 3.0. Med det här gränssnittet definieras ett protokoll som används för serialisering av ett objekt, vilket betyder konvertering av ett objekt till ett format som passar för lagring på en enhet eller för transport över ett nätverk.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

Gränssnittet IExternalizable deklareras med åtkomstkontrollsmodifieringen `public`. Gränssnittsdefinitioner kan bara ändras med åtkomstkontrollsspecifikationerna `public` och `internal`. Metoddeklarationerna i en gränssnittsdefinition får inte ha några åtkomstkontrollsspecifikationer.

I ActionScript 3.0 används en standard där gränssnittsnamn börjar med ett versalt `I`, men du kan använda vilken tillåten identifierare som helst som gränssnittsnamn. Gränssnittsdefinitioner placeras ofta på den översta paketnivån. Gränssnittsdefinitioner får inte placeras i en klassdefinition eller i en annan gränssnittsdefinition.

Gränssnitt kan utöka ett eller flera andra gränssnitt. Gränssnittet IExample utökar till exempel gränssnittet IExternalizable:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Alla klasser som implementerar gränssnittet IExample måste innehålla implementeringar för metoden `extra()` men också för metoderna `writeExternal()` och `readExternal()` som ärvts från gränssnittet IExternalizable.

Implementera ett gränssnitt i en klass

En klass är det enda språkelementet i ActionScript 3.0 som kan implementera ett gränssnitt. Använd nyckelordet `implements` i en klassdeklaration för att implementera ett eller flera gränssnitt. I följande exempel definieras de två gränssnitten IAlpha och IBeta, samt klassen Alpha, som implementerar båda två:

```
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 {}
}
```

I en klass som implementerar ett gränssnitt måste följande göras av de implementerade metoderna:

- Använda åtkomstkontrollsidentifieraren `public`.
- Använda samma namn som gränssnittsmetoden.
- Ha samma antal parametrar, var och en med datatyper som matchar gränssnittsmetodens parametrars datatyper.
- Använda samma returtyp.

```
public function foo(param:String):String {}
```

Du kan vara ganska flexibel när det gäller namnet på parametrarna för de metoder du implementerar. Även om antalet parametrar och datatypen för varje parameter i den implementerade metoden måste matcha gränssnittsmetodens, behöver parameternamnet inte matcha. I föregående exempel har parametern för metoden `Alpha.foo()` fått namnet `param`:

Men parametern har namnet `str` i gränssnittsmetoden `IAlpha.foo()`:

```
function foo(str:String):String;
```

Du kan också vara flexibel när det gäller standardparametervärden. En gränssnittsdefinition kan innehålla funktionsdeklarationer med standardparametervärden. En metod som implementerar en sådan funktionsdeklaration måste ha ett standardparametervärde som är medlem av samma datatyp som värdet som angetts i gränssnittsdefinitionen, men det verkliga värdet behöver inte matcha. I följande kod definieras ett gränssnitt som innehåller en metod som har standardparametervärdet 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

Följande klassdefinition implementerar gränssnittet IGamma, men använder ett annat standardparametervärde:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Orsaken till detta är att reglerna för implementering av ett gränssnitt utformats speciellt för att säkra datatypskompatibiliteten, och det är inte nödvändigt att parameternamn och standardparametervärden är identiska för att detta mål ska uppnås.

Arv

Arv är en form av kodåteranvändning som gör att programmerare kan utveckla nya klasser som baseras på befintliga klasser. De befintliga klasserna kallas ofta *basklasser* eller *superklasser*, och de nya klasserna kallas *underklasser*. Den största fördelen med arv är att du kan återanvända kod från en basklass och ändå låta den befintliga koden vara oförändrad. Dessutom kräver arv inga ändringar i det sätt som andra klasser samverkar med basklassen. I stället för att du ändrar en befintlig klass som har testats noggrant eller som redan används, kan du med arv hantera denna klass som en integrerad modul som du kan utöka med ytterligare egenskaper och metoder. Därför kan du använda nyckelordet `extends` för att ange att en klass ska ärva från en annan klass.

Tack vare arv kan du också dra fördel av *polymorfism* i koden. Polymorfism är möjligheten att använda ett enstaka metodnamn för en metod som beter sig på olika sätt när den tillämpas på olika datatyper. Ett enkelt exempel är en basklass som heter Shape som har två underklasser som heter Circle och Square. Med klassen Shape definieras metoden `area()` som returnerar formens område. Om du implementerat polymorfism kan du anropa metoden `area()` på objekt av typen Circle och Square och få korrekta beräkningar. Tack vare arv möjliggörs polymorfism genom att underklasser kan ärva och omdefiniera, eller *åsidosätta*, metoder från basklassen. I följande exempel omdefinieras metoden `area()` med klasserna Circle och 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
```

Eftersom varje klass definierar en datatyp skapas med arv en speciell relation mellan en basklass och en klass som utökar den. En underklass garanteras äga alla basklassens egenskaper, vilket betyder att en förekomst av en underklass alltid kan ersättas med en förekomst av basklassen. Om en metod definierar en parameter av typen Shape, är det tillåtet att skicka ett argument av typen Circle eftersom Circle utökar Shape:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Förekomstegenskaper och arv

En förekomstegenskap, som definierats med nyckelordet `function`, `var` eller `const`, ärvs av alla underklasser såvida inte egenskapen deklarerats med attributet `private` i basklassen. Klassen Event i ActionScript 3.0 har till exempel ett antal underklasser som ärver de egenskaper som är gemensamma för alla händelseobjekt.

För vissa typer av händelser innehåller klassen Event alla egenskaper som är nödvändiga för att definiera händelsen. Dessa typer av händelser kräver inte några fler förekomstegenskaper än de som definierats i klassen Event. Ett exempel på sådana händelser kan vara `complete` som inträffar när data har lästs in samt `connect` som inträffar när en nätverksanslutning har upprättats.

Följande exempel är ett utdrag från klassen Event som visar några av de egenskaper och metoder som ärvs av underklasser. Eftersom egenskaperna ärvs kan en förekomst av valfri underklass komma åt dessa egenskaper.

```
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 {}
    ...
}
```

Andra typer av händelser kräver unika egenskaper som inte är tillgängliga i klassen Event. Dessa händelser definieras med hjälp av underklassen till klassen Event så att nya egenskaper kan läggas till i de egenskaper som definierats i klassen Event. Ett exempel på en sådan underklass är klassen MouseEvent som lägger till egenskaper som är unika för händelser som kopplats till musrörelser och musklickningar, till exempel händelserna `mouseMove` och `click`. Följande exempel är ett utdrag från klassen MouseEvent som visar definitionen av egenskaper som finns i underklassen men inte i basklassen:

```
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 {}
    ...
}
```

Åtkomstkontrollsspecifikationer och arv

Om en egenskap har deklarerats med nyckelordet `public` visas egenskapen för all kod. Detta betyder att nyckelordet `public`, i motsats till nyckelorden `private`, `protected` och `internal`, inte har några begränsningar för egenskapsarv.

Om en egenskap har deklarerats med nyckelordet `private` visas den bara i den klass som definierar den, vilket betyder att den inte ärvs av någon underklass. Detta skiljer sig från tidigare versioner av ActionScript, där nyckelordet `private` beter sig snarlikt nyckelordet `protected` i ActionScript 3.0.

Med nyckelordet `protected` anges att en egenskap visas i den klass som definierar den men också i alla underklasser. Med nyckelordet `protected` i ActionScript 3.0 visas en egenskap inte för alla andra klasser i samma paket, vilket sker med nyckelordet `protected` i programmeringsspråket Java. I ActionScript 3.0 kan bara underklasser komma åt en egenskap som deklarerats med nyckelordet `protected`. Dessutom visas en skyddad egenskap för en underklass vare sig underklassen är i samma paket som basklassen eller i ett annat paket.

Om du vill begränsa visningen av en egenskap i det paket som egenskapen är definierad i, använder du nyckelordet `internal` eller också använder du inte någon åtkomstkontrollsspecifikation. Åtkomstkontrollsspecifikationen `internal` är den standardspecifikation som tillämpas när ingen annan har angetts. En egenskap som markerats som `internal` ärvs bara av en underklass som finns i samma paket.

Du kan använda följande exempel för att visa hur varje åtkomstkontrollsspecifikation påverkar arv över paketgränserna. I följande kod definieras en huvudprogramklass som heter AccessControl och två andra klasser som heter Base och Extender. Klassen Base finns i paketet foo och klassen Extender, som är en underklass till Base, finns i paketet bar. Klassen AccessControl importerar bara klassen Extender och skapar en förekomst av Extender som försöker komma åt variabeln `str` som definierats i klassen Base. Variabeln `str` deklareras som `public` så att koden kompileras och körs, vilket visas i följande utdrag:

```
// 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
        }
    }
}
```

Om du vill visa hur de andra åtkomstkontrollsspecifikationerna påverkar kompilering och körning av föregående exempel ändrar du `str`-variabelns åtkomstkontrollsspecifikation till `private`, `protected` eller `internal` när du har tagit bort eller kommenterat ut följande rad från klassen `AccessControl`:

```
trace(myExt.str);// error if str is not public
```

Du kan inte åsidosätta variabler

Egenskaper som har deklarerats med nyckelorden `var` eller `const` ärvs, men kan inte åsidosättas. Att åsidosätta en egenskap är att definiera om egenskapen i en underklass. Den enda typ av egenskap som kan åsidosättas är get- och set-åtkomstfunktioner (egenskaper som deklarerats med nyckelordet `function`). Även om du inte kan åsidosätta en förekomstvariabel kan du uppnå ett liknande resultat genom att skapa get- och set-metoder för förekomstvariabeln och åsidosätta metoderna.

Åsidosätta metoder

Att åsidosätta en metod är att omdefiniera beteendet för en ärvd metod. Statiska metoder ärvs inte och kan inte åsidosättas. Förekomstmetoder ärvs emellertid av underklasser och kan åsidosättas om följande två villkor uppfylls:

- Förekomstmetoden har inte deklarerats med nyckelordet `final` i basklassen. När nyckelordet `final` används med en förekomstmetod anges programmerarens avsikt att förhindra att underklasser åsidosätter metoden.
- Förekomstmetoden har inte deklarerats med åtkomstkontrollsspecifikationen `private` i basklassen. Om en metod har markerats som `private` i basklassen behöver du inte använda nyckelordet `override` när du definierar en metod med samma namn i underklassen, eftersom basklassens metod inte är synlig för underklassen.

  Om du vill åsidosätta en förekomstmetod som uppfyller dessa villkor, måste metoddefinitionen i underklassen använda nyckelordet `override` och dessutom matcha superklassversionen av metoden på följande sätt:

- Åsidosättningsmetoden måste ha samma åtkomstkontrollnivå som basklassmetoden. Metoder som markerats som interna har samma åtkomstkontrollnivå som metoder som inte har någon åtkomstkontrollsspecifikation.
- Åsidosättningsmetoden måste ha samma antal parametrar som basklassmetoden.
- Åsidosättningsparametrarna måste ha samma datatypsanteckningar som parametrarna i basklassmetoden.
- Åsidosättningsmetoden måste ha samma returtyp som basklassmetoden.

Namnen på parametrarna i åsidosättningsmetoden behöver däremot inte matcha namnen på parametrarna i basklassen, så länge som antalet parametrar och datatypen för respektive parameter matchar varandra.

Programsatsen super

När en programmerare åsidosätter en metod vill han ofta lägga till beteendet hos superklassmetoden som åsidosätts i stället för att ersätta beteendet helt och hållet. Detta kräver en funktion som tillåter att en metod i en underklass kan anropa sin egen superklassversion. Programsatsen `super` har en sådan funktion eftersom den innehåller en referens till den omedelbara superklassen. I följande exempel definieras klassen Base som innehåller metoden `thanks()` och underklassen Extender för klassen Base som åsidosätter metoden `thanks()`. Metoden `Extender.thanks()` använder programsatsen `super` för att anropa `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Åsidosätta get- och set-metoder

Även om du inte kan åsidosätta variabler som definierats i en superklass kan du åsidosätta get- och set-metoder. I följande kod åsidosätts en get-metod som heter `currentLabel` som definierats i klassen MovieClip i ActionScript 3.0:

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

Utdata för programsatsen `trace()` i klasskonstruktorn OverrideExample är `Override: null`, vilket visar att exemplet kunde åsidosätta den ärvda egenskapen `currentLabel`.

Statiska egenskaper ärvs inte

Statiska egenskaper ärvs inte av underklasser. Det betyder att statiska egenskaper inte är tillgängliga via en förekomst av en underklass. En statisk egenskap är bara tillgänglig via klassobjektet som den definierats på. I följande kod definieras basklassen Base och underklassen Extender som utökar Base. Den statiska variabeln `test` definieras i klassen Base. Koden som skrivits i följande utdrag kompileras inte i strikt läge och genererar ett körningsfel i standardläge.

```
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 { }
```

Enda sättet att komma åt den statiska variabeln `test` är via objektet class, vilket visas i följande kod:

```
Base.test;
```

Du kan emellertid definiera en förekomstegenskap med samma namn som en statisk egenskap. En sådan förekomstegenskap kan definieras i samma klass som den statiska egenskapen eller i en underklass. Klassen Base i föregående exempel kan alltså ha en förekomstegenskap som heter `test`. Följande kod kompileras och körs eftersom förekomstegenskapen ärvs av klassen Extender. Koden kompileras och körs också om definitionen av förekomstvariabeln test flyttas, men inte kopieras, till klassen 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 {}
```

Statiska egenskaper och omfångskedjan

Även om statiska egenskaper inte ärvs finns de i omfångskedjan för den klass som definierar dem och i alla underklasser till denna klass. Statiska egenskaper sägs vara *i omfånget* för de två klasser som de definierats i och i alla underklasser. Detta betyder att en statisk egenskap är direkt tillgänglig i brödtexten för den klass som definierar den statiska egenskapen och i alla underklasser till denna klass.

I följande exempel ändras de klasser som definierats i föregående exempel så att du kan se att den statiska variabeln `test` som definierats i klassen Base finns i omfånget till klassen Extender. Klassen Extender kan m.a.o. komma åt den statiska variabeln `test` utan att använda namnet på den klass som definierar `test` som prefix.

```
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
    }

}
```

Om du definierar en förekomstegenskap och ger den samma namn som en statisk egenskap i samma klass eller en superklass, får förekomstegenskapen högre prioritet i omfångskedjan. Förekomstegenskapen *skuggar* den statiska egenskapen, vilket betyder att värdet för förekomstegenskapen används i stället för värdet för den statiska egenskapen. I följande kod visas, att om klassen Extender definierar en förekomstvariabel som heter `test`, använder programsatsen `trace()` värdet för förekomstvariabeln i stället för värdet för den statiska variabeln:

```
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
    }

}
```

Avancerade användningsområden

Historik över ActionScript OOP-stöd

Eftersom ActionScript 3.0 bygger på tidigare versioner av ActionScript bör du känna till hur ActionScript-objektmodellen har utvecklats. ActionScript började som en enkel skriptfunktion för tidiga versioner av Flash Professional. Programmerarna började efterhand bygga mera komplexa program med ActionScript. Som svar på deras behov har efterföljande versioner fått nya språkfunktioner som förenklar komplexa program.

ActionScript 1.0

ActionScript 1.0 är den version av språket som används i Flash Player 6 och tidigare versioner. Även i detta tidiga utvecklingsstadium var ActionScript-objektmodellen baserad på konceptet med objekt som fundamental datatyp. Ett ActionScript-objekt är en sammansatt datatyp med en grupp *egenskaper*. När objektmodellen beskrivs ingår i termen *egenskaper* allt som bifogats till ett objekt, till exempel variabler, funktioner och metoder.

Även om den första generationen ActionScript inte stöder definitionen av klasser med nyckelordet `class`, kan du definiera en klass med ett särskilt objekt som kallas prototypobjekt. I stället för att använda nyckelordet `class` för att skapa en abstrakt klassdefinition som du instansierar till konkreta objekt, som du gör i klassbaserade språk som Java och C++, använder prototypbaserade språk som ActionScript 1.0 ett befintligt objekt som modell (eller prototyp) för andra objekt. Medan objekt i ett klassbaserat språk kan peka på en klass som fungerar som mall, pekar objekt i ett prototypbaserat språk på ett annat objekt, prototypen, som fungerar som mall.

Om du vill skapa en klass i ActionScript 1.0 definierar du en konstruktorfunktion för denna klass. I ActionScript är funktioner verkliga objekt och inte bara abstrakta definitioner. Konstruktorfunktionen som du skapar fungerar som prototypobjekt för förekomster av den här klassen. I följande kod skapas klassen Shape och definieras egenskapen `visible` som har värdet `true` som standard:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Den här konstruktorfunktionen definierar klassen Shape som du kan instansiera med operatorn `new`:

```
myShape = new Shape();
```

På samma sätt som konstruktorfunktionsobjektet `Shape()` fungerar som prototyp för förekomster av klassen Shape, kan det också fungera som prototyp för underklasser till Shape, d.v.s. andra klasser som utökar klassen Shape.

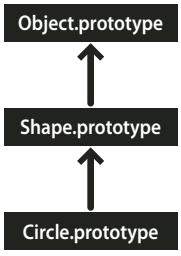
Att skapa en klass som är underklass till klassen Shape är en tvåstegsprocess. Först definierar du en konstruktorfunktion för klassen:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Sedan använder du operatorn `new` för att deklarera att klassen Shape är prototyp för klassen Circle. Som standard används klassen Object som prototyp för alla klasser du skapar, vilket betyder att `Circle.prototyp` innehåller ett generiskt objekt (en förekomst av klassen Object). Om du vill ange att Circles prototyp är Shape och inte Object använder du följande kod för att ändra värdet för `Circle.prototyp` så att det innehåller ett Shape-objekt i stället för ett generiskt objekt:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

Klassen Shape och Circle har nu kopplats ihop i en arvsrelation som kallas *prototypkedja*. Bilden visar relationen i en prototypkedja:



Basklassen i slutet av alla prototypkedjor är klassen Object. Klassen Object innehåller den statiska egenskapen `Object.prototype`, som pekar på basprototypsobjektet för alla objekt som skapas i ActionScript 1.0. Nästa objekt i prototypkedjan är objektet Shape. Detta beror på att egenskapen `Shape.prototype` aldrig angavs explicit, och den innehåller fortfarande ett generiskt objekt (en förekomst av klassen Object). Den sista länken i kedjan är klassen Circle som är kopplad till sin prototyp, klassen Shape (egenskapen `Circle.prototype` innehåller ett Shape-objekt).

Om du skapar en instans av klassen Circle, som i följande exempel, ärver instansen prototypkedjan för klassen Circle:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Kom ihåg att exemplet också innehöll egenskapen `visible` som medlem i klassen Shape. I det här exemplet finns egenskapen `visible` inte som en del av objektet `myCircle`, utan bara som medlem i objektet Shape, och ändå returnerar följande kodrad `true`:

```
trace(myCircle.visible); // output: true
```

Körningsmiljön kan fastställa att objektet `myCircle` ärver egenskapen `visible` genom att gå upp i prototypkedjan. När den här koden körs genomsöks först egenskaperna för objektet `myCircle` efter egenskapen `visible`, men utan att hitta den. Därefter genomsöks objektet `Circle.prototype`, men fortfarande hittas inte egenskapen `visible`. Efterhand som sökningen rör sig uppåt i prototypkedjan hittas egenskapen `visible`, som definierats för objektet `Shape.prototype`, och värdet för egenskapen matas ut.

För enkelhets skull utelämnas många av de invecklade detaljerna kring prototypkedjan. Målet är i stället att ge tillräckligt mycket information för att ge en förståelse för objektmodellen i ActionScript 3.0.

ActionScript 2.0

ActionScript 2.0 innehåller nya nyckelord som `class`, `extends`, `public` och `private`, som gör att du kan definiera klasser på ett sätt som är bekant för alla som arbetar med klassbaserade språk som Java och C++. Du bör känna till att den underliggande arvsmekanismen inte ändrats mellan ActionScript 1.0 och ActionScript 2.0. I ActionScript 2.0 har bara infogats ny syntax för klassdefinition. Prototypkedjan fungerar på samma sätt i båda språkversionerna.

Med den nya syntaxen som introducerades i ActionScript 2.0 kan du definiera klasser på ett mera intuitivt sätt, vilket visas i följande utdrag:

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

Observera att ActionScript 2.0 också innehåller typanteckningar som ska användas för typkontroll vid kompilering. Du kan då deklarera att egenskapen `visible` i föregående exempel ska innehålla bara ett booleskt värde. Med det nya nyckelordet `extends` förenklas också processen att skapa en underklass. I följande exempel har tvåstegsprocessen som är nödvändig i ActionScript 1.0 utförts i ett steg med nyckelordet `extends`.

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

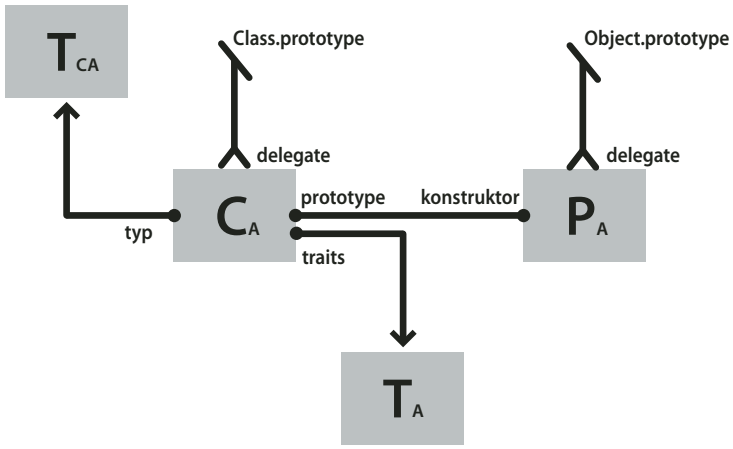
Konstruktorn deklareras nu som en del av klassdefinitionen, och klassegenskaperna `id` och `radius` måste också deklareras explicit.

ActionScript 2.0 har också stöd för definition av gränssnitt vilket gör att du ytterligare kan förfina dina objektorienterade program med formellt definierade protokoll för kommunikation mellan objekt.

Klassobjektet i ActionScript 3.0

Ett vanligt objektorienterat programmeringsparadigm, som vanligtvis associeras med Java och C++, använder klasser för att definiera olika typer av objekt. Programmeringsspråk som innehåller det här paradigmet tenderar att använda klasser för att skapa förekomster av den datatyp som definieras av klassen. I ActionScript används klasser för båda dessa ändamål, men programmets rötter som ett prototypbaserat språk inbegriper en intressant egenskap. I ActionScript skapas för varje klassdefinition ett speciellt klassobjekt som tillåter delning av både beteende och tillstånd. För de flesta ActionScript-programmerare får denna skillnad inga praktiska kodningskonsekvenser. I ActionScript 3.0 kan du skapa sofistikerade objektorienterade ActionScript-program utan att behöva använda, eller ens förstå, dessa speciella klassobjekt.

I följande bild visas strukturen hos ett klassobjekt som representerar en enkel klass som heter A som definierats med programsatsen `class A {}`:



Varje rektangel i bilden representerar ett objekt. Varje objekt i bilden har ett nedsänkt A som anger att det hör till klass A. Klassobjektet (CA) innehåller referenser till ett antal andra viktiga objekt. Ett förekomst-traits-objekt (TA) lagrar förekomstegenskaperna som definierats i en klassdefinition. Ett klass-traits-objekt (TCA) representerar den interna typen av klassen och lagrar de statiska egenskaper som definierats av klassen (det nedsänkta tecknet C står för ”class”). Prototypobjektet (PA) avser alltid det klassobjekt som det ursprungligen kopplades till via egenskapen `constructor`.

Objektet traits

Objektet traits, som är nytt i ActionScript 3.0, har implementerats för att höja prestanda. I tidigare versioner av ActionScript kunde namnsökning vara ganska tidsödande eftersom Flash Player gick uppåt i prototypkedjan. I ActionScript 3.0 är namnsökningen mycket effektivare och mindre tidsödande eftersom ärvda egenskaper kopieras ned från superklasserna till underklassernas traits-objekt.

Objektet traits är inte direkt åtkomligt för programmeringskoden, men du märker att det finns genom att prestanda och minnesanvändning har förbättrats. Via objektet traits får AVM2 detaljerad information om en klass layout och innehåll. Med denna kunskap kan körningstiden med AVM2 förkortas eftersom direkta maskininstruktioner kan genereras till åtkomstegenskaperna och metoderna kan anropas direkt utan tidsödande namnsökning.

Tack vare objektet traits kan ett objekts minnesavtryck bli mycket mindre än ett liknande objekts avtryck i tidigare versioner av ActionScript. Om en klass är fast (d.v.s. inte har deklarerats dynamiskt) behöver en förekomst av klassen inte en hashtabell för dynamiskt infogade egenskaper och förekomsten kan innehålla lite mer än en pekare till traits-objekt och några fack för de fasta egenskaper som definierats i klassen. Därför kan ett objekt som kräver 100 byte minne i ActionScript 2.0 bara kräva 20 byte i ActionScript 3.0.

Obs! *Objektet traits är en intern implementeringsdetalj och det finns ingen garanti för att det inte ändras eller försvinner i framtida versioner av ActionScript.*

Prototypobjektet

Varje objekt tillhörande klassen ActionScript har egenskapen `prototype`, som är en referens till klassens prototypobjekt. Prototypobjektet är ett arv från ActionScript som prototypbaserat språk. Mer information finns i Historik över ActionScript OOP-stöd.

Egenskapen `prototype` är skrivskyddad, vilket betyder att den inte kan ändras så att den pekar på andra objekt. Så är inte fallet med klassegenskapen `prototype` i tidigare versioner av ActionScript, där prototypen kan omtilldelas så att den pekar på en annan klass. Även om egenskapen `prototype` är skrivskyddad, är prototypobjektet som det refererar till inte skrivskyddat. Nya egenskaper kan alltså läggas till i prototypobjektet. Egenskaper som lagts till i prototypobjektet delas bland alla klassens förekomster.

Prototypkedjan, som var den enda arvsmekanismen i tidigare versioner av ActionScript, fungerar bara som en andra roll i ActionScript 3.0. Den primära arvsmekanismen, arv av fasta egenskaper, hanteras internt av objektet traits. En fast egenskap är en variabel, eller metod, som definierats som en del av en klassdefinition. Arv av fasta egenskaper kallas också klassarv eftersom det är den arvsmekanism som associeras med nyckelord som `class`, `extends` och `override`.

Med prototypkedjan fås en alternativ arvsmekanism som är mera dynamisk än arv av fasta egenskaper. Du kan lägga till egenskaper i en klass prototypobjekt som en del av klassdefinitionen, men också vid körning via klassobjektets `prototype`-egenskap. Om du ställer in kompilatorn på strikt läge kanske du bara kan komma åt egenskaper som lagts till i prototypobjektet om du deklarerar en klass med nyckelordet `dynamic`.

Ett bra exempel på en klass med flera egenskaper bifogade till prototypobjektet är klassen Object. Klassen Objects metoder `toString()` och `valueOf()` är i själva verket funktioner som kopplats till egenskaper för klassens prototypobjekt. Nedanstående är ett exempel på hur deklarationen av dessa metoder i teorin kan se ut (den verkliga implementeringen ser en aning annorlunda ut på grund av implementeringsdetaljer):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Som tidigare nämnts kan du bifoga en egenskap till prototypobjektet för en klass utanför klassdefinitionen. Metoden `toString()` kan till exempel också definieras utanför definitionen av klassen Object:

```
Object.prototype.toString = function()
{
    // statements
};
```

Prototyparv kräver inte, i motsats till arv av fasta egenskaper, att du använder nyckelordet `override` om du vill omdefiniera en metod i en underklass. Till exempel. Om du vill omdefiniera metoden `valueOf()` i en underklass till klassen Object finns det tre alternativ. För det första kan du definiera metoden `valueOf()` på underklassens prototypobjekt inuti klassdefinitionen. I följande kod skapas en underklass av Object som heter Foo och omdefinieras metoden `valueOf()` på Foos prototypobjekt som en del av klassdefinitionen. Eftersom alla klasser ärver från Object behöver du inte använda nyckelordet `extends`.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

För det andra kan du definiera metoden `valueOf()` på Foos prototypobjekt utanför klassdefinitionen, vilket visas i följande kod:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

För det tredje kan du definiera en fast egendom som heter `valueOf()` som en del av klassen Foo. Denna teknik skiljer sig från de andra genom att den blandar arv av fast egenskap med prototyparv. Alla underklasser till Foo som vill omdefiniera `valueOf()` måste använda nyckelordet `override`. I följande kod visas `valueOf()` som definierats som en fast egenskap i Foo:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

Namnutrymmet AS3

Eftersom det finns två separata arvsmekanismer, arv av fast egenskap och prototyparv, skapas en intressant kompatibilitetsutmaning med hänsyn till egenskaper och metoder i huvudklasserna. Kompatibilitet med språkdefinitionen ECMAScript som ActionScript baseras på kräver prototyparv, vilket betyder att egenskaper och metoder i en huvudklass definieras för prototypobjektet för denna klass. Kompatibilitet med ActionScript 3.0 kräver däremot arv av fast egenskap, vilket betyder att egenskaper och metoder i en huvudklass definieras i klassdefinitionen med nyckelorden `const`, `var` och `function`. Användningen av fasta egenskaper i stället för prototypversionerna kan dessutom ge rejäla höjningar av körningsprestanda.

I ActionScript 3.0 löses detta problem genom att både prototyparv och arv av fasta egenskaper används för huvudklasserna. Varje huvudklass innehåller två uppsättningar egenskaper och metoder. Den ena uppsättningen definieras för prototypobjektet för att ge kompatibilitet med ECMAScript-specifikationen och den andra uppsättningen definieras med fasta egenskaper och AS3-namnutrymmet för att ge kompatibilitet med ActionScript 3.0.

Med hjälp av AS3-namnutrymmet är det lätt att välja mellan de två uppsättningarna egenskaper och metoder. Om du inte använder AS3-namnutrymmet ärver en förekomst av en huvudklass de egenskaper och metoder som definierats på huvudklassens prototypobjekt. Om du bestämmer dig för att använda AS3-namnutrymmet ärver en förekomst av en huvudklass AS3-versionerna eftersom fasta egenskaper alltid prioriteras framför prototypegenskaper. När en fast egenskap är tillgänglig används den alltså alltid i stället för en prototypegenskap med samma namn.

Du kan selektivt använda AS3-namnutrymmesversionen för en egenskap eller metod genom att kvalificera den med AS3-namnutrymmet. I följande kod används AS3-versionen av metoden `Array.pop()`:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Du kan också använda direktivet `use namespace` för att öppna AS3-namnutrymmet för alla definitioner i ett kodblock. I följande kod används direktivet `use namespace` för att öppna AS3-namnutrymmet för båda metoderna `pop()` och `push()`:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

I ActionScript 3.0 finns också kompilatoralternativ för alla uppsättningar av egenskaper så att du kan tillämpa AS3-namnutrymmet på hela programmet. Kompilatoralternativet `-as3` representerar AS3-namnutrymmet och kompilatoralternativet `-es` representerar alternativet för prototyparv (`es` står för ECMAScript). Om du vill öppna AS3-namnutrymmet för hela programmet anger du värdet `true` för kompilatoralternativet `-as3` och värdet `false` för kompilatoralternativet `-es`. Om du vill använda prototypversionerna anger du omvända värden för kompilatoralternativen. Standardkompileringsinställningarna för Flash Builder och Flash Professional är `-as3 = true` och `-es = false`.

Om du tänker utöka någon av huvudklasserna och åsidosätta några metoder, måste du känna till hur AS3-namnutrymmet kan påverka deklarationen av en åsidosatt metod. Om du använder AS3-namnutrymmet måste alla metoder som åsidosätts av en huvudklassmetod också använda AS3-namnutrymmet tillsammans med attributet `override`. Om du inte använder AS3-namnutrymmet och vill definiera om en huvudklassmetod i en underklass, kan du inte använda AS3-namnutrymmet eller nyckelordet `override`.

Exempel: GeometricShapes

I exempelprogrammet GeometricShapes visas hur ett antal objektorienterade begrepp och funktioner kan tillämpas med hjälp av ActionScript 3.0, däribland:

- Definiera klasser
- Utöka klasser
- Polymorfism och nyckelordet `override`
- Definition, utökning och implementering av gränssnitt

Här finns också en standardmetod som skapar klassförekomster och som visar hur du deklarerar ett returvärde som en förekomst av ett gränssnitt och använder det returnerade objektet på ett allmänt sätt.

Programfilerna för det här exemplet finns på www.adobe.com/go/learn_programmingAS3samples_flash_se. Programfilerna för GeometricShapes kan finnas i mappen Samples/GeometricShapes. Programmet består av följande filer:

Fil	Beskrivning
GeometricShapes.mxml eller GeometricShapes.fla	Huvudprogramfilen i Flash (FLA) eller Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	Metoder för definition av basgränssnittet som ska implementeras med alla GeometricShapes-programklasser.
com/example/programmingas3/geometricshapes/IPolygon.as	Ett gränssnitt som definierar metoder som ska implementeras med GeometricShapes-programklasser som ha flera sidor.
com/example/programmingas3/geometricshapes/RegularPolygon.as	En typ av geometrisk form som har lika långa sidor som förskjuts symmetriskt runt formens mittpunkt.
com/example/programmingas3/geometricshapes/Circle.as	En typ av geometrisk form som definierar en cirkel.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	En underklass till RegularPolygon som definierar en liksidig triangel.
com/example/programmingas3/geometricshapes/Square.as	En underklass till RegularPolygon som definierar en liksidig rektangel.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	En klass som innehåller en standardmetod för att skapa former som har fått en formtyp och en storlek.

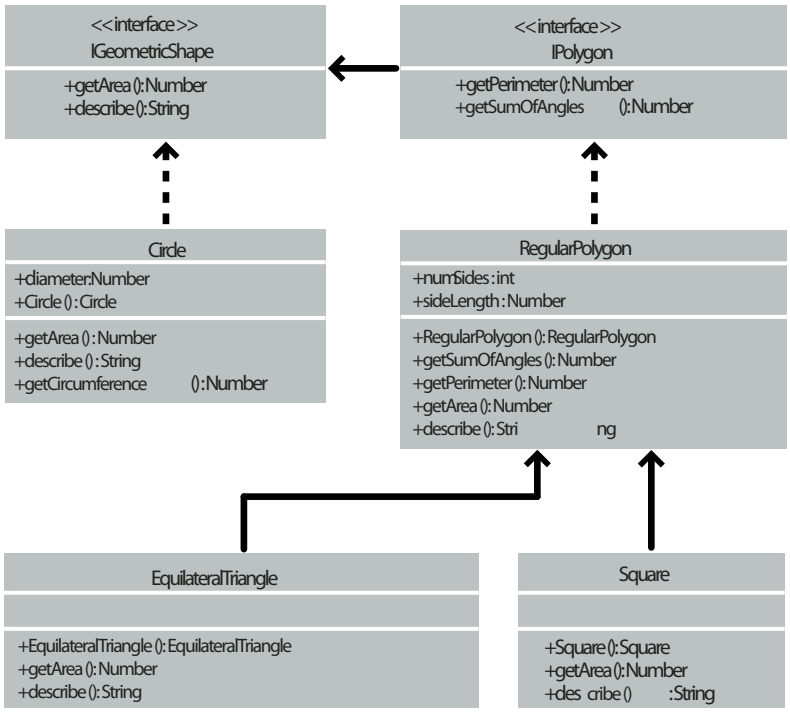
Definiera klasserna GeometricShapes

Med GeometricShapes-programmet kan användaren ange en typ av geometrisk form och storlek. Därefter svarar programmet med en beskrivning av formen, dess area och dess perimetersträcka.

Programmets användargränssnitt är enkelt och innehåller några få kontroller som du använder för att välja formtyp och storlek samt för att visa beskrivningen. Den intressantaste delen av programmet finns under ytan, i klasstrukturen och i själva gränssnittet.

Programmet hanterar geometriska former, men formerna visas inte grafiskt.

De klasser och gränssnitt som definierar de geometriska formerna i exemplet visas i följande bild med UML-notationen (Unified Modeling Language):



GeometricShapes, exempelklasser

Definiera allmänt beteende för gränssnitt

GeometricShapes-programmet hanterar tre typer av former: cirklar, fyrkanter och liksidiga trianglar. Strukturen i klasserna GeometricShapes börjar med ett mycket enkelt gränssnitt, IGeometricShape, som visar vilka metoder som är gemensamma för alla tre formtyperna:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

Gränssnittet definierar två metoder: metoden `getArea()`, som beräknar och returnerar formens area samt metoden `describe()`, som sätter ihop en textbeskrivning av formens egenskaper.

Vi vill också veta varje forms perimetersträcka. Perimetern för en cirkel kallas omkrets och beräknas på ett unikt sätt, så att beteendet skiljer sig från en triangels eller en fyrkants beteende. Det finns fortfarande vissa likheter mellan trianglar, fyrkanter och andra polygoner som gör att det verkar klokt att definiera en ny gränssnittsklass bara för dem: IPolygon. IPolygon-gränssnittet är också ganska enkelt, vilket visas här:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Med det här gränssnittet definieras två metoder som är gemensamma för alla polygoner: metoden `getPerimeter()` som mäter den kombinerade sträckan för alla sidor, samt metoden `getSumOfAngles()` som lägger till alla inre vinklar.

IPolygon-gränssnittet utökar IGeometricShape-gränssnittet, vilket betyder att alla klasser som implementerar IPolygon-gränssnittet måste deklarera alla fyra metoderna — de två från IGeometricShape-gränssnittet och de två från IPolygon-gränssnittet.

Definiera formklasser

När du vet metoderna för varje formtyp kan du definiera själva formklasserna. När det gäller hur många metoder du måste implementera är klassen Circle den enklaste formen och den visas här:

```
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;
        }
    }
}
```

Med klassen Circle implementeras gränssnittet IGeometricShape och den måste tillhandahålla kod för både metoden `getArea()` och för `describe()`. Den definierar dessutom metoden `getCircumference()` som är unik för klassen Circle. Klassen Circle deklarerar också egenskapen `diameter` som inte finns i andra polygonklasser.

De andra två formtyperna, fyrkanter och liksidiga trianglar, har andra saker gemensamt: de har sidor som är lika långa och det finns gemensamma formler du kan använda för att beräkna perimetern och summan av bådas inre vinklar. Dessa gemensamma formler gäller också för alla andra vanliga polygoner som du definierar i framtiden.

Klassen RegularPolygon är superklassen för både klassen Square och klassen EquilateralTriangle. Med en superklass kan du definiera gemensamma metoder på ett enda ställe, och du behöver alltså inte definiera dem separat i varje underklass. Här är koden för klassen 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;
        }
    }
}
```

Klassen RegularPolygon deklarerar två egenskaper som är gemensamma för alla vanliga polygoner: längden på varje sida (egenskapen `sideLength`) och antalet sidor (egenskapen `numSides`).

Klassen RegularPolygon implementerar gränssnittet IPolygon och deklarerar alla fyra IPolygon-gränssnittsmetoderna. Klassen implementerar två av dessa, metoderna `getPerimeter()` och `getSumOfAngles()`, med hjälp av vanliga formler.

Eftersom formeln för metoden `getArea()` skiljer sig från form till form, kan basklassversionen av metoden inte innehålla normalt logiskt beteende som kan ärvas av underklassmetoderna. I stället returneras standardvärdet 0 vilket anger att arean inte beräknats. Om arean för varje form ska beräknas på rätt sätt måste underklasserna för klassen RegularPolygon åsidosätta metoden `getArea()`.

I följande kod för klassen EquilateralTriangle visas hur metoden `getArea()` åsidosätts:

```
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;
        }
    }
}
```

Nyckelordet `override` anger att metoden `EquilateralTriangle.getArea()` avsiktligt åsidosätter metoden `getArea()` från superklassen RegularPolygon. När metoden `EquilateralTriangle.getArea()` anropas beräknas arean med hjälp av formeln i föregående kod, och koden i metoden `RegularPolygon.getArea()` körs aldrig.

Klassen EquilateralTriangle definierar däremot inte sin egen version av metoden `getPerimeter()`. När metoden `EquilateralTriangle.getPerimeter()` anropas, går anropet uppåt i arvskedjan och koden körs i metoden `getPerimeter()` för superklassen RegularPolygon.

Konstruktorn `EquilateralTriangle()` använder programsatsen `super()` för att starta konstruktorn `RegularPolygon()` för sin superklass. Om båda konstruktorerna har samma uppsättning parametrar kunde du ha uteslutit konstruktorn `EquilateralTriangle()` helt och hållet och konstruktorn `RegularPolygon()` kunde ha körts i stället. Konstruktorn `RegularPolygon()` måste ha en extra parameter, `numSides`. Konstruktorn `EquilateralTriangle()` anropar `super(len, 3)` som skickar indataparametern `len` och värdet 3 för att ange att triangeln har 3 sidor.

Metoden `describe()` använder också programsatsen `super()`, men på ett annat sätt. Den används där för att anropa superklassen RegularPolygons version av `describe()`-metoden. Metoden `EquilateralTriangle.describe()` anger först strängvariabeln `desc` till en programsats om typen av form. Därefter får den resultatet av metoden `RegularPolygon.describe()` genom att anropa `super.describe()`, och resultatet läggs till i strängen `desc`.

Klassen Square beskrivs inte här, men den liknar klassen EquilateralTriangle, som tillhandahåller en konstruktor och egna implementeringar av metoderna `getArea()` och `describe()`.

Polymorfism och standardmetoden

En uppsättning klasser som kan använda gränssnitt och arv kan användas på många olika intressanta sätt. Alla formklasserna som beskrivits hittills implementerar gränssnittet IGeometricShape eller utökar en superklass som gör det. Om du definierar en variabel som en instans av IGeometricShape behöver du inte känna till om det är en instans av klassen Circle eller Square för att kunna anropa dess `describe()`-metod.

Följande kod visar hur det fungerar:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

När `myShape.describe()` anropas körs metoden `Circle.describe()`; även om variabeln definierats som en förekomst av gränssnittet IGeometricShape interface är Circle dess underliggande klass.

Det här exemplet visar principen för hur polymorfism fungerar: exakt samma metodanrop resulterar i att olika koder körs, beroende på klassen för det objekt vars metod startas.

GeometricShapes-programmet tillämpar den här sortens gränssnittsbaserad polymorfism med hjälp av en förenklad version av ett designmönster som kallas standardmetoden. Termen *standardmetod* avser en funktion som returnerar ett objekt, vars underliggande datatyp eller innehåll kan vara olika beroende på sammanhanget.

Klassen GeometricShapeFactory som visas här definierar en standardmetod som kallas `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();
        }
    }
}
```

Med standardmetoden `createShape()` kan formens underklasskonstruktorer definiera detaljer för de förekomster de skapar, medan de returnerar de nya objekten som IGeometricShape-förekomster så att de kan hanteras av programmet på ett mera allmänt sätt.

Metoden `describeShape()` i föregående exempel visar hur ett program kan använda standardmetoden för att hämta en generisk referens till ett mera specifikt objekt. Programmet kan hämta beskrivningen för ett nyligen skapat Circle-objekt så här:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

Metoden `describeShape()` anropar därefter standardmetoden `createShape()` med samma parametrar, lagrar det nya Circle-objektet i en statisk variabel som kallas `currentShape` som skrivits in som ett IGeometricShape-objekt. Därefter anropas metoden `describe()` på objektet `currentShape` och detta anrop kör automatiskt metoden `Circle.describe()` som returnerar en detaljerad beskrivning av cirkeln.

Förbättra exempelprogrammet

Den verkliga styrkan hos gränssnitt och arv blir tydlig när du förbättrar eller ändrar programmet.

Låt oss anta att du vill lägga till en ny form, en femhörning, i det här exempelprogrammet. Du skapar en klass som heter Pentagon som utökar klassen RegularPolygon och definierar egna versioner av metoderna `getArea()` och `describe()`. Därefter lägger du till ett nytt Pentagon-alternativ i kombinationsrutan i programmets gränssnitt. Men det är allt. Klassen Pentagon får automatiskt funktionerna för metoden `getPerimeter()` och metoden `getSumOfAngles()` från den vanliga klassen RegularPolygon genom arv. Eftersom Pentagon-förekomsten ärver från en klass som implementerar gränssnittet IGeometricShape kan förekomsten också hanteras som en IGeometricShape-förekomst. Det innebär, att om du vill lägga till en ny typ av form, behöver du inte ändra metodsignaturen för någon av metoderna i klassen GeometricShapeFactory (och därför behöver du heller inte ändra någon kod som använder klassen GeometricShapeFactory).

Du kan lägga till en klass av typen Pentagon i Geometric Shapes-exemplet som övning om du vill se hur gränssnitt och arv kan minska arbetsbördan för dig när du lägger till nya funktioner i ett program.