ACTIONSCRIPT® 3.0'ı Öğrenme

Yasal bildirimler

Yasal bildirimler için bkz. http://help.adobe.com/tr_TR/legalnotices/index.html.

İçindekiler

Bölüm 1: ActionScript 3.0'a giriş

ActionScript hakkında

ActionScript, Adobe® Flash® Player ve Adobe® AIR™ çalışma zamanı ortamları için programlama dilidir. Flash, Flex ve AIR içerik ve uygulamalarında etkileşim, veri işleme ve daha fazlasına olanak sağlar.

ActionScript, Flash Player ve AIR'nin bir parçası olan ActionScript Virtual Machine (AVM) uygulamasında çalıştırılır. ActionScript kodu genellikle bir derleyici tarafından bayt kodu biçimine dönüştürülür. (*Bayt kodu*, bilgisayarlar tarafından yazılan ve anlaşılan bir programlama dili türüdür.) Derleyici örnekleri, Adobe® Flash®Professional'da ve Adobe® Flash®Builder™ uygulamalarında yerleşik olanı ve Adobe®Flex™SDK'de mevcut olanı içerir. Bayt kodu, Flash Player ve AIR'nin yürüttüğü SWF dosyalarına gömülüdür.

ActionScript 3.0, nesne tabanlı programlama konusunda temel bilgisi olan geliştiricilere tanıdık gelen sağlam bir programlama modeli sunar. ActionScript 3.0'ın önceki ActionScript sürümlerinden daha gelişmiş olan bazı önemli özellikleri arasında şunlar yer alır:

- AVM2 adı verilen ve yeni bir bayt kodu talimat kümesi kullanıp önemli ölçüde performans artışı sağlayan yeni bir ActionScript Virtual Machine.
- Önceki derleyici sürümlerinden daha derin eniyileştirmeler gerçekleştiren daha modern bir derleyici kodu
- Düşük düzeyde nesne denetimi ve gerçek bir nesne odaklı model içeren, genişletilmiş ve geliştirilmiş bir uygulama programlama arabirimi (API)
- XML için ECMAScript (E4X) belirtimini (ECMA-357 sürüm 2) esas alan XML API'si. E4X, dilin yerel veri türü olarak ECMAScript'e XML ekleyen bir dil uzantısıdır.
- Belge Nesnesi Modeli (DOM) Düzey 3 Olaylar Belirtimi'ni esas alan bir olay modeli

ActionScript 3.0'ın avantajları

ActionScript 3.0, önceki ActionScript sürümlerinin komut dosyası oluşturma yeteneklerinden çok daha fazlasını sunar. Büyük veri kümeleri ve nesne tabanlı, yeniden kullanılabilir kod tabanları ile oldukça karmaşık uygulamaların oluşturulmasını kolaylaştırmak üzere tasarlanmıştır. ActionScript 3.0, Adobe Flash Player'da çalışan içerik için gerekli değildir. Ancak, yalnızca AVM2'de (ActionScript sanal makinesi) mevcut performans geliştirmelerine giden bir yol açar. ActionScript 3.0 kodu, eski ActionScript kodundan on kata kadar daha hızlı çalışabilir.

Eski ActionScript Virtual Machine sürümü olan AVM1, ActionScript 1.0 ve ActionScript 2.0 kodunu çalıştırır. Flash Player 9 ve 10 geriye doğru uyumluluk için AVM1'i destekler.

ActionScript 3.0'daki yenilikler

ActionScript 3.0, ActionScript 1.0 ve 2.0'a benzeyen birçok sınıf ve özellik içerir. Ancak, ActionScript 3.0 mimari ve kavramsal açıdan önceki ActionScript sürümlerinden farklıdır. ActionScript 3.0'daki geliştirmeler arasında, çekirdek dilin yeni özellikleri ve düşük düzeyli nesneler üzerinde daha yüksek denetim sağlayan gelişmiş API'si yer alır.

Çekirdek dil özellikleri

Çekirdek dil, programlama dilinin deyimler, ifadeler, koşullar, döngüler ve türler gibi temel bina bloklarını tanımlar. ActionScript 3.0, geliştirme işlemini hızlandıran birçok özellik içerir.

Çalışma zamanı istisnaları

ActionScript 3.0, önceki ActionScript sürümlerinden daha çok hata koşulu bildirir. Yaygın hata koşulları için çalışma zamanı istisnaları kullanılarak hata ayıklama deneyimini geliştirir ve hataları daha güçlü şekilde işleyen uygulamalar geliştirmenizi sağlar. Çalışma zamanı hataları, kaynak dosya ve satır numarası bilgilerini ek açıklama olarak veren yığın izleri sağlayarak hızlı şekilde hataların yerini belirlemenize yardımcı olur.

Çalışma zamanı türleri

ActionScript 3.0'da tür bilgisi çalışma zamanında korunur. Bu bilgi, çalışma zamanı türü denetimleri gerçekleştirmek için kullanılır ve sistemin tür güvenliliğini geliştirir. Tür bilgileri, aynı zamanda yerel makine temsillerinde değişkenleri temsil etmek için de kullanılarak performansı artırır ve bellek kullanımını azaltır. Karşılaştırma yoluyla, ActionScript 2.0'da tür ek açıklamaları çoğunlukla geliştirici yardımıdır ve tüm değerler dinamik olarak çalışma zamanında yazılır.

Mühürlenmiş sınıflar

ActionScript 3.0, mühürlenmiş sınıf kavramını içerir. Mühürlenmiş bir sınıf, derleme zamanında tanımlanan yalnızca sabit özellikler ve yöntemler kümesine sahiptir; bu sınıfa ek özellikler ve yöntemler eklenemez. Bir sınıfın çalışma zamanında değiştirilememesi, daha sıkı derleme zamanı denetimine yol açar ve bu da daha sağlam programlar oluşturulmasını sağlar. Buna ek olarak, nesne örneklerinin her biri için dahili bir karma tablo gerektirmeyerek bellek kullanımını da azaltır. `dynamic` anahtar sözcüğünün kullanılmasıyla dinamik sınıflar da mümkündür. ActionScript 3.0'daki tüm sınıflar varsayılan olarak mühürlenmiştir ancak `dynamic` anahtar sözcüğüyle bu sınıfların dinamik olduğu bildirilebilir.

Yöntem kapanışı

ActionScript 3.0, yöntem kapanışının otomatik olarak orijinal nesne örneğini hatırlamasına olanak sağlar. Bu özellik, olay işlemesi için kullanışlıdır. ActionScript 2.0'da, yöntem kapanışları hangi nesne örneğinden ayıklandıkları bilgisini hatırlamaz ve bu da yöntem kapanışı çağrıldığında beklenmeyen bir davranış oluşmasına neden olur.

XML için ECMAScript (E4X)

ActionScript 3.0, en son ECMA-357 olarak standartlaştırılmış olan XML için ECMAScript (E4X) uygular. E4X, XML'in işlenmesi için doğal ve akıcı bir dil yapıları kümesi sunar. Geleneksel XML ayrıştırma API'lerinin tersine, E4X ile XML, dilin yerel bir veri türüymüş gibi hareket eder. E4X, ihtiyaç duyulan kod miktarını büyük ölçüde azaltarak XML'i işleyen uygulamaların geliştirilmesini kolaylaştırır.

ECMA E4X belirtimini görüntülemek için www.ecma-international.org adresine gidin.

Normal ifadeler

ActionScript 3.0, hızlı şekilde dizeleri arayabilmeniz ve işleyebilmeniz amacıyla normal ifadeler için yerel destek içerir. ECMAScript (ECMA-262) sürüm 3 dil belirtiminde belirtildiği şekilde ActionScript 3.0, normal ifadeler için destek uygular.

Ad alanları

Ad alanları, bildirimlerin görünürlüğünü (`public`, `private`, `protected`) kontrol etmek için kullanılan geleneksel erişim belirticilerine benzer. Bunlar, seçtiğiniz adlara sahip olabilen özel erişim belirticileri olarak çalışır. Çakışmaları önlemek için, ad alanlarında bir Universal Resource Identifier (URI) bulunur ve E4X ile çalıştığınızda XML ad alanlarını temsil etmek için de ad alanları kullanılır.

Yeni ilkel türler

ActionScript 3.0 üç adet sayısal tür içerir: Number, int ve uint. Number çift kesinlikli, kayan nokta sayısını temsil eder. int türü, ActionScript'in CPU için hızlı tam sayı matematik yeteneklerinden faydalanmasına olanak sağlayan 32-bit işaretli bir tam sayıdır. int türü, döngü sayaçları ve tam sayıların kullanıldığı değişkenler için kullanışlıdır. uint türü, RGB renk değerleri, bayt sayıları ve daha fazlası için kullanışlı olan işaretsiz, 32-bit tam sayı türüdür. Tam tersine, ActionScript 2.0 yalnızca tek bir sayısal değere sahiptir: Number.

API özellikleri

ActionScript 3.0'daki API'ler, düşük bir düzeydeki nesneleri kontrol etmenize olanak sağlayan birçok sınıf içerir. Dil mimarisi, önceki sürümlere göre daha sezgisel olacak şekilde tasarlanmıştır. Ayrıntılı olarak incelemek için çok fazla sınıf olsa da, bazı önemli farklılıkların önemi yoktur.

DOM3 olay modeli

Document Object Model Level 3 olay modeli (DOM3), olay mesajlarını oluşturmanın ve işlemenin standart bir yolunu sağlar. Bu olay modeli uygulamalardaki nesnelerin etkileşimde ve iletişimde bulunma, durumlarını koruma ve değişikliğe cevap vermelerine izin verecek şekilde tasarlanmıştır. ActionScript 3.0 olay modeli World Wide Web Consortium DOM Level 3 Events Belirtimi'nin ardından modellenmiştir. Bu model, ActionScript'in önceki sürümlerinde mevcut olay sistemlerinden daha temiz ve daha etkili bir mekanizma sunar.

Olaylar ve hata olayları, flash.events paketinde bulunur. Flash Professional bileşenleri ve Flex çerçevesi aynı olay modelini kullanır, böylece olay sistemi Flash Platform'da birleşir.

Görüntüleme listesi API'si

Görüntüleme listesine (uygulamadaki görsel öğeleri içeren ağaç) erişme API'si, görsel ilkel öğelerle çalışmaya yönelik sınıfları içerir.

Sprite sınıfı, kullanıcı arabirimi bileşenleri gibi görsel öğeler için bir temel sınıf olmak için tasarlanmış hafif bir yapı taşıdır. Shape sınıfı ham vektör şekillerini temsil eder. Bu sınıflar `new` operatörüyle normal şeilde başlatılabilir ve herhangi bir zamanda dinamik olarak yeniden üst öğeye sahip olabilir.

Derinlik yönetimi otomatiktir. Nesnelerin yığınlanma sırasının belirtilmesi ve yönetilmesi için yöntemler sağlanır.

Dinamik verileri ve içerikleri işleme

ActionScript 3.0, uygulamanızda varlıkların ve verilerin yüklenmesi için tüm API'de sezgisel ve tutarlı olan mekanizmalar içerir. Loader sınıfı, SWF dosyalarının ve görüntü varlıklarının yüklenmesi için tek bir mekanizma ve yüklenen içerikle ilgili ayrıntılı bilgilere erişme yolu sağlar. URLLoader sınıfı, veri tabanlı uygulamalarda metin ve ikili verilerin yüklenmesi için ayrı bir mekanizma sağlar. Socket sınıfı, sunucu soketlerine herhangi bir formatta ikili verileri okuyup yazmak için bir araç sağlar.

Düşük düzeyli veri erişimi

Çeşitli API'ler veriye düşük düzeyli erişim sağlar. URLStream sınıfı, indirilen veriler için, indirme sırasında verilere ham ikili veriler olarak erişilmesini sağlar. ByteArray sınıfı, ikili verilerle okumayı, yazmayı ve çalışmayı eniyileştirmenize olanak sağlar. Sound API'si, SoundChannel ve SoundMixer sınıfları üzerinden ayrıntılı ses denetimi sağlar. Güvenlik API'leri, SWF dosyasının veya yüklenen içeriğin güvenlik ayrıcalıkları hakkında bilgi sağlayarak güvenlik hatalarını işlemenize olanak tanır.

Metinle çalışma

ActionScript 3.0, tüm metinle ilgili API'ler için bir flash.text paketi içerir. TextLineMetrics sınıfı, bir metin alanındaki metin satırı için ayrıntılı ölçütler sağlar; ActionScript 2.0'daki `TextFormat.getTextExtent()` öğesinin yerini alır. TextField sınıfı, bir metin satırı veya bir metin satırındaki tek bir karakter hakkında belirli bilgiler sağlayabilen düşük düzeyli yöntemler içerir. Örneğin, `getCharBoundaries())` yöntemi bir karakterin sınırlama kutusunu temsil eden bir dikdörtgen döndürür. `getCharIndexAtPoint()` yöntemi belirli bir noktadaki karakterin dizinini döndürür. `getFirstCharInParagraph()` yöntemi paragraftaki ilk karakterin dizinini döndürür. Satır düzeyindeki yöntemler arasında, belirtilen bir metin satırındaki karakterlerin sayısını döndüren `getLineLength()` ve belirtilen satırın metnini döndüren `getLineText()` yer alır. Font sınıfı, SWF dosyalarındaki gömülü fontların yönetilmesi için araçlar sağlar.

Metin üzerindeki daha düşük düzeyli denetimler için, flash.text.engine paketindeki sınıflar Flash Text Engine'i oluşturur. Bu sınıflar metin üzerinde düşük düzeyli denetim sağlar ve metin çerçeveleri ve bileşenleri oluşturmak için tasarlanmıştır.

Bölüm 2: ActionScript ile çalışmaya başlama

Programlama temelleri

ActionScript bir programlama dili olduğundan, ilk önce birkaç genel bilgisayar programlama kavramını anlamanız, ActionScript'i öğrenmenizi kolaylaşacaktır.

Bilgisayar programları ne yapar

Öncelikle, bilgisayar programının ne olduğuna ve ne yaptığına dair kavramsal bir fikir edinilmesi yardımcı olacaktır. Bilgisayar programının iki yönü vardır:

- Program, bilgisayarın gerçekleştirmesi için tasarlanmış talimatlar veya adımlar serisidir.
- Her adım bazı bilgi veya verilerin işlenmesini içerir.

Genel anlamda bilgisayar programı, bilgisayara verdiğiniz ve bilgisayar tarafından birer birer gerçekleştirilen adım adım talimatlar listesidir. Talimatların her birine *deyim* denir. ActionScript'te, her deyim sonunda bir noktalı virgül ile yazılır.

Temelde, bir programdaki talimatın yaptığı tüm şey, bilgisayarın belleğinde saklanan bazı veri bitlerini işlemekten ibarettir. Buna basit bir örnek olarak, bilgisayara iki sayıyı eklemesi ve sonucu belleğinde depolaması talimatını verme işlemi gösterilebilir. Daha karmaşık bir örnek vermek gerekirse, ekranda çizili bir dikdörtgen olduğunu ve bu dikdörtgeni başka bir yere taşımak için bir program yazmak istediğinizi varsayalım. Bilgisayar, dikdörtgenle ilgili belirli bilgileri hatırlar, örn. dikdörtgenin bulunduğu x, y koordinatları, genişliği, uzunluğu, rengi vb. Bu bilgi bitlerinin her biri bilgisayarın belleğinde bir yerde saklanır. Dikdörtgeni farklı bir konuma taşıyacak bir program "x koordinatını 200 olarak değiştir; y koordinatını 150 olarak değiştir" gibi adımlara sahip olurdu. Başka bir şekilde ifade etmek gerekirse, x ve y koordinatları için yeni değerler belirlerdi. Arka planda, bilgisayar bu numaraları gerçekten bilgisayar ekranında görüntülenen görüntüye dönüştürmek için bu veri üzerinde işlem yapar. Ancak temel ayrıntı düzeyinde, "ekran üzerinde bir dikdörtgen taşıma" sürecinin, bilgisayarın belleğindeki veri bitlerini değiştirme işlemini içerdiğini bilmeniz yeterlidir.

Değişkenler ve sabitler

Programlama genellikle bilgisayarın belleğindeki bilgileri değiştirmeyi içerir. Sonuç olarak, bir programdaki bilgi parçasını temsil etmek için bir yolun olması önemlidir. *Değişken*, bilgisayar belleğindeki bir değeri temsil eden addır. Değerleri işlemek için deyimler yazdığınızda, değerin yerine değişkenin adını yazarsınız. Bilgisayar değişken adını programınızda her gördüğünde, belleğine bakar ve orada bulduğu değeri kullanır. Örneğin, her biri bir sayı içeren, `value1` ve `value2` adında iki değişkeniniz varsa, bu iki sayıyı eklemek için şu deyimi yazabilirsiniz:

```
value1 + value2
```

Bilgisayar adımları uygularken, her değişkendeki değerlere bakar ve bunları birbirine ekler.

ActionScript 3.0'da, bir değişken üç farklı bölümden oluşur:

- Değişkenin adı
- Değişkende saklanabilen veri türü
- Bilgisayarın belleğinde saklanan gerçek değer

Bilgisayarın, değerin yer tutucusu olarak adı nasıl kullandığını gördünüz. Veri türü de önemlidir. ActionScript'te bir değişken oluşturduğunuzda, tutması gereken belirli veri türünü belirlersiniz. Bu noktadan sonra, programın talimatları değişkende yalnızca o veri türünü depolayabilir. Veri türüyle ilişkili belirli özellikleri kullanarak değeri işleyebilirsiniz. ActionScript'te, bir değişken oluşturmak için (değişkeni *bildirmek* de denilebilir), `var` deyimini kullanırsınız:

```
var value1:Number;
```

Bu örnek bilgisayara `value1` adında, yalnızca Number verilerini tutabilen bir değişken oluşturmasını bildirir. ("Number" ActionScript'te tanımlanan belirli bir veri türüdür.) Bir değeri hemen değişkende de saklayabilirsiniz:

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

Adobe Flash Professional
Flash Professional'da bir değişken bildirmenin başka bir yolu vardır. Sahne Alanı'na bir film klibi sembolü, düğme sembolü veya metin alanı yerleştirdiğinizde, Özellik denetçisinde buna bir örnek adı verebilirsiniz. Flash Professional arka planda, örnek adıyla aynı adı taşıyan bir değişken oluşturur. Bu adı Sahne Alanı öğesini temsil etmesi için ActionScript kodunda kullanabilirsiniz. Örneğin, Sahne Alanı'nda bir film klibi sembolü olduğunu ve ona `rocketShip` örnek adını verdiğinizi varsayın. AdobeScript kodunda `rocketShip` değişkenini her kullandığınızda aslında o film klibini işlersiniz.

Bir *sabit*, bir değişkene benzer. Belirli bir veri türüyle, bilgisayarın belleğinde bir değer temsil eden bir addır. Tek farkı, sabite bir ActionScript uygulaması sırasında yalnızca bir kere değer atanabilmesidir. Sabitin değeri atandıktan sonra tüm uygulamada bu değer aynı kalır. Bir sabit bildirirken kullandığınız sözdizimi neredeyse bir değişken bildirirken kullandığınızla aynıdır. Tek fark, `var` anahtar sözcüğü yerine `const` anahtar sözcüğünü kullanmanızdır.

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

Bir proje boyunca birden çok yerde kullanılan ve normal koşullarda değişmeyen bir değeri tanımlamak için bir sabit kullanılabilir. Değişmez değer yerine bir sabit kullanılması, kodunuzu daha okunaklı hale getirir. Örneğin, aynı kodun iki sürümünü düşünün. Bunlardan biri, bir fiyatı `SALES_TAX_RATE` ile çarpar. Diğeri ise fiyatı `0,07` ile çarpar. `SALES_TAX_RATE` sabitini kullanan sürümü kavramak daha kolaydır. Ek olarak, sabit tarafından tanımlanan değerin değiştiğini varsayın. Projeniz boyunca bu değeri temsil etmesi için bir sabit kullanırsanız, değeri bir yerde değiştirebilirsiniz (sabit bildirimi). Tam tersine, sabit kodlanmış değişmez değerler kullanırsanız, değeri çeşitli yerlerde değiştirmek zorunda kalırsınız.

Veri türleri

ActionScript'te, oluşturduğunuz değişkenlerin veri türü olarak kullanabileceğiniz birçok veri türü vardır. Bu veri türlerinden bazıları "basit" veya "temel" veri türleri olarak değerlendirilebilir:

- Dize: bir ad veya kitabın bir bölümü gibi, metin değeri
- Sayısal: ActionScript 3.0, sayısal veriler için üç özel veri türü içerir:
  - Sayı: kesirli veya kesirsiz sayılar da dahil olmak üzere herhangi bir sayısal değer
  - int: bir tam sayı (kesirsiz bir tam sayı)
  - uint: “işaretsiz” tam sayı, başka bir deyişle negatif olamayan bütün bir sayı
- Boolean: bir düğmenin etkin olup olmadığı veya iki değerin eşit olup olmadığı gibi, doğru veya yanlış değeri

Basit veri türleri tek bir bilgiyi temsil eder: örneğin, tek bir sayı veya tek bir metin sırası Ancak, ActionScript'te tanımlı birçok veri türü karmaşıktır. Tek bir kaptaki bir değerler setini temsil ederler. Örneğin, Date veri türüne sahip bir değişken, tek bir değeri (tek bir zamanı) temsil eder. Ancak, bu değer birçok değer olarak temsil edilir: gün, ay, yıl, saat, dakika, saniye vb. ve bunların her biri ayrı ayrı bir sayıdır. Tarih genellikle tek bir değer olarak düşünülür ve bir Date değişkeni oluşturularak tarih tek bir değer olarak alınabilir. Ancak, bilgisayar bunu tek bir tarihi tanımlayan değerler grubu olarak göz önünde bulundurur.

Programcıların tanımladığı veri türlerinin yanı sıra, yerleşik veri türlerinin çoğu da karmaşık veri türleridir. Tanıyabileceğiniz karmaşık veri türlerinden bazıları şunlardır:

- MovieClip: bir film klibi sembolü
- TextField: dinamik bir alan veya girdi metni alanı
- SimpleButton: bir düğme sembolü
- Date: tek bir zaman (tarih ve saat) hakkındaki bilgi

Sınıf ve nesne sözcükleri genellikle veri türü için eşanlamlı olarak kullanılır. *Sınıf*, basit bir şekilde bir veri türünün tanımıdır. "Example veri türünün tüm değişkenleri şu özelliklere sahiptir: A, B ve C." ifadesinde olduğu şekilde veri türünün tüm nesneleri için bir şablon gibidir. Diğer yandan bir *nesne* ise bir sınıfın gerçek bir örneğidir. Örneğin, veri türü MovieClip olan bir değişken bir MovieClip nesnesi olarak tanımlanabilir. Aşağıda, aynı şeyi söylemenin birkaç farklı yolu verilmiştir:

- `myVariable` değişkeninin veri türü Number'dır.
- `myVariable` değişkeni bir Number örneğidir.
- `myVariable` değişkeni bir Number nesnesidir.
- `myVariable` değişkeni, Number sınıfının bir örneğidir.

Nesnelerle çalışma

ActionScript, nesne odaklı programlama dili olarak bilinir. Nesne odaklı programlama, bir programlama yaklaşımıdır. Bir programdaki kodu nesneler kullanarak organize etmekten daha fazlası değildir.

Daha önce "bilgisayar programı" terimi bilgisayarın gerçekleştirdiği bir dizi adım veya talimat olarak tanımlanmıştı. Bu durumda kavramsal olarak, bilgisayar programını tek bir uzun talimat listesi olarak düşünebilirsiniz. Ancak, nesne odaklı programlamada, program talimatları farklı nesneler arasında bölünmüştür. Kod işlevsellik öbekleri olarak gruplanır, böylece ilişkili işlevsellik türleri veya ilişkili bilgiler bir kapta gruplanır.

Adobe Flash Professional

Flash Professional'da sembollerle çalıştıysanız, nesnelerle de çalışmaya hazırsınız demektir. Bir dikdörtgen çizimi gibi bir film klibi sembolü tanımladığınızı ve bunun bir kopyasını Sahne Alanı'na yerleştirdiğinizi varsayın. Bu film klibi aynı zamanda (gerçekten) ActionScript'te bir nesnedir; MovieClip sınıfının bir örneğidir.

Film klibinin değiştirebileceğiniz çeşitli özellikleri vardır. Seçildiğinde, Özellik denetçisinden değerleri değiştirebilirsiniz. X koordinatı veya genişliği gibi. Alfasını (saydamlık) değiştirme veya alt gölge filtresi uygulama gibi çeşitli renk ayarlamaları da yapabilirsiniz. Diğer Flash Professional araçları, dikdörtgeni döndürmek için Serbest Dönüştürme aracının kullanılması gibi daha fazla değişiklik yapmanıza olanak tanır. Flash Professional'da bir film klibi sembolünü değiştirebileceğiniz tüm bu yollar ActionScript'te de mevcuttur. ActionScript'te bir film klibini, MovieClip nesnesi olarak adlandırılmış tek bir kümede bir araya getirilmiş veri parçalarını değiştirerek değiştirebilirsiniz.

ActionScript nesne odaklı programlamada, herhangi bir sınıfın içerebileceği üç özellik türü vardır:

- Özellikler
- Yöntemler
- Olaylar

Bu öğeler, program tarafından kullanılan verileri yönetmek ve hangi eylemlerin ne sırada yapılacağına karar vermek için kullanılır.

Özellikler

Özellik, bir nesnede kümelenmiş olan verilerden birini temsil eder. Bir örnek şarkı nesnesi, `artist` ve `title` adında özelliklere sahip olabilir; MovieClip sınıfı `rotation`, `x`, `width` ve `alpha` gibi özelliklere sahiptir. Özellikler üzerinde ayrı ayrı değerler şeklinde çalışırsınız. Aslında, özellikleri bir nesnede bulunan "alt" değişkenler olarak düşünebilirsiniz.

Aşağıda, özellik kullanan birkaç ActionScript kodu örnekleri verilmiştir. Bu kod satırı, `square` adındaki MovieClip öğesini 100 piksel x koordinatına taşır:

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

Bu kod, `triangle` MovieClip öğesinin dönüşüyle eşleşecek şekilde `square` MovieClip öğesinin dönmesini sağlamak için rotation özelliğini kullanır:

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

Bu kod, `square` MovieClip öğesinin eski halinden bir buçuk kat daha geniş olmasını sağlayacak şekilde yatay ölçeğini değiştirir:

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

Ortak yapıya dikkat edin: nesnenin adı sırayla nesnenin adını (`square`, `triangle`), bir nokta işaretini (`.`) ve özelliğin adını (`x`, `rotation`, `scaleX`) içerir. *Nokta operatörü* olarak da bilinen nokta işareti, bir nesnenin alt öğelerinden birine erişmekte olduğunu belirtmek için kullanılır. Tüm yapı olduğu gibi "değişken adı-nokta-özellik adı", tek bir değişken gibi, bilgisayar belleğindeki tek bir değerin adı olarak kullanılır.

Yöntemler

Yöntem bir nesnenin gerçekleştirebileceği bir eylemdir. Örneğin, Flash Professional'da birkaç anahtar karesi ve zaman çizelgesinde animasyon olan bir film klibi sembolü yaptığınızı düşünün. Bu film klibi oynatılabilir, durdurulabilir veya oynatma kafasını belirli bir kareye getirmek için talimat verilebilir.

Bu kod, `shortFilm` adındaki MovieClip öğesine oynatmayı başlatmasını bildirir:

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

Bu satır, `shortFilm` adındaki MovieClip öğesinin oynatmayı durdurmasını sağlar (oynatma kafası, video duraklatılmış gibi yerinde durdurulur):

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

Bu kod, `shortFilm` adındaki MovieClip öğesinin oynatma kafasını Kare 1'e taşıyıp oynatmayı durdurmasını sağlar (videoyu geri sarmak gibi):

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

Yöntemlere de, tıpkı özellikler gibi sırayla nesnenin adı (bir değişken), ardından nokta işareti, yöntemin adı ve parantez işaretleri yazılarak erişilir. Parantezler, yöntemi *çağırdığınızı* veya başka bir deyişle nesneye o eylemi gerçekleştirmesini bildirdiğinizi belirtmenin yoludur. Bazen eylemi gerçekleştirmek için gerekli olan ek bilgileri iletmenin bir yolu olarak, değerler (veya değişkenler) parantez içine yerleştirilir. Bu değerler, yöntem *parametreleri* olarak bilinir. Örneğin, `gotoAndStop()` yönteminin hangi kareye gidileceği hakkında bilgiye ihtiyacı vardır, bu nedenle parantezin içinde tek bir parametre olmasını gerektirir. `play()` ve `stop()` gibi diğer yöntemler kendinden açıklayıcıdır ve bu nedenle de fazladan bilgi gerektirmez. Ancak yine de parantez içinde yazılır.

Özelliklerden (ve değişkenlerden) farklı olarak, yöntemler değer yer tutucuları olarak kullanılmaz. Ancak bazı yöntemler hesaplamalar gerçekleştirebilir ve bir değişken olarak kullanılabilecek bir sonuç döndürebilir. Örneğin, Number sınıfının `toString()` yöntemi, sayısal değeri metin olarak temsil edilen haline dönüştürür:

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

Örneğin, Bir Number değişkeninin değerini ekranda bir metin alanında görüntülemek isterseniz, `toString()` yöntemini kullanırsınız. TextField sınıfının `text` özelliği bir Dize olarak tanımlanır, böylece yalnızca metin değerleri içerebilir. (`property` metni ekranda görüntülenen gerçek metin içeriğini temsil eder). Kodun bu satırı `numericData` değişkenindeki sayısal değeri metne dönüştürür. Daha sonra değerin ekranda `calculatorDisplay` adlı TextField nesnesinde görünmesini sağlar:

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

Olaylar

Bilgisayar programı, bilgisayarın adım adım gerçekleştirdiği bir talimatlar dizisidir. Bazı basit bilgisayar programları, bilgisayarın gerçekleştirdiği ve programı sona erdiren birkaç adımdan fazlasını içermez. Ancak, ActionScript programları sürekli çalışacak ve kullanıcı girdisinin veya başka şeylerin oluşmasını bekleyecek şekilde tasarlanmıştır. Olaylar, bilgisayarın hangi talimatları ne zaman gerçekleştireceğini belirleyen mekanizmadır.

Temel olarak *olaylar*, ActionScript'in farkında olduğu ve yanıt verdiği, gerçekleşen şeylerdir. Kullanıcının bir düğmeyi tıklatması veya klavyede bir tuşa basması gibi, çoğu olay kullanıcı etkileşimiyle ilişkilidir. Ayrıca başka olay türleri de vardır. Örneğin, harici bir görüntüyü yüklemek için ActionScript'i kullanırsanız, görüntü yüklemesinin bittiğini size bildiren bir olay vardır. Bir ActionScript programı çalışırken, kavramsal olarak yalnızca belirli olayların meydana gelmesini bekler. Bunlar gerçekleştiğinde, o olaylar için belirlediğiniz ActionScript kodu çalışır.

Temel olay işleme

Belirli olaylara yanıt olarak gerçekleştirilecek belirli eylemleri belirtme tekniği, *olay işleme* olarak bilinir. Olay işleme gerçekleştirmek için ActionScript kodu yazarken tanımlamanız gereken üç önemli öğe vardır:

- Olay kaynağı: Olayın gerçekleşeceği nesne hangisidir? Örneğin, hangi düğme tıklatıldı veya hangi Loader nesnesi görüntüyü yüklüyor? Olay kaynağı aynı zamanda *olay hedefi* olarak da bilinir. Bilgisayarın olayı hedeflediği nesne olduğu için bu ada sahiptir (yani, olayın gerçekten meydana geldiği yerdir).
- Olay: Gerçekleşecek şey, yanıt vermek istediğiniz şey nedir? Birçok nesne çok sayıda olayı tetiklediğinden belirli bir olayın tanımlanması önemlidir.

- Yanıt: Olay meydana geldiğinde hangi adımların gerçekleştirilmesini istiyorsunuz?

Olayları işlemek için ActionScript kodunu her yazdığınızda, şu üç öğeyi gereklidir: Kod şu temel yapıyı takip eder (kalın harfle yazılmış öğeler belirli durumunuz için dolduracağınız yer tutuculardır):

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

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

Bu kod iki işlem gerçekleştirir. İlk olarak, olaya yanıt olarak gerçekleştirilmesini istediğiniz eylemleri belirtmenin bir yolu olan bir işlevi tanımlar. Daha sonra, kaynak nesnenin `addEventListener()` yöntemini çağırır. `addEventListener()` yöntemini çağırmak esasında işlevi belirli olaya "kaydeder". Olay meydana geldiğinde, işlevin eylemleri gerçekleştirilir. Bu parçaların her birini daha ayrıntılı şekilde ele alın.

İşlev, eylemleri gerçekleştirmek üzere kısa yol adı gibi tek bir ad ile eylemleri bir arada gruplandırmanız için bir yol sağlar. Belirli bir sınıfla ilişkide olmak zorunda olmaması dışında, işlev yönteme benzerdir. (Aslında, "yöntem" terimi belli bir sınıfla ilişkili olan bir işlev olarak tanımlanabilir.) Olay işleme için bir işlev oluştururken, işlevin adını seçersiniz (bu durumda `eventResponse` olarak adlandırılmıştır). Ayrıca tek bir parametre belirtirsiniz (bu örnekte `eventObject` adında). İşlev parametresinin belirtilmesi bir değişkenin belirtilmesine benzer, bu nedenle parametrenin veri türünü de belirtmeniz gerekir. (Bu örnekte, `EventType` parametrenin veri türüdür.)

Dinlemek istediğiniz her olay türünün kendisiyle ilişkilendirilmiş bir ActionScript sınıfı vardır. İşlev parametresi için belirttiğiniz veri türü her zaman yanıt vermek istediğiniz belirli bir olayın ilişkilendirilmiş sınıfıdır. Örneğin, `click` olayı (kullanıcı fareyle bir öğeyi tıklattığında tetiklenir), MouseEvent sınıfıyla ilişkilendirilir. `click` olayı için bir dinleyici işlevi yazmak üzere, MouseEvent veri türüne sahip bir parametreyle dinleyici işlevini tanımlarsınız. Son olarak, açma ve kapatma küme parantezleri arasına ({ ... }), olay gerçekleştiğinde bilgisayarın uygulamasını istediğiniz talimatları yazarsınız.

Olay işleme işlevi yazılıdır. Daha sonra olay kaynak nesnesine (olayın gerçekleştiği nesne, örneğin düğme) olay gerçekleştiğinde işlevinizi çağırmak istediğinizi bildirirsiniz. Bu nesnenin `addEventListener()` yöntemini çağırarak işlevinizi olay kaynak nesnesiyle kaydedersiniz (bir olaya sahip tüm nesnelerde aynı zamanda `addEventListener()` yöntemi de vardır). `addEventListener()` yöntemi iki parametre alır:

- İlk olarak, yanıt vermek istediğiniz belirli olayın adı. Her olay belirli bir sınıfla ilişkilidir. Her olay sınıfı özel bir değere sahiptir. Bunlar, olaylarından her biri için tanımlanmış benzersiz bir ad gibidir. Bu değeri ilk parametre için kullanırsınız.
- İkinci olarak, olay yanıtı işlevinizin adı. İşlev adının parametre olarak iletildiğinde parantez olmadan yazıldığını unutmayın.

Olay işleme süreci

Aşağıda, bir olay dinleyicisi oluşturduğunuzda gerçekleşen işlemin adım adım bir açıklaması yer almaktadır. Bu durumda, `myButton` adındaki bir nesne tıklatıldığında çağrılan bir dinleyici işlevinin oluşturulması örneği yer almaktadır.

Programcı tarafından yazılan gerçek kod şu şekildedir:

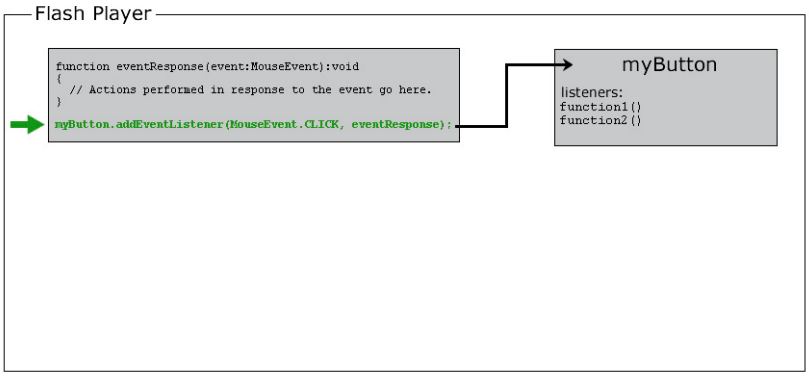
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

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

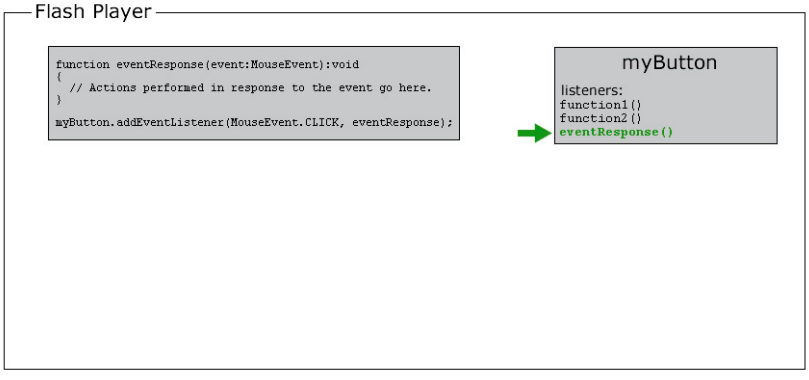
Çalışırken bu kodun gerçekten işleme şekli şöyledir:

1 SWF dosyası yüklendiğinde, bilgisayar, `eventResponse()` adında bir işlevin olduğunu not eder.

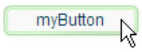
2 Daha sonra bilgisayar, kodu (daha açık olmak gerekirse, işlevde olmayan kod satırlarını) çalıştırır. Bu durumda yalnızca bir kod satırı vardır: Olay kaynağı nesnesinde (`myButton` adında) `addEventListener()` yöntemini çağırma ve parametre olarak `eventResponse` işlevini iletme.



Dahili olarak `myButton` öğesi, olaylarının her birini dinleyen işlevlerin bir listesini tutar. `addEventListener()` yöntemi çağrıldığında,`myButton` öğesi `eventResponse()` işlevini olay dinleyicileri listesinde depolar.

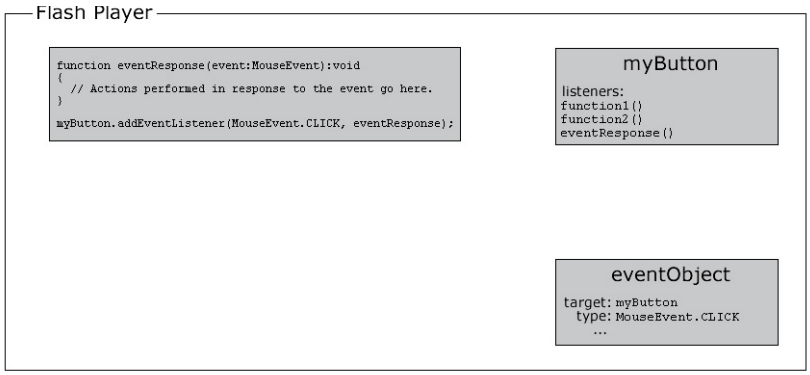


3 Bir noktada kullanıcı `myButton` nesnesini tıklatarak `click` olayını tetikler. (Kodda `MouseEvent.CLICK` olarak tanımlanmıştır.)

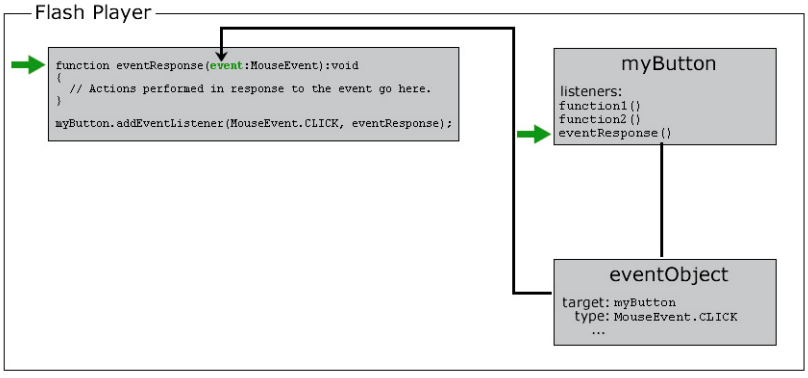


Bu noktada şunlar gerçekleşir:

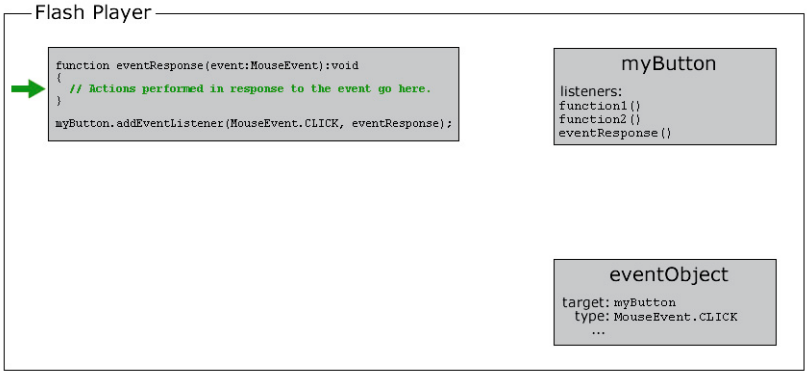
a Söz konusu olayla ilişkili sınıfın bir örneği olan bir nesne oluşturulur (bu örnekte MouseEvent). Birçok olay için, bu nesne Event sınıfının bir örneğidir. Fare olayları için, bu bir MouseEvent örneğidir. Diğer olaylar için, o olayla ilişkili bir sınıfın örneğidir. Oluşturulan nesne *olay nesnesi* olarak bilinir ve gerçekleşen olayla ilgili belirli bilgiler içerir: ne tür olay olduğu, nerede gerçekleştiği ve varsa diğer olaya özgü bilgiler.



b Daha sonra bilgisayar, `myButton` tarafından depolanan olay dinleyicileri listesine gözatar. Birer birer bu işlevlerde dolaşarak işlevlerin her birini çağırır ve olay nesnesini parametre olarak işleve iletir. `eventResponse()` işlevi, `myButton` öğesinin dinleyicilerinden biri olduğundan, bu işlemin bir parçası olarak bilgisayar `eventResponse()` işlevini çağırır.



c `eventResponse()` işlevi çağrıldığında, o işlevdeki kod çalıştırılır, böylece belirttiğiniz eylemler gerçekleştirilir.



Olay işleme örnekleri

Burada olay işleme kodu için daha somut örnekler bulabilirsiniz. Bu örnekler, size olay işleme kodu yazdığınızda karşınıza çıkabilecek yaygın olay öğeleri ve olası çeşitliliklerle ilgili bir fikir vermek içindir:

- Geçerli film klibini oynatmaya başlamak için bir düğmeyi tıklatma. Aşağıdaki örnekte, `playButton` düğmenin örnek adı ve `this` öğesi de "geçerli nesne" anlamına gelen özel bir addır:

```
this.stop();

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

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

- Metin alanındaki yazıyı algılama. Bu örnekte, `entryText` bir girdi metni alanı ve `outputText` ise bir dinamik metin alanıdır:

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

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

- URL'ye gitmek için bir düğmeyi tıklatma. Bu durumda, `linkButton` düğmenin örnek adıdır:

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

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

Nesne örnekleri oluşturma

ActionScript'te bir nesneyi kullanabilmeniz için öncelikle nesnenin varolması gerekir. Nesne oluşturma işleminin bir bölümünü değişkenin bildirilmesi oluşturur; ancak bir değişken belirtildiğinde, bilgisayarın belleğinde boş bir yer oluşturulur. Değişkeni kullanmadan veya işlemeden önce, her zaman değişkene gerçek bir değer atayın (bir nesne oluşturup bu nesneyi değişkende saklayın). Nesne oluşturma süreci *örnekleme* olarak bilinir. Başka bir deyişle, belirli bir sınıfın örneğini oluşturursunuz.

Nesne örneği oluşturmanın basit bir yolunda ActionScript hiç kullanılmaz. Flash Professional'da Sahne Alanı'na bir film klibi sembolü, düğme sembolü veya metin alanı yerleştirin ve bir örnek adı atayın. Flash Professional otomatik olarak o örnek adına sahip bir değişken belirler, bir nesne örneği oluşturur ve o nesneyi değişkende depolar. Benzer şekilde, Flex'te MXML'de bir bileşeni MXML etiketini kodlayarak veya bileşeni Flash Builder Design modunda editöre yerleştirerek oluşturursunuz. Bu bileşene bir kimlik atadığınızda, bu kimlik bileşen örneğini içeren bir ActionScript değişkeninin adı olur.

Ancak, her zaman bir nesneyi görsel olarak oluşturmak istemeyebilirsiniz ve görsel olmayan nesneler için de oluşturamazsınız. Yalnızca ActionScript'i kullanarak nesne örneği oluşturmanın bir kaç yolu daha vardır.

Birçok ActionScript veri türüyle, bir *değişmez ifade* (doğrudan ActionScript koduna yazılan bir değer) kullanarak bir örnek oluşturabilirsiniz. Aşağıda bazı örneklere yer verilmiştir:

- Değişmez sayısal değer (doğrudan sayı girilir):

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

- Değişmez String değeri (metin tırnak işareti içine alınır):

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

- Değişmez Boolean değeri (`true` veya `false` değişmez değerleri kullanılır):

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

- Değişmez Array değeri (virgülle ayrılmış değerler listesi köşeli ayraç içine alınır):

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

- Değişmez XML değeri (doğrudan XML girilir):

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

ActionScript ayrıca Array, RegExp, Object ve Function veri türleri için değişmez ifadeleri de tanımlar.

Burada gösterildiği gibi, herhangi bir veri türü için bir örnek oluşturmanın en yaygın yolu`new` operatörünü sınıf adıyla kullanmaktır:

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

`new` operatörü kullanılarak nesne oluşturulması genellikle "sınıfın yapıcısını çağırma" olarak tanımlanır. *Yapıcı*, bir sınıf örneği oluşturma işleminin parçası olarak çağrılan özel bir yöntemdir. Bu yolla bir örnek oluşturduğunuzda, sınıfın adından sonra parantez koymayı unutmayın. Bazen parantez içinde parametre değerleri belirlersiniz. Bu iki işlemi ayrıca bir yöntem çağırırken de gerçekleştirirsiniz.

Değişmez bir ifade kullanarak örnekler oluşturmanıza olanak sağlayan bu veri türleri için de bir nesne örneği oluşturmak üzere `new` operatörünü kullanabilirsiniz. Örneğin, bu iki kod satırı tamamen aynı işlemi gerçekleştirir:

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

Nesne oluşturmanın `new` *ClassName*`()` yolunun bilinmesi önemlidir. Birçok ActionScript veri türünün görsel temsili yoktur. Bu veriler sonuç olarak, Flash Professional Sahne Alanı'na veya Flash Builder'ın MXML editörünün Tasarım moduna bir öğe yerleştirilerek oluşturulamaz. ActionScript'te, bu veri türlerinin herhangi birinin örneğini `new` operatörünü kullanarak oluşturabilirsiniz.

Adobe Flash Professional

Flash Professional'da, `new` operatörü ayrıca Kütüphane'de tanımlanmış ancak Sahne Alanı'na yerleştirilmeyen bir film klibi sembolü örneği oluşturmak için de kullanılabilir.

Daha fazla Yardım konusu

Dizilerle çalışma

Normal ifadeler kullanma

ActionScript ile MovieClip nesneleri oluşturma

Ortak program öğeleri

ActionScript programı oluşturmak için kullandığınız bir kaç yapı taşı daha vardır.

Operatörler

Operatörler, hesaplamaları gerçekleştirmek için kullanılan özel sembollerdir (veya rastgele sözcüklerdir). Bunlar daha çok matematik işlemleri için kullanılır ve değerleri birbiriyle karşılaştırırken de kullanılabilir. Genel olarak, bir operatör bir veya birkaç değer kullanır ve tek bir sonuç "üretir". Örneğin:

- Toplama operatörü (`+`), iki değerini birbirine ekleyerek tek bir sayı ortaya çıkarır:

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

- Çarpma operatörü (`*`), bir değeri diğeriyle çarparak tek bir sayı ortaya çıkarır:

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

- Eşitlik operatörü (`==`), iki değerin eşit olup olmadığını görmek için iki değeri karşılaştırarak tek bir doğru veya yanlış (Boolean) değerini ortaya çıkarır:

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

  Burada gösterildiği gibi, eşitlik operatörü ve diğer "karşılaştırma" operatörleri, belirli talimatların uygulanıp uygulanmadığını belirlemek için en yayın şekilde `if` deyimiyle birlikte kullanılır.

Yorumlar

ActionScript yazarken, çoğu zaman kendiniz için notlar bırakmak isteyebilirsiniz. Örneğin, bazen belirli kod satırlarının nasıl çalıştığını veya belirli bir tercihi neden yaptığınızı açıklamak isteyebilirsiniz. *Kod yorumları*, bilgisayarınızın kodunuzda yok saydığı metinleri yazmak için kullanabileceğiniz araçlardır. ActionScript, iki tür yorum içerir:

- Tek satırlı yorum: Tek satırlı yorum, bir satırın herhangi bir yerine iki eğik çizgi yerleştirilerek belirlenir. Bilgisayar kesme işaretlerinden o satırın sonuna kadar olan tüm öğeleri yoksayar:

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

- Çok satırlı yorumlar: Çok satırlı yorum, bir yorum başlatma işaretçisini (`/*`), ardından yorum içeriğini ve sonra da yorum bitirme işaretçisini (`*/`) içerir. Bilgisayar yorumun kaç satır sürdüğüne bakmaksızın başlangıç ve bitiş işaretlerinin arasındaki tüm öğeleri yoksayar:

```
/*
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.
*/
```

Yorumların diğer bir yaygın kullanımı, geçici olarak bir veya daha fazla kod satırını "kapatmak" içindir. Örneğin, bir işlem gerçekleştirmenin farklı bir yolunu deniyorsanız yorumları kullanabilirsiniz. Yorumları, belirli bir ActionScript kodunun neden beklediğiniz şekilde çalışmadığını çözmek için de kullanabilirsiniz.

Akış denetimi

Bir programda birçok kez belirli eylemleri yinelemek, yalnızca belirli eylemleri gerçekleştirirken diğer eylemleri gerçekleştirmemek, belirli koşullara bağlı olarak alternatif eylemleri uygulamak vb. istersiniz. *Akış denetimi*, hangi eylemlerin gerçekleştirileceğiyle ilgili denetimdir. ActionScript'te birçok kullanılabilir akış denetimi öğesi türü vardır.

- İşlevler: İşlevler kısayollar gibidir. Tek bir ad altında bir eylemler dizisini gruplandırmanın bir yolunu sağlarlar ve hesaplamaları gerçekleştirmek için kullanılabilirler. İşlevler olayları işlemek için gereklidir ancak bir talimat dizisini gruplandırmak için genel bir araç olarak da kullanılır.
- Döngüler: Döngü yapıları, bilgisayarın belirli sayıda veya bazı koşullar değişene kadar gerçekleştirdiği bir talimatlar dizisi belirlemenize olanak tanır. Döngüler genellikle bilgisayar döngüde her çalıştığında değeri değişen bir değişkeni kullanarak birçok ilgili öğeyi işlemek için kullanılır.
- Koşullu deyimler: Koşullu deyimler yalnızca belirli koşullar altında gerçekleştirilen belirli talimatları belirlemeniz için bir yol sağlar. Ayrıca farklı koşullar için alternatif talimatlar sağlamak amacıyla da kullanılırlar. En yaygın koşul deyimi türü `if` deyimidir. `if` deyimi, parantezleri içindeki bir değeri veya ifadeyi kontrol eder. Değer `true` ise, küme parantezlerindeki kod satırları yürütülür. Aksi takdirde yok sayılırlar. Örneğin:

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

`if` deyiminin eşi olan `else` deyimi, koşul `true` olmadığında gerçekleştirilecek alternatif talimatları belirlemenize olanak tanır:

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

Örnek: Animasyon portföyü parçası (Flash Professional)

Bu örnek, ActionScript bit'lerini tam bir uygulamada nasıl bir araya getirebileceğini görmeniz için size ilk fırsatı sunmak üzere tasarlanmıştır. Animasyon portföyü varolan doğrusal bir animasyonu alıp, nasıl bazı küçük etkileşimli öğeleri ekleyebileceğinizin bir örneğidir. Örneğin, bir istemci için oluşturulmuş animasyonu bir çevrimiçi portföye dahil edebilirsiniz. Animasyona ekleyeceğiniz etkileşimli davranış, izleyenin tıklatabileceği iki düğmeyi içerir: bu düğmelerden biri animasyonu başlatmak ve biri de ayrı bir URL'ye (örn. portföy menüsü veya yazarın ana sayfası) gitmek içindir.

Bu parçayı oluşturma işlemi şu ana bölümlere ayrılabilir:

1. ActionScript ve etkileşimli öğeler eklemek için FLA dosyasını hazırlama.
2. Düğme oluşturma ve ekleme.
3. ActionScript kodu yazma.
4. Uygulamayı test etme.

Etkileşim eklemeye hazırlama

Animasyonunuza etkileşimli öğeler eklemeden önce, yeni içeriğinizin ekleneceği bazı konumlar oluşturarak FLA dosyasını ayarlamanız fayda sağlar. Bu görev, düğmelerin yerleştirileceği Sahne Alanı'nda gerçek alan oluşturmayı içerir. Ayrıca farklı öğeleri ayrı tutmak için FLA dosyasında “boşluk” oluşturmayı da içerir.

Etkileşimli öğeler eklemek üzere FLA'nızı ayarlamak için:

1. Tek ara hareket veya şekil arası gibi basit bir animasyon ile FLA dosyası oluşturun. Bir projede sunduğunuz bir animasyonu içeren FLA dosyasına zaten sahipseniz, o dosyayı açın ve yeni bir adla kaydedin.
2. İki düğmenin ekranın neresinde görüntülenmesini istediğinize karar verin. Bir düğme, animasyonu başlatmak, diğeri ise yazar portföyüne veya ana sayfaya bağlantı vermek içindir. Gerekirse, bu yeni içerik için Sahne Alanı'nda bazı boşlukları temizleyin veya Sahne Alanı'na bazı boşluklar ekleyin. Animasyonda yoksa, ilk karede bir başlangıç ekranı oluşturabilirsiniz. Bu durumda, büyük olasılıkla Kare 2'de veya daha sonraki bir noktada başlaması için animasyonu kaydırmak isteyebilirsiniz.
3. Zaman Çizelgesi'nde diğer katmanların üzerine yeni bir katman ekleyin ve bunu **düğmeler** olarak adlandırın. Bu katman, düğmeleri ekleyeceğiniz yerdir.
4. Düğmeler katmanının yukarısına yeni bir katman ekleyin ve bu katmana **eylemler** adını verin. Bu katman, uygulamanıza ActionScript kodunu ekleyeceğiniz yerdir.

Düğme oluşturma ve ekleme

Ardından, etkileşimli uygulamanın merkezini oluşturan düğmeleri oluşturup yerleştirirsiniz.

Düğmeleri oluşturup FLA dosyasına eklemek için:

1. Çizim araçlarını kullanarak, düğmeler katmanındaki birinci düğmenizin ("oynat" düğmesi) görsel görünümünü oluşturun. Örneğin, en üst kısmında metnin yer aldığı yatay bir oval çizin.
2. Seçim aracını kullanarak tek bir düğmenin tüm grafik parçalarını seçin.
3. Ana menüden Değiştir > Sembole Dönüştür seçeneklerini belirleyin.
4. İletişim kutusunda sembol türü olarak Düğme seçeneğini belirleyin ve sembole bir ad verip Tamam'ı tıklatın.
5. Düğme seçili durumdayken, Özellik denetçisinde düğmeye **playButton** örnek adını verin.
6. İzleyeni, yazarın ana sayfasına götüren düğmeyi oluşturmak için 1 ile 5 arasındaki adımları yineleyin. Bu düğmeye **homeButton** adını verin.

Kod yazma

Tümüne aynı yerden girilse de, bu uygulamanın ActionScript kodu üç işlev kümesine ayrılabilir. Kodun gerçekleştirdiği üç işlem şunlardır:

- SWF dosyası yüklendiği anda (oynatma kafası Kare 1'e girdiğinde) oynatma kafasını durdurma.
- Kullanıcı oynatma düğmesini tıklattığında SWF dosyasının oynatılmaya başlaması için bir olayı dinleme.
- Kullanıcı yazarın ana sayfası düğmesini tıklattığında tarayıcıyı uygun URL'ye göndermek için bir olayı dinleme.

Kare 1'e girdiğinde oynatma kafasını durdurmak için kod oluşturma:

1. Eylemler katmanının Kare 1'inde anahtar kareyi seçme.
2. Eylemler panelini açmak için, ana menüden Pencere > Eylemler seçeneklerini belirleyin.
3. Komut Dosyası bölmesinde şu kod girin:

```
stop();
```

Oynatma düğmesi tıklatıldığında animasyonu başlatmak üzere kod yazmak için:

1. Önceki adımlarda girilen kodun sonuna iki boş satır ekleyin.
2. Komut dosyasının alt kısmına şu kodu girin:

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

   Bu kod, `startMovie()` adında bir işlevi tanımlar. `startMovie()` çağrıldığında, ana zaman çizelgesinin oynatmayı başlatmasını sağlar.

3. Önceki adımda eklenen kodun ardından gelen satıra bu kod satırını girin:

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

   Bu kod satırı, `playButton` öğesinin `click` olayının dinleyicisi olarak `startMovie()` işlevini kaydeder. Başka bir deyişle, `playButton` adındaki düğme her tıklatıldığında `startMovie()` işlevinin çağrılmasını sağlar.

Ana sayfa düğmesi tıklatıldığında tarayıcıyı bir URL'ye göndermek üzere kod yazmak için:

1. Önceki adımlarda girilen kodun sonuna iki boş satır ekleyin.
2. Komut dosyasının alt kısmına bu kodu girin:

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

Bu kod, `gotoAuthorPage()` adında bir işlevi tanımlar. Bu işlev öncelikle http://example.com/ URL'sini temsil eden bir URLRequest örneği oluşturur. Daha sonra bu URL'yi `navigateToURL()` işlevine aktarır ve kullanıcı tarayıcısının bu URL'yi açmasına neden olur.

3 Önceki adımda eklenen kodun ardından gelen satıra bu kod satırını girin:

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

Bu kod satırı, `homeButton` öğesinin `click` olayının dinleyicisi olarak `gotoAuthorPage()` işlevini kaydeder. Başka bir deyişle, `homeButton` adındaki düğme her tıklatıldığında `gotoAuthorPage()` işlevinin çağrılmasını sağlar.

Uygulamayı test etme

Bu uygulama artık tamamen işlevseldir. Emin olmak için durumun böyle olup olmadığını test edelim.

Uygulamayı test etmek için:

1 Ana menüden, Kontrol Et > Filmi Test Et seçeneklerini belirleyin. Flash Professional, SWF dosyasını oluşturur ve bir Flash Player penceresinde açar.

2 Beklediğiniz işlemi gerçekleştirdiklerinden emin olmak için her iki düğmeyi de deneyin.

3 Düğmeler çalışmazsa, kontrol etmeniz gereken şeyler şunlardır:

- Düğmelerin belirgin örnek adları var mı?
- `addEventListener()` yöntemi çağrıları, düğmelerin örnek adlarıyla aynı adları mı kullanıyor?
- `addEventListener()` yöntemi çağrılarında doğru olay adları kullanılıyor mu?
- İşlevlerin her biri için doğru parametre belirtilmiş mi? (Her iki yöntem için de MouseEvent veri türünde tek bir parametre gerekir.)

Tüm bu hatalar ve meydana gelmesi olası diğer hataların çoğu bir hata iletisinin verilmesine yol açar. Hata iletisi, Filmi Test Et komutunu seçtiğinizde veya projeyi test ederken düğmeyi tıklattığınızda görüntülenebilir. Derleyici hataları (Filmi Test Et'i ilk seçtiğinizde meydana gelen) için Derleyici Hataları paneline bakın. İçerik oynatılırken, örneğin bir düğmeyi tıklattığınızda oluşan çalışma zamanı hataları için Çıktı Paneli'ne bakın.

ActionScript ile uygulamalar oluşturma

Uygulama oluşturmak için ActionScript yazma işlemi, sözdiziminin ve kullanacağınız sınıfların adlarının bilinmesinden daha fazlasını içerir. Birçok Flash Platform belgesi bu iki konuyu kapsar (sözdizimi ve ActionScript sınıflarını kullanma). Ancak, bir ActionScript uygulaması oluşturmak için şunları da bilmek isteyebilirsiniz:

- ActionScript yazmak için hangi programlar kullanılabilir?
- ActionScript kodu nasıl organize edilir?
- ActionScript kodu bir uygulamaya nasıl dahil edilebilir?
- Bir ActionScript uygulaması geliştirilirken hangi adımlar takip edilir?

Kodunuzu organize etme seçenekleri

Basit grafik animasyonlarından karmaşık istemci-sunucu işlemi işleme sistemlerine kadar her şeyi desteklemek için ActionScript 3.0'ı kullanabilirsiniz. Oluşturduğunuz uygulama türüne bağlı olarak, projenize ActionScript dahil etmenin bu farklı yollarından birini veya birkaçını kullanın.

Flash Professional zaman çizelgesinde karelerde kod saklama

Flash Professional'da, zaman çizelgesindeki herhangi bir kareye ActionScript kodu ekleyebilirsiniz. Film oynatılırken, oynatma kafası bu kareye girdiğinde bu kod çalıştırılır.

ActionScript kodunun karelere yerleştirilmesi, Flash Professional'daki yerleşik uygulamalara davranış eklemek için kolay bir yöntem sağlar. Ana zaman çizelgesinde herhangi bir kareye veya herhangi bir MovieClip sembolünün zaman çizelgesinde herhangi bir kareye kod ekleyebilirsiniz. Ancak bu esneklik bir maliyete de yol açar. Daha büyük uygulamalar oluşturduğunuzda, hangi karelerin hangi komut dosyalarını içerdiğinin takibini kaybetmek kolaylaşır. Bu karmaşık yapı, uygulamanın zamanla korunmasını zorlaştırabilir.

Geliştiricilerin çoğu, zaman çizelgesinin yalnızca birinci karesine veya Flash belgesindeki belirli bir katmana kod yerleştirerek Flash Professional'da ActionScript kodunun organize edilmesini kolaylaştırır. Kodunuzun ayrılması, Flash FLA dosyalarınızda kodun bulunmasını ve korunmasını kolaylaştırır. Ancak, aynı kod kopyalanıp yeni bir dosyaya yapıştırılmadan, başka bir Flash Professional projesinde kullanılamaz.

ActionScript kodunuzu gelecekteki Flash Professional projelerinizde kullanmayı kolaylaştırmak için, kodunuzu harici ActionScript dosyalarında saklayın (.as uzantısı olan metin dosyaları).

Flex MXML dosyalarına kod gömme

Flash Builder gibi bir Flex geliştirme ortamında, bir Flex MXML dosyasının içindeki `<fx:Script>` etiketine ActionScript kodu dahil edebilirsiniz. Ancak bu teknik, geniş projelere karmaşıklık getirebilir ve aynı kodu başka bir Flex projesinde kullanmayı zorlaştırabilir. Gelecekte ActionScript kodunuzu diğer Flex projelerinde kullanmayı kolaylaştırmak için, kodunuzu harici ActionScript dosyalarında saklayın.

Not: *Bir `<fx:Script>` etiketi için kaynak parametresi belirleyebilirsiniz. Kaynak parametresi kullanmak, doğrudan `<fx:Script>` etiketinde yazılmış gibi, ActionScript kodunu bir harici dosyadan "içe aktarmanıza" izin verir. Ancak kullandığınız kaynak dosyası kendi sınıfını tanımlayamaz ve bu da dosyanın yeniden kullanılabilirliğini sınırlar.*

ActionScript dosyalarında kod saklama

Projenizde önemli ActionScript kodu varsa, kodunuz en iyi şekilde ayrı ActionScript kaynak dosyalarında (.as uzantısına sahip metin dosyaları) organize edilir. ActionScript dosyası, uygulamanızda kullanılma amacına bağlı olarak, iki yoldan biri kullanılarak yapılandırılabilir.

- Yapılandırılmamış ActionScript kodu: Doğrudan bir zaman çizelgesi komut dosyasına veya MXML dosyasına girilmiş gibi yazılmış olan ve deyimleri ya da işlev tanımlarını içeren ActionScript kodu satırları.

  ActionScript'te `include` deyimi veya Flex MXML'de `<fx:Script>` etiketi kullanılarak bu şekilde yazılmış ActionScript'e erişilebilir. ActionScript `include` deyimi derleyiciye, belirli bir konum ve bir komut dosyasında verilmiş kapsam dahilinde olan harici bir ActionScript dosyasının içeriklerini dahil etmesini belirtir. Sonuç, kodun doğrudan girildiğinde ortaya çıkan sonuçla aynıdır. MXML dilinde, kaynak niteliği taşıyan bir `<fx:Script>` etiketinin kullanılması, derleyicinin uygulamada o anda yüklediği harici ActionScript'i tanımlar. Örneğin, aşağıdaki etiket, Box.as adında harici bir ActionScript dosyası yükler:

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

- ActionScript sınıfı tanımı: ActionScript sınıfının yöntemini ve özellik tanımlarını içeren bir tanımı.

Bir sınıf tanımladığınızda, sınıfın bir örneğini oluşturup, onun özelliklerini, yöntemini ve olaylarını kullanarak sınıftaki ActionScript koduna erişim sağlayabilirsiniz. Kendi sınıflarınızı kullanmanız herhangi bir yerleşik ActionScript sınıfını kullanmanızla aynıdır ve iki kısım gerektirir:

- ActionScript derleyicisinin nerede bulacağını bilmesi için, sınıfın tam adını belirtmek amacıyla `import` deyimini kullanın. Örneğin, ActionScript'te MovieClip sınıfını kullanmak için, sınıfın tam adını kullanarak ve paket ile sınıfını da dahil ederek sınıfı içe aktarın.

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

  Alternatif olarak, MovieClip sınıfını içeren paketi içe aktarabilirsiniz. Bu işlem paketteki her sınıf için ayrı `import` deyimleri yazılmasına eşdeğerdir:

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

  Bir sınıfı kodunuzda kullanabilmek için içe aktarılması gerektiği kuralının tek istisnası üst düzey sınıflardır. Bu sınıflar pakette tanımlanmamıştır.

- Özellikle sınıf adını kullananan kodu yazın. Örneğin, veri türü olarak o sınıfta olan bir değişken bildirin ve değişkende saklamak için sınıfın bir örneğini oluşturun. ActionScript kodunda bir sınıf adını kullanarak, derleyiciye o sınıfın tanımını yüklemesini bildirirsiniz. Örneğin, Box adında harici bir sınıf olduğunu düşünürsek, bu deyim Box sınıfının bir örneğini oluşturur:

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

  Derleyici, Box sınıfına olan başvuruyla ilk karşılaştığında, Box sınıfı tanımının konumunu belirleyebilmek için mevcut kaynak kodunu arar.

Doğru aracı seçme

ActionScript kodunuzu yazmak veya düzenlemek için birkaç araçtan birini (veya birden fazla aracı birlikte) kullanabilirsiniz.

Flash Builder

Adobe Flash Builder, Flex çerçeveli projeler veya çoğunlukla ActionScript kodundan oluşan projeler oluşturmak için kullanılan asıl araçtır. Flash Builder ayrıca, görsel mizanpaj ve MXML düzenleme özelliklerinin yanı sıra tam özellikli bir ActionScript editörüne de sahiptir. Flex ve yalnızca ActionScript projelerini oluşturmak için kullanılabilir. Flex, zengin bir önceden oluşturulmuş kullanıcı arabirimi denetimleri kümesi, esnek dinamik mizanpaj denetimleri ve harici veri kaynaklarıyla çalışılması ve harici verilerin kullanıcı arabirimi öğelerine bağlanmasına yönelik yerleşik merkezler içeren bir çok avantaj sağlar. Ancak, bu özellikleri sağlamak için ek kod gerektiğinden, Flex kullanan projeler Flex kullanmayan eşlerinden daha büyük bir SWF dosyası boyutuna sahiptir.

Flex ile tam donanımlı, veri tabanlı, zengin İnternet uygulamaları oluşturuyorsanız Flash Builder'ı kullanın. Tek bir araç içinde Action Script kodunuzu düzenlemek, MXML kodunuzu düzenlemek ve uygulamanızı görsel olarak düzenlemek istediğinizde bunu kullanın.

ActionScript ağırlıklı projeler oluşturan birçok Flash Professional kullanıcısı, Flash Professional'ı görsel varlıklar oluşturmak için, Flash Builder'ı ise ActionScript kodu için bir editör olarak kullanırlar.

Flash Professional

Flash Professional, grafik ve animasyon oluşturma özelliklerinin yanı sıra ActionScript koduyla çalışmak için araçlar içerir. Kod, FLA dosyasındaki öğelere veya harici yalnızca ActionScript dosyalarına eklenebilir. Flash Professional, önemli animasyon veya video içeren projeler için idealdir. Grafik varlıklarının çoğunu kendiniz oluşturmak istediğinizde değerlidir. ActionScript projenizi geliştirirken Flash Professional kullanmak için bir diğer neden, görsel varlıklar oluşturmak ve aynı uygulamada kod yazmaktır. Flash Professional önceden oluşturulmuş kullanıcı arabirimi bileşenleri de içerir. Bu bileşenleri daha küçük SWF dosya boyutu elde etmek ve görsel araçları projeniz için dış görünüm olarak kullanmak için kullanabilirsiniz.

Flash Professional, ActionScript kodunun yazılması için iki araç içerir:

- Eylemler paneli: FLA dosyasında çalışılırken kullanılabilir olan bu panel, zaman çizelgesinde karelere eklenmiş ActionScript kodu yazmanıza olanak sağlar.
- Komut dosyası penceresi: Komut dosyası penceresi, ActionScript (.as) kod dosyalarıyla çalışmak için adanmış bir metin düzenleyicisidir.

Üçüncü taraf ActionScript düzenleyici

ActionScript (.as) dosyaları basit metin dosyaları olarak saklandığından, düz metin dosyalarını düzenleyebilen herhangi bir program ActionScript dosyaları yazmak için kullanılabilir. Adobe'nin ActionScript ürünlerine ek olarak, ActionScript'e özgü yetenekler içeren birçok üçüncü taraf metin düzenleme programları da oluşturulmuştur. Herhangi bir metin düzenleyici programı kullanarak bir MXML dosyası veya ActionScript sınıfları yazabilirsiniz. Bundan sonra Flex SDK kullanarak bu dosyalardan bir uygulama oluşturabilirsiniz. Proje Flex kullanabilir veya bir yalnızca ActionScript uygulaması olabilir. Alternatif olarak, bazı geliştiriciler ActionScript sınıfları yazmak için Flash Builder veya bir üçüncü taraf ActionScript editörünü, grafik içeriği oluşturan Flash Professional ile birlikte kullanır.

Üçüncü taraf ActionScript editörünü kullanmak için nedenler şunları içerir:

- ActionScript kodunu ayrı bir programda yazmayı ve görsel öğeleri Flash Professional'da tasarlamayı tercih etmeniz.
- ActionScript olmayan programlama (örn. HTML sayfaları oluşturma veya başka bir programlama dilinde uygulamalar oluşturma) için bir uygulama kullanmanız. ActionScript kodlamanız için de aynı uygulamayı kullanmak istemeniz.
- Flash Professional veya Flex Builder olmadan Flex SDK kullanarak yalnızca ActionScript veya Flex projeleri oluşturmak istemeniz.

ActionScript'e özgü destek sağlayan dikkate değer kod düzenleyicilerinden bazıları şunlardır:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (ActionScript ve Flex paketleri ile)

ActionScript geliştirme işlemi

ActionScript projeniz büyük veya küçük olsa da, tasarlamak için bir süreç kullanmak ve uygulamanızı geliştirmek işinizi daha verimli ve etkili hale getirir. Aşağıdaki adımlar, ActionScript 3.0'ı kullanan bir uygulamanın oluşturulmasına yönelik temel bir geliştirme işlemini açıklamaktadır:

1 Uygulamanızı tasarlayın.

Uygulamanızı oluşturmaya başlamadan önce bir şekilde açıklayın.

2 ActionScript 3.0 kodunuzu oluşturun.

Flash Professional, Flash Builder, Dreamweaver veya bir metin düzenleyicisi kullanarak ActionScript kodu oluşturabilirsiniz.

3 Kodunuzu çalıştırmak için bir Flash veya Flex projesi oluşturun.

Flash Professional'da, bir FLA dosyası oluşturun, yayınlama ayarlarını ayarlayın, uygulamaya kullanıcı arabirimi bileşenlerini ve ActionScript koduna başvuru ekleyin. Flex'te uygulamayı tanımlayın, MXML kullanarak kullanıcı arabirimi bileşenlerini ve ActionScript koduna başvuru ekleyin.

4 ActionScript uygulamanızı yayınlayın ve test edin.

Uygulama testi, uygulamayı geliştirme ortamından çalıştırmanızı ve istediğiniz her işlemi gerçekleştirdiğinden emin olmanızı içerir.

Bu adımları aynı sırayla izlemek veya bir adım üzerinde çalışmadan önce bir önceki adımı tamamen bitirmek zorunda değilsiniz. Örneğin uygulamanızın bir ekranını tasarlayabilir (adım 1) ve daha sonra ActionScript kodu yazmadan (adım 2) ve test işlemini yapmadan (adım 4) önce, grafik düğmeleri vb. oluşturabilirsiniz (adım 3). Ya da bir kısmını tasarlayabilir ve sonra aynı anda tek bir düğme veya arabirim öğesi ekleyebilir ve her biri için ActionScript yazıp bu öğeler oluşturulduğunda da öğeleri test edebilirsiniz. Geliştirme sürecinin bu dört aşamasını hatırlamak faydalıdır. Ancak, gerçek dünya geliştirmesinde uygun olduğu şekilde aşamalar arasında gidip gelmek daha etkilidir.

Kendi sınıflarınızı oluşturma

Projelerinizde kullanılmak üzere sınıf oluşturulması işlemi yorucu görünebilir. Ancak sınıf oluşturulmasının daha zor bir kısmını sınıfın yönteminin, özelliklerin ve olayların tanımlanması görevi oluşturur.

Sınıf tasarlama stratejileri

Nesne tabanlı tasarım konusu karmaşıktır; tüm kariyerler, bu disiplinin akademik çalışmalarına ve profesyonel uygulamasına adanmıştır. Ancak yine de başlangıç aşamasında yardımcı olabilecek birkaç önerilen yaklaşım şunlardır.

1 Bu sınıf örneklerinin uygulamada oynadığı rolü düşünün. Genellikle nesneler bu üç rolden biri görevini görür:

- Değer nesnesi: Bu nesneler daha çok veri kapları görevi görür. Bunlar büyük olasılıkla, çeşitli özelliklere ve daha az yönteme sahiptir (veya bazen herhangi bir yönteme sahip değildir). Genelde, açıkça tanımlanmış öğelerin kod olarak temsilleridir. Örneğin, bir müzik oynatıcısı uygulaması tek bir gerçek dünya şarkısı temsil eden bir Song sınıfını ve kavramsal bir şarkı grubunu temsil eden Playlist sınıfını içerebilir.
- Görüntüleme nesnesi: Bunlar gerçekten ekranda görüntülenen nesnelerdir. Bu nesnelerin örnekleri arasında, açılır liste veya durum gösterimi gibi kullanıcı arabirimi öğeleri, video oyunundaki yaratıklar gibi grafiksel öğeler vb. yer alır.

- Uygulama yapısı: Bu nesneler, uygulamalar tarafından gerçekleştirilen mantık veya işlemede çok sayıda destekleyici rol oynar. Örneğin, bir biyoloji simülasyonunda bir nesneye belirli hesaplamaları yaptırabilirsiniz. Bir nesneye, bir müzik oynatıcısı uygulamasında arama denetimi ve ses düzeyi bilgisi arasındaki değerleri senkronize etme sorumluluğu verebilirsiniz. Başka bir nesne ise bir video oyunundaki kuralları yönetebilir. Ya da çizim uygulamasındaki kaydedilmiş bir resmi yüklemek için sınıf oluşturabilirsiniz.

2 Sınıfın ihtiyaç duyduğu belirli işlevleri belirleyin. Farklı işlev türleri genellikle sınıfın yöntemleri olur.

3 Sınıf bir değer nesnesi görevi görecek şekilde tasarlanmışsa, örneklerin içereceği verileri belirleyin. Bu öğeler, iyi özellik adaylarıdır.

4 Sınıfınız projeniz için özel olarak tasarlandığından, en önemli şey, uygulamanızın ihtiyaç duyduğu işlevselliği sağlamaktır. Bu soruları kendiniz için cevaplamaya çalışın:

- Uygulamanız hangi bilgileri saklıyor, izliyor ve işliyor? Bu soruları cevaplamak, ihtiyacınız olan herhangi bir değer nesnesini ve özelliği tanımlamanıza yardımcı olur.
- Uygulama ne tür eylemler gerçekleştiriyor? Örneğin, uygulama ilk yüklendiğinde, belirli bir düğme tıklatıldığında, bir filmin oynatılması durdurulduğunda vb. ne oluyor? Bunlar iyi yöntem adaylarıdır. "Eylemler" bireysel değerleri değiştirmeyi içerirse bunlar özellikler de olabilir.
- Herhangi bir eylem için, o eylemi gerçekleştirmek amacıyla ilgiler gereklidir Bu bilgiler yöntemin parametresi olur.
- Uygulama, işlemi gerçekleştirmek için ilerledikçe, sınıfınızda, uygulamanızın diğer bölümlerinin de bilmesi gereken hangi durumlar değişir? Bunlar iyi olay adaylarıdır.

5 Eklemek istediğiniz ek işlevsellikten yoksun olması dışında, ihtiyacınız olan nesneye benzer varolan bir nesne var mı? Alt sınıf oluşturmayı göz önünde bulundurun. (*Alt sınıf* tüm işlevselliğini tanımlamak yerine, varolan bir sınıfın işlevselliği üzerine kurulan bir sınıftır.) Örneğin, ekranda görsel bir nesne olan bir sınıf oluşturmak için, sınıfınızın temeli olarak varolan bir görüntü nesnesinin davranışını kullanın. Bu durumda, görüntü nesnesi (MovieClip veya Sprite gibi) *temel sınıf* olur ve sınıfınız bu sınıfı genişletir.

Sınıf için kod yazma

Sınıfınız için bir tasarımınız olduğunda veya en azından ne tür bilgi depoladığını ve ne tür eylem gerçekleştirdiği hakkında fikriniz olduğunda, bir sınıf yazmanın gerçek sözdizimi oldukça basittir.

Aşağıda, kendi ActionScript sınıfınızı oluşturmanız için gereken minimum adımlar verilmiştir:

1 ActionScript metin düzenleyici programında yeni bir metin belgesi açın.

2 Sınıfın adını tanımlamak için bir `class` deyimi girin. `class` deyimi eklemek için `public class` sözcüklerini ve ardından sınıfın adını girin. Sınıfın içeriklerini (yöntem ve özellik tanımları) barındırması için açılış ve kapanış küme parantezleri ekleyin. Örneğin:

```
public class MyClass
{
}
```

`public` sözcüğü, sınıfa herhangi bir başka koddan erişilebildiğini belirtir. Diğer alternatifler için, bkz. Denetim ad alanı niteliklerine erişme.

3 Sınıfınızı içeren paketin adını belirtmek için bir `package` deyimi yazın. Sözdizimi, tam paket adı, `class` deyim bloğunun, ardından açılış ve kapanış küme parantezleri gelen `package` sözcüğüdür. Örneğin, aşağıdaki işlemi gerçekleştirmek için bir önceki adımdaki kodu değiştirin:

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

4 Sınıf gövdesinin içinde `var` deyimini kullanarak sınıftaki özelliklerin her birini tanımlayın. Sözdizimi, herhangi bir değişkeni bildirmek için kullandığınız sözdizimiyle aynıdır (`public` değiştiricisi eklenir). Örneğin, sınıf tanımının açma ve kapatma küme parantezlerinin arasına bu satırlar eklendiğinde, `textProperty`, `numericProperty` ve `dateProperty` adındaki özellikler oluşturulur:

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

5 İşlev tanımlamak için kullanılan aynı sözdizimini kullanarak sınıfta her bir yöntemi tanımlayın. Örneğin:

- `myMethod()` yöntemi oluşturmak için şunu girin:

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

- Yapıcı (sınıf örneği oluşturma işleminin bir parçası olarak çağrılan özel yöntem) oluşturmak için, adı sınıfın adıyla tamamen eşleşen bir yöntem oluşturun:

  ```
  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);
  }
  ```

  Sınıfınıza yapıcı yöntemi dahil etmediyseniz, derleyici sınıfınızda otomatik olarak boş bir yapıcı oluşturur. (Başka bir deyişle, parametreye ve deyime sahip olmayan bir yapıcı.)

Tanımlayabileceğiniz bir kaç adet daha sınıf öğesi vardır. Bu öğeler daha karmaşıktır.

- *Erişimciler*, bir yöntem ile özellik arasında özel bir köprüdür. Sınıfı tanımlamak için kodu yazdığınızda, erişimciyi bir yöntem şeklinde yazarsınız. Bir özelliği tanımladığınızda tek gerçekleştirebildiğiniz işlem olan yalnızca bir değer okuma veya atama yerine, bir çok işlem gerçekleştirebilirsiniz. Ancak sınıfınızın bir örneğini oluşturduğunuzda, değeri okumak veya atamak için yalnızca adı kullanarak erişimciyi bir özellik gibi değerlendirirsiniz.
- ActionScript'teki olaylar, belirli bir sözdizimi kullanılarak tanımlanmaz. Bunun yerine, EventDispatcher sınıfının işlevselliğini kullanarak sınıfınızda olayları tanımlarsınız.

Daha fazla Yardım konusu

Olayları işleme

Örnek: Temel bir uygulama oluşturma

ActionScript 3.0, Flash Professional ve Flash Builder araçları veya herhangi bir metin düzenleyiciyi içeren bir takım uygulama geliştirme ortamlarında kullanılabilir.

Bu örnekte, Flash Professional veya Flex Builder kullanarak basit bir ActionScript 3.0 uygulamasının oluşturulup geliştirilmesine yönelik adımlar açıklanmaktadır. Oluşturacağınız uygulama, Flash Professional ve Flex uygulamalarında harici ActionScript 3.0 sınıfı dosyalarının kullanılmasına yönelik basit bir desen sunar.

ActionScript uygulamanızı tasarlama

Bu örnek ActionScript uygulama örneği, standart "Hello World" uygulamasıdır, bu nedenle tasarımı basittir:

- Uygulamanın adı Hello World'tür.
- Söz konusu uygulama, “Hello World!” sözcüklerini içeren tek bir metin alanını görüntüler.
- Uygulama, Greeter adında tek bir nesne odaklı nesne kullanır. Bu tasarım, sınıfın Flash Professional veya Flex projesi içerisinde kullanılmasına izin verir.
- Bu örnekte, öncelikle uygulamanın basit bir sürümünü oluşturursunuz. Daha sonra kullanıcının kullanıcı adı girebilmesini ve uygulamanın da adı kullanıcı listesinden kontrol etmesini sağlayacak işlevselliği eklersiniz.

Bu kısa tanımın yerinde olmasıyla, uygulamayı oluşturmaya başlayabilirsiniz.

HelloWorld projesini ve Greeter sınıfını oluşturma

Hello World uygulamasının tasarım deyimi, uygulama kodunun kolayca yeniden kullanılabilmesi gerektiğini belirtir. Bu amaca ulaşmak için, uygulama Greeter adında tek bir nesne odaklı sınıf kullanır. Bu sınıfı, Flash Builder veya Flash Professional'da oluşturduğunuz bir uygulamadan kullanırsınız.

Flex'te HelloWorld projesini ve Greeter sınıfını oluşturmak için:

1. Flash Builder'da, Dosya > Yeni> Flex Projesi seçeneklerini belirleyin,
2. Proje Adı olarak Hello World yazın. Uygulama türünün "Web (Adobe Flash Player'da çalışır)"' olarak ayarlandığından emin olun ve Bitir'i tıklatın.

   Flash Builder projenizi oluşturur ve Package Explorer'da görüntüler. Proje varsayılan olarak, HelloWorld.mxml adında bir dosyayı zaten içerir ve bu dosya Düzenleyici panelinde açılır.
3. Şimdi Flash Builder aracında özel bir ActionScript sınıf dosyası oluşturmak için, Dosya > Yeni > ActionScript Sınıfı seçeneğini belirleyin.
4. Yeni ActionScript Sınıfı iletişim kutusunda, Ad alanına sınıf adı olarak **Greeter** yazın ve ardından Bitir'i tıklatın.

   Yeni bir ActionScript düzenleme penceresi görüntülenir.

   Greeter sınıfına kod ekleme bölümünden devam edin.

Flash Professional'da Greeter sınıfını oluşturmak için:

1. Flash Professional'da, Dosya >Yeni seçeneklerini belirleyin.
2. Yeni Belge iletişim kutusunda ActionScript dosyasını seçip Tamam'ı tıklatın.

   Yeni bir ActionScript düzenleme penceresi görüntülenir.
3. Dosya > Kaydet'i seçin. Uygulamanızı içerecek bir klasörü seçin, ActionScript dosyasına **Greeter.as** adını verin ve Tamam'ı tıklatın.

Greeter sınıfına kod ekleme bölümünden devam edin.

Greeter sınıfına kod ekleme

Greeter sınıfı, HelloWorld uygulamanızda kullanabileceğiniz bir `Greeter` nesnesi tanımlar.

Greeter sınıfına kod eklemek için:

1 Yeni dosyaya aşağıdaki kodu yazın (kodun bir kısmı sizin için eklenmiş olabilir)

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

Greeter sınıfı, “Hello World!” dizesini döndüren tek bir `sayHello()` yöntemini içerir.

2 Bu ActionScript dosyasını kaydetmek için Dosya > Kaydet seçeneklerini belirleyin.

Greeter sınıfı şimdi bir uygulamada kullanılmaya hazırdır.

ActionScript kodunuzu kullanan bir uygulama oluşturma

Oluşturduğunuz Greeter sınıfı, kendiliğinden bulunan bir yazılım işlevleri kümesini tanımlar ancak tam bir uygulamayı temsil etmez. Sınıfı kullanmak için, Flash Professional belgesi veya Flex projesi oluştururursunuz.

Kodun Greeter sınıfının örneğine ihtiyacı vardır. Greeter sınıfını uygulamanızda şu şekilde kullanabilirsiniz.

Flash Professional'ı kullanarak bir ActionScript uygulaması oluşturmak için:

1 Dosya > Yeni'yi seçin.

2 Yeni Belge iletişim kutusunda Flash Dosyası (ActionScript 3.0) öğesini seçip Tamam'ı tıklatın.

Yeni bir belge penceresi görüntülenir.

3 Dosya > Kaydet'i seçin. Greeter.as sınıf dosyasını içeren aynı klasörü seçin, Flash belgesine **HelloWorld.fla** adını verin ve Tamam'ı tıklatın.

4 Flash Professional araç paletinde, Metin aracını seçin. Yaklaşık 300 piksel genişliğinde ve 100 piksel yüksekliğinde yeni bir metin alanı tanımlamak için Sahne Alanı boyunca sürükleyin.

5 Özellikler panelinde, metin alanı Sahne Alanı'nda hala seçiliyken, metin türünü “Dinamik Metin” olarak ayarlayın ve metin alanının örnek adı olarak **mainText** yazın.

6 Ana zaman çizelgesinin birinci karesini tıklatın. Pencere > Eylemler'i seçerek Eylemler panelini açın.

7 Eylemler paneline şu komut dosyasını yazın:

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

8 Dosyayı kaydedin.

ActionScript uygulamanızı yayınlama ve test etme bölümünden devam edin.

Flash Builder'ı kullanarak bir ActionScript uygulaması oluşturmak için:

1 HelloWorld.mxml dosyasını açın ve aşağıdaki listelemeyle eşleşmesi için kod ekleyin:

```
<?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>
```

Bu Flex projesi dört MXML etiketi içerir:

- Uygulama kabını tanımlayan bir `<mx:Application>` etiketi
- Uygulama etiketi için mizanpaj stilini (dikey mizanpaj) tanımlayan `<s:layout>` etiketi
- ActionScript kodu içeren bir `<fx:Script>` etiketi
- Metin mesajlarını kullanıcıya görüntülemek için bir alanı tanımlayan `<s:TextArea>` etiketi

`<fx:Script>` etiketindeki kod, uygulama yüklendiğinde çağrılan bir `initApp()` yöntemini tanımlar. `initApp()` yöntemi, `mainTxt` TextArea öğesinin metin değerini, henüz yazdığınız Greeter özel sınıfının `sayHello()` yöntemi tarafından döndürülen "Hello World!" dizesine ayarlar.

2 Uygulamayı kaydetmek için Dosya > Kaydet seçeneğini belirleyin.

ActionScript uygulamanızı yayınlama ve test etme bölümünden devam edin.

ActionScript uygulamanızı yayınlama ve test etme

Yazılım geliştirme yinelemeli bir işlemdir Kod yazarsınız, yazdığınız kodları derlersiniz ve düzgün şekilde derleninceye kadar kodu düzenlersiniz. Derlenmiş uygulamayı çalıştırırsınız ve beklenen tasarımı gerçekleştirip gerçekleşmediğini test edersiniz. Gerçekleşmezse, gerçekleşene kadar kodu düzenlersiniz. Flash Professional ve Flash Builder geliştirme ortamları, uygulamalarınızın yayınlanması, test edilmesi ve hatalarının ayıklanması için birçok yol sunar.

Her ortamda HelloWorld uygulamasının test edilmesine yönelik temel adımlar şunlardır.

Flash Builder'ı kullanarak bir ActionScript uygulamasını yayınlamak ve test etmek için:

1. Uygulamanızı yayınlayın ve derleme hatası olup olmadığına bakın. Flash Professional'da, ActionScript kodunuzu derlemek ve HelloWorld uygulamasını çalıştırmak için, Kontrol Et > Filmi Test Et seçeneklerini belirleyin.
2. Uygulamayı test ettiğinizde Çıktı penceresinde hatalar veya uyarılar görüntülenirse, hataları HelloWorld.fla veya HelloWorld.as dosyalarında düzeltin. Bundan sonra uygulamayı test etmeyi tekrar deneyin.
3. Herhangi bir derleme hatası yoksa, Hello World uygulamasını gösteren bir Flash Player penceresi görüntülenir.

ActionScript 3.0'ı kullanan basit ancak tam bir nesne tabanlı uygulama oluşturdunuz. HelloWorld uygulamasını geliştirme bölümünden devam edin.

Flash Builder'ı kullanarak bir ActionScript uygulamasını yayınlamak ve test etmek için:

1. Çalıştır > HelloWorld'ü Çalıştır öğesini seçin.
2. HelloWorld uygulaması başlatılır.
   - Uygulamayı test ettiğinizde Çıktı penceresinde hatalar veya uyarılar görüntülenirse, HelloWorld.mxml veya Greeter.as dosyalarındaki hataları düzeltin. Bundan sonra uygulamayı test etmeyi tekrar deneyin.
   - Herhangi bir derleme hatası yoksa, Hello World uygulamasını gösteren bir tarayıcı penceresi açılır. “Hello World!” metninin olduğu bir ekran görüntülenir.

   ActionScript 3.0'ı kullanan basit ancak tam bir nesne tabanlı uygulama oluşturdunuz. HelloWorld uygulamasını geliştirme bölümünden devam edin.

HelloWorld uygulamasını geliştirme

Uygulamayı daha ilgi çekici hale getirmek için şimdi uygulamanın bir kullanıcı adı istemesini ve bu kullanıcı adını önceden tanımlı bir adlar listesine göre doğrulamasını sağlayacaksınız.

İlk olarak, yeni işlevsellik eklemek için Greeter sınıfını güncellersiniz. Ardından yeni işlevselliği kullanmak için uygulamayı güncellersiniz.

Greeter.as dosyasını güncellemek için:

1. Greeter.as dosyasını açın.
2. Dosyanın içeriklerini şunlarla değiştirin (yeni ve değiştirilen satırlar kalın yazı tipiyle gösterilmektedir):

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

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                            + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

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

Greeter sınıfı şimdi birkaç yeni özelliğe sahiptir:

- `validNames` dizisi geçerli kullanıcı adlarını listeler. Greeter sınıfı yüklendiğinde bu dizi üç addan oluşan bir liste şeklinde başlatılır.
- `sayHello()` yöntemi şimdi bir kullanıcı adını kabul eder ve bazı koşulları esas alarak selamlamayı değiştirir. `userName` boş bir dize (`""`) olursa, `greeting` özelliği, kullanıcıdan bir ad isteyecek şekilde ayarlanır. Kullanıcı adı geçerliyse, selamlama şöyle olur: `"Hello, `*userName*`."` Son olarak, bu iki koşuldan herhangi biri karşılanmazsa, `greeting` değişkeni `"Sorry `*userName*`, you are not on the list."` olarak ayarlanır.

- `inputName` öğesi `validNames` dizisinde bulunuyorsa, `validName()` yöntemi `true` değerini döndürür, bulunmuyorsa, `false` değerini döndürür. `validNames.indexOf(inputName)` deyimi, `validNames` dizisindeki dizelerin her birini `inputName` dizesinde kontrol eder. `Array.indexOf()` yöntemi bir dizideki nesnenin ilk örneğinin dizin konumunu döndürür. Nesne dizede bulunamazsa -1 değerini döndürür.

Daha sonra bu ActionScript sınıfına başvuran uygulama dosyasını düzenlersiniz.

Uygulamayı Flash Professional kullanarak değiştirmek için:

1. HelloWorld.fla dosyasını açın.
2. Greeter sınıfının `sayHello()` yöntemine boş bir dize (`""`) iletilecek şekilde Kare 1'de komut dosyasını değiştirin:

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

3. Araçlar paletinden Metin aracını seçin. Sahne Alanı'nda iki yeni metin alanı oluşturun. Bunları doğrudan varolan `mainText` metin alanının altında yan yana yerleştirin.
4. Etiket olan ilk yeni metin alanında **User Name:** metnini yazın.
5. Diğer yeni metin alanını seçin ve Özellik denetçisinde metin alanı türü olarak Input Text seçeneğini belirleyin. Satır türü olarak Tek satır seçin. Örnek adı olarak **textIn** yazın.
6. Ana zaman çizelgesinin birinci karesini tıklatın.
7. Eylemler panelinde, varolan komut dosyasının sonuna şu satırları ekleyin:

```
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);
    }
}
```

Yeni kod şu işlevselliği ekler:

- İlk iki satır, iki metin alanının kenarlıklarını tanımlar.
- `textIn` alanı gibi bir girdi metni alanı, gönderebileceği olayların bir kümesini içerir. `addEventListener()` yöntemi, bir olay türü gerçekleştiğinde çalıştırılan bir işlevi tanımlamanıza olanak sağlar. Bu durumda, söz konusu olay klavyedeki bir tuşa basılmasıdır.
- `keyPressed()` özel işlevi, basılan tuşun Enter tuşu olup olmadığını kontrol eder. Basılan tuş Enter ise, `myGreeter` nesnesinin `sayHello()` yöntemini çağırarak `textIn` metin alanındaki metni parametre olarak iletir. Bu yöntem, iletilen değeri esas alarak bir selamlama dizesi döndürür. Döndürülen dize daha sonra `mainText` metin alanının `text` özelliğine atanır.

Kare 1'in tam komut dosyası şudur:

```
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 Dosyayı kaydedin.

9 Uygulamayı çalıştırmak için Kontrol Et > Filmi Test Et'i seçin.

Uygulamayı çalıştırdığınızda, bir kullanıcı adı girmenizi ister. Kullanıcı adı geçerliyse (Sami, Ferdi veya Deniz), uygulama "hello" onaylama mesajını görüntüler.

Uygulamayı Flash Builder kullanarak değiştirmek için:

1 HelloWorld.mxml dosyasını açın.

2 Daha sonra, kullanıcıya yalnızca görüntülenmeye ilişkin olduğunu bildirmek için `:<mx:TextArea>` etiketini değiştirin. Arka plan rengini açık griye dönüştürün ve `editable` niteliğini `false` olarak ayarlayın:

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

3 Şimdi, `<s:TextArea>` kapatma etiketinden hemen sonra şu satırları ekleyin. Bu satırlar, kullanıcının bir kullanıcı adı değeri girmesine olanak tanıyan yeni bir TextInput bileşeni oluşturur:

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

`enter` niteliği, kullanıcı `userNameTxt` alanında Enter tuşuna bastığında gerçekleşen eylemleri tanımlar. Bu örnekte, kod alandaki metni `Greeter.sayHello()` yöntemine iletir. `mainTxt` alanındaki karşılama buna göre değişir.

HelloWorld.mxml dosyası aşağıdaki gibi görünür:

```
<?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 Düzenlenen HelloWorld.mxml dosyasını kaydedin. Uygulamayı çalıştırmak için Çalıştır > HelloWorld'ü Çalıştır öğesini seçin.

Uygulamayı çalıştırdığınızda, uygulama bir kullanıcı adı girmenizi ister. Kullanıcı adı geçerliyse (Sami, Ferdi veya Deniz), uygulama "`Hello`,*userName*" onaylama mesajını görüntüler.

Bölüm 3: ActionScript dili ve sözdizimi

ActionScript 3.0, hem ActionScript dilini hem de Adobe Flash Player Uygulaması Programlama Arabirimi'ni (API) içerir. Çekirdek dil, ActionScript'in dil sözdizimini ve üst düzey veri türlerini tanımlayan bölümüdür. ActionScript 3.0 Adobe Flash Platform çalışma zamanlarına programlı erişim sağlar: Adobe Flash Player ve Adobe AIR.

Dile genel bakış

ActionScript 3.0 dilinin temelini nesneler oluşturur, bunlar temel bina bloklarıdır. Bildirdiğiniz her değişken, yazdığınız her işlev ve oluşturduğunuz her sınıf örneği bir nesnedir. ActionScript 3.0 programını, görevleri gerçekleştiren, olayları yanıtlayan ve birbiriyle iletişim kuran bir nesne grubu olarak düşünebilirsiniz.

Java veya C++ uygulamasında nesne tabanlı programlamayı (OOP) bilen programcılar, nesneleri, iki tür üye içeren modüller olarak düşünebilir: üye değişkenlerde veya özelliklerde saklanan veriler ve yöntemler üzerinden erişilebilir davranış. ActionScript 3.0, nesneleri benzer ancak biraz daha farklı bir şekilde tanımlar. ActionScript 3.0'da nesneler yalnızca özellikler koleksiyonudur. Bu özellikler, yalnızca verileri değil aynı zamanda işlevleri veya diğer nesneleri de içeren konteynerlerdir. Bir işlev bir nesneye bu şekilde eklenirse, buna yöntem denir.

ActionScript 3.0 tanımı, Java veya C++ arka planına sahip programcılar için biraz garip görünse de, uygulamada, ActionScript 3.0 sınıflarıyla nesne türlerinin tanımlanması, Java ya da C++ uygulamalarında sınıfların tanımlanmasına çok benzer. ActionScript nesne modeli ile diğer gelişmiş konular ele alınırken, iki nesne tanımı arasındaki fark önemlidir ancak diğer durumların çoğunda, *özellikler* terimi, yöntemlerin tersine, sınıf üyesi değişkenleri anlamına gelir. Flash Platform için Adobe Action Script 3.0 Referansı, örneğin, *properties* terimini değişken veya alıcı-ayarlayıcı özelliğini belirtmek için kullanır. Bir sınıfın parçası olan işlevleri ifade etmek için *yöntemler* terimini kullanır.

ActionScript'teki sınıflar ile Java veya C++ uygulamalarındaki sınıflar arasındaki bir fark, ActionScript'te sınıfların yalnızca soyut varlıklar olmamasıdır. ActionScript sınıfları, sınıfın özelliklerini ve yöntemlerini saklayan *sınıf nesneleri* ile temsil edilir. Bu da, bir sınıfın veya paketin üst düzeyine deyimler ya da çalıştırılabilir kod dahil etmek gibi Java ve C++ programcılarına yabancı görünebilecek tekniklere olanak sağlar.

ActionScript sınıfları ile Java veya C++ sınıfları arasındaki diğer bir fark, her ActionScript sınıfında *prototipnesnesi* adında bir öğenin olmasıdır. Önceki ActionScript sürümlerinde, *prototip zincirleri* içinde bağlanan prototip nesneleri, topluca tüm sınıf mirası hiyerarşisinin temeli görevini görürdü. ActionScript 3.0'da ise prototip nesneleri miras sisteminde yalnızca küçük bir rol oynar. Ancak bir özelliği ve o özelliğin değerini bir sınıfın tüm örnekleri arasında paylaşmak istiyorsanız, prototip nesnesi, statik özellik ve yöntemler için alternatif olarak yine kullanışlı olabilir.

Geçmişte, ileri düzey ActionScript programcıları, özel yerleşik dil öğeleriyle prototip zincirini doğrudan işleyebilirdi. Şimdi dil, sınıf tabanlı programlama arabiriminin daha olgun bir uygulamasını sağladığından, `__proto__` ve `__resolve` gibi bu özel dil öğelerinin çoğu, artık dilin bir parçası değildir. Üstelik, performansta önemli ölçüde artış sağlayan dahili miras mekanizmalarının en iyileştirilmesi, miras mekanizmasına doğrudan erişimi önler.

Nesneler ve sınıflar

ActionScript 3.0'da, her nesne bir sınıf tarafından tanımlanır. Sınıf, bir nesne türünün şablonu veya şeması olarak düşünülebilir. Sınıf tanımları, sınıfa bağlı davranışı kapsayan işlevler niteliğindeki veri değerlerini ve yöntemlerini barındıran değişkenleri ve sabitleri içerebilir. Özelliklerde saklanan değerler, *ilkel değerler* veya başka nesneler olabilir. İlkel değerler, sayılar, dizeler veya Boolean değerleridir.

ActionScript, çekirdek dilin parçası olan birçok yerleşik sınıfı içerir. Number, Boolean ve String gibi bu yerleşik sınıflardan bazıları, ActionScript'te kullanılabilir olan ilkel değerleri temsil eder. Array, Math ve XML sınıfları gibi diğer sınıflar daha karmaşık nesneleri tanımlar.

Yerleşik veya kullanıcı tanımlı tüm sınıflar, Object sınıfından türetilir. Önceki ActionScript sürümlerinde deneyimli programcılar için, diğer tüm sınıflar Object veri türünden türetilse de, Object veri türünün artık varsayılan veri türü olmadığının unutulmaması önemlidir. ActionScript 2.0'da, tür ek açıklamasının olmaması değişkenin Object türünde olduğu anlamına geldiğinden, şu iki kod satırı eşdeğerdir:

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

Ancak ActionScript 3.0, şu iki yöntemle belirlenebilen türlenmemiş değişken kavramını getirmiştir:

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

Türlenmemiş bir değişken, Object türünde bir değişkenle aynı değildir. Önemli olan fark, Object türündeki bir değişken `undefined` özel değerini barındıramazken, türlenmemiş değişkenlerin bu değeri barındırabilmesidir.

`class` anahtar sözcüğünü kullanarak kendi sınıflarınızı tanımlayabilirsiniz. Sınıf özelliklerini üç şekilde bildirebilirsiniz: bir yöntem bildiriminde `const` anahtar sözcüğüyle sabitler tanımlanabilir, `var` anahtar sözcüğüyle değişkenler tanımlanır ve `get` ve `set` nitelikleriyle alıcı ve ayarlayıcı özellikleri tanımlanır. `function` anahtar sözcüğüyle yöntemler bildirebilirsiniz.

`new` operatörünü kullanarak bir sınıfın örneğini oluşturursunuz. Aşağıdaki örnek, `myBirthday` adında bir Date sınıfı örneğini oluşturur.

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

Paketler ve ad alanları

Paketler ve ad alanları ilişkili kavramlardır. Paketler, kod paylaşımını kolaylaştıracak ve adlandırma çakışmalarını en düşük düzeye indirecek şekilde sınıf tanımlarını bir arada paketlemenize olanak sağlar. Ad alanları, özellik ve yöntem adları gibi tanımlayıcıların görünürlüğünü denetlemenize olanak sağlar ve bir paketin içinde veya dışında da kalsa, koda uygulanabilir. Paketler, sınıf dosyalarınızı organize etmenize olanak sağlarken, ad alanları, tek tek özelliklerin ve yöntemlerin görünürlüğünü yönetmenizi sağlar.

Paketler

ActionScript 3.0'daki paketler, ad alanları ile uygulanır ancak bunlarla eşanlamlı değildir. Bir paket bildirdiğinizde, derleme zamanında bilinmesi garantilenen özel bir ad alanı türünü açıkça oluşturursunuz. Ad alanları açıkça oluşturulduğunda derleme zamanında mutlaka bilinmek zorunda değildir.

Aşağıdaki örnek, tek bir sınıf içeren basit bir paket oluşturmak için `package` direktifini kullanır:

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

Bu örnekteki sınıfın adı SampleCode'dur. Sınıf, samples paketinin içinde olduğundan, derleyici otomatik olarak derleme zamanında sınıf adını samples.SampleCode tam nitelendirilmiş adıyla niteler. Derleyici de, `sampleGreeting` ve `sampleFunction()` öğeleri sırayla `samples.SampleCode.sampleGreeting` ve `samples.SampleCode.sampleFunction()` olacak şekilde özelliklerin veya yöntemlerin adlarını niteler.

Geliştiricilerin çoğu, özellikle de Java programlama arka planına sahip olanlar, bir paketin üst düzeyine yalnızca sınıfları yerleştirmeyi seçebilir. Ancak ActionScript 3.0, bir paketin üst düzeyinde yalnızca sınıfları değil, değişkenleri, işlevleri ve deyimleri de destekler. Bu özelliğin gelişmiş bir kullanımı, bir paketin üst düzeyinde bir ad alanını, paketteki tüm sınıflar için kullanılabilir olacak şekilde tanımlamaktır. Ancak bir paketin üst düzeyinde yalnızca `public` ve `internal` olmak üzere iki erişim belirticisine izin verildiğini unutmayın. Yuvalanmış sınıfları özel olarak bildirmenize olanak sağlayan Java'dan farklı olarak ActionScript 3.0, yuvalanmış veya özel sınıfları desteklemez.

Ancak ActionScript 3.0 paketleri, diğer birçok yönden Java programlama dilindeki paketlere benzer. Önceki örnekte görebildiğiniz gibi, tam nitelendirilmiş paket başvuruları, tıpkı Java'daki gibi nokta operatörü (.) kullanılarak ifade edilir. Kodunuzu diğer programcıların da kullanması için sezgisel bir hiyerarşik yapıda organize etmek üzere paketleri kullanabilirsiniz. Bu, kod paylaşımını kolaylaştırır ve böylece kendi paketinizi oluşturup başkalarıyla paylaşmanıza ve başkaları tarafından oluşturulan paketleri kodunuzda kullanmanıza olanak sağlar.

Ayrıca, paketlerin kullanılması, kullandığınız tanımlayıcı adlarının benzersiz olmasının ve diğer tanımlayıcı adlarıyla çakışmamasının sağlanmasına yardımcı olur. Aslında bazılarına göre, paketlerin birincil avantajı budur. Örneğin, kodlarını birbiriyle paylaşmak isteyen iki programcı, SampleCode adında bir sınıf oluşturabilir. Paketler olmadan bu bir ad çakışması oluşturur ve bunun tek çözümü sınıflardan birinin yeniden adlandırılmasıdır. Ancak paketler sayesinde, paketlerdeki sınıflardan biri veya tercihen ikisi benzersiz adlarla yerleştirilerek ad çakışması kolayca önlenir.

Yuvalanmış paketler oluşturmak için paket adınıza gömülü noktalar da dahil edebilirsiniz. Bu, paketlerin hiyerarşik organizasyonunu oluşturmanıza olanak sağlar. Buna iyi bir örnek, ActionScript 3.0 tarafından sağlanan flash.display paketidir. flash.display paketi, flash paketinin içinde yuvalanmıştır.

ActionScript 3.0'ın çoğu, flash paketi altında organize edilir. Örneğin, flash.display paketi, görüntüleme listesi API'sini içerirken, flash.events paketi de yeni olay modelini içerir.

Paketler oluşturma

ActionScript 3.0, paketlerinizi, sınıflarınızı ve kaynak dosyalarınızı organize etme şeklinizde önemli ölçüde esneklik sağlar. Önceki ActionScript sürümleri, her kaynak dosyası için yalnızca bir sınıfa izin verirdi ve kaynak dosyasının adının, sınıfın adıyla eşleşmesini gerektirirdi. ActionScript 3.0, tek bir kaynak dosyasına birden çok sınıf dahil etmenize olanak sağlar ancak her dosyadaki yalnızca bir sınıf o dosyaya harici olan kod için kullanılabilir duruma getirilebilir. Başka bir deyişle, her dosyadaki yalnızca bir sınıf, paket bildiriminde bildirilebilir. Ek sınıfları paket tanımınızın dışında bildirmeniz gerekir ve böylece bu sınıflar, o kaynak dosyasının dışındaki kod için görünmez olur. Paket tanımının içinde bildirilen sınıfın adı, kaynak dosyasının adıyla eşleşmelidir.

ActionScript 3.0, ayrıca paketleri bildirme şeklinizde de daha fazla esneklik sağlar. Önceki ActionScript sürümlerinde, paketler yalnızca kaynak dosyalarını yerleştirdiğiniz dizinleri temsil ederdi ve siz `package` deyimiyle paketleri bildirmez, bunun yerine sınıf bildiriminize tam nitelendirilmiş sınıf adının parçası olarak paket adını dahil ederdiniz. Paketler halen ActionScript 3.0'da dizinleri temsil etmeye devam etse de, yalnızca sınıfları değil, daha fazlasını da içerebilir. ActionScript 3.0'da, bir paketi bildirmek için `package` deyimini kullanırsınız, başka bir deyişle, bir paketin üst düzeyinde değişkenleri, işlevleri ve ad alanlarını da bildirebilirsiniz. Paketin üst düzeyine çalıştırılabilir deyimler de dahil edebilirsiniz. Bir paketin üst düzeyinde değişkenler, işlevler veya ad alanları bildirirseniz, o düzeyde kullanılabilir olan nitelikler yalnızca `public` ve `internal` olur ve bildirim bir sınıf, değişken, işlev ya da ad alanı olsa da, her dosya için yalnızca bir paket düzeyi bildirimi, `public` niteliğini kullanabilir.

Paketler, kodunuzun organize edilmesinde ve ad çakışmalarının önlenmesinde kullanışlıdır. Paket kavramlarını, ilgili olmayan sınıf mirası kavramıyla karıştırmamalısınız. Aynı pakette kalan iki sınıf ortak bir ad alanına sahip olur ancak bu iki sınıfın mutlaka herhangi bir şekilde birbiriyle ilgili olması gerekmez. Aynı şekilde, yuvalanmış bir paketin, üst paketiyle herhangi bir anlamsal ilişkisi olmayabilir.

Paketleri içe aktarma

Bir paketin içindeki sınıfı kullanmak istiyorsanız, paketi veya belirli sınıfı içe aktarmanız gerekir. Bu, sınıfların içe aktarılmasının isteğe bağlı olduğu ActionScript 2.0'dan farklılık gösterir.

Örneğin, daha önce sunulan SampleCode sınıf örneğini düşünün. Sınıf, samples adında bir pakette kalıyorsa, SampleCode sınıfını kullanmadan önce aşağıdaki içe aktarma deyimlerinden birini kullanmanız gerekir:

```
import samples.*;
```

veya

```
import samples.SampleCode;
```

Genelde, `import` deyimleri olabildiğince belirli olmalıdır. samplespaketinden yalnızca SampleCode sınıfını kullanmayı planlıyorsanız, sınıfın ait olduğu paketin tamamını değil, yalnızca SampleCode sınıfını içe aktarmanız gerekir. Paketin tamamının içe aktarılması, beklenmeyen ad çakışmalarına yol açabilir.

Sınıf yolunuzun içine, paketi veya sınıfı tanımlayan kaynak kodunu da yerleştirmeniz gerekir. Sınıf yolu, derleyicinin içe aktarılan paketleri ve sınıfları nerede arayacağını belirleyen yerel dizin yollarının kullanıcı tanımlı bir listesidir. Sınıf yolu bazen *oluşturma yolu* veya *kaynak yolu* olarak da adlandırılır.

Sınıfı veya paketi düzgün şekilde içe aktardıktan sonra, sınıfın tam nitelendirilmiş adını (samples.SampleCode) veya yalnızca sınıf adını (SampleCode) kullanabilirsiniz.

Tam nitelendirilmiş adlar, aynı ada sahip sınıflar, yöntemler veya özellikler belirsiz koda neden olduğunda kullanışlıdır ancak tüm tanımlayıcılar için kullanıldığında yönetimleri zor olabilir. Örneğin, bir SampleCode sınıf örneğini başlattığınızda, tam nitelendirilmiş ad kullanılması, ayrıntılı bir kod verir:

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

Yuvalanmış paketlerin düzeyleri arttıkça, kodunuzun okunabilirliği de azalır. Belirsiz tanımlayıcıların sorun yaratmayacağından emin olduğunuz durumlarda basit tanımlayıcıları kullanarak kodunuzun okunmasını kolaylaştırabilirsiniz. Örneğin, yeni bir SampleCode sınıf örneğinin başlatılması, yalnızca sınıf tanımlayıcısını kullanırsanız daha az ayrıntılıdır.

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

Önce uygun paketi veya sınıfı içe aktarmadan tanımlayıcı adlarını kullanmayı denerseniz, derleyici sınıf tanımlarını bulamaz. Diğer yandan, paketi veya sınıfı içe aktarırsanız, içe aktarılmış bir adla çakışan bir adı tanımlama girişiminde bulunulduğunda bir hata oluşur.

Bir paket oluşturulduğunda, o paketin tüm üyeleri için varsayılan erişim belirticisi `internal` olur, başka bir deyişle, varsayılan olarak paket üyeleri yalnızca o paketin diğer üyelerine görünür. Bir sınıfın, paket dışındaki kod için kullanılabilir olmasını istiyorsanız, o sınıfın `public` olduğunu bildirmeniz gerekir. Örneğin, aşağıdaki paket iki sınıf içerir: SampleCode ve CodeFormatter:

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

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

SampleCode sınıfı, `public` sınıfı olarak bildirildiğinden, paket dışında da görünebilir. CodeFormatter sınıfı ise yalnızca samples paketinin içinde görünebilir. CodeFormatter sınıfına samples paketinin dışında erişmeye çalışırsanız, aşağıdaki örnekte gösterildiği gibi bir hata oluşturur:

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

Her iki sınıfın da paket dışında kullanılabilir olmasını istiyorsanız, her iki sınıfın `public` olduğunu bildirmeniz gerekir. Paket bildirimine `public` niteliğini uygulayamazsınız.

Tam nitelendirilmiş adlar, paketler kullanılırken oluşabilecek ad çakışmalarının çözümlenmesinde kullanışlıdır. Sınıfları aynı tanımlayıcı ile tanımlayan iki paketi içe aktarırsanız böyle bir senaryo gerçekleşebilir. Örneğin, ayrıca SampleCode adında bir sınıfa sahip olan şu paketi göz önünde bulundurun:

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

Her iki sınıfı da aşağıdaki gibi içe aktarırsanız, SampleCode sınıfını ifade ederken bir ad çakışması olur:

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

Derleyici, hangi SampleCode sınıfının kullanılacağını bilemez. Bu çakışmayı çözümlemek için, aşağıdaki gibi her sınıfın tam nitelendirilmiş adını kullanmanız gerekir:

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

Not: *C++ arka planına sahip programcılar genellikle `import` deyimini `#include` deyimiyle karıştırır. C++ derleyicileri aynı anda tek bir dosya işlediğinden ve açıkça bir başlık dosyası dahil edilmediyse sınıf tanımları için diğer dosyalara bakmadığından, `#include` direktifi C++ uygulamasında gereklidir. ActionScript 3.0, bir `include` direktifine sahiptir ancak bu direktif, sınıfları ve paketleri içe aktarmak için tasarlanmamıştır. ActionScript 3.0'da sınıfları veya paketleri içe aktarmak için, `import` deyimini kullanmanız ve paketi içeren kaynak dosyasını sınıf yoluna yerleştirmeniz gerekir.*

Ad alanları

Ad alanları, oluşturduğunuz özellik ve yöntemlerin görünürlüğü üzerinde denetim elde etmenizi sağlar. `public`, `private`, `protected` ve `internal` erişim denetimi belirticilerini, yerleşik ad alanları olarak düşünün. Bu önceden tanımlı erişim denetimi belirticileri ihtiyaçlarınıza uymuyorsa, kendi ad alanlarınızı oluşturabilirsiniz.

XML ad alanları hakkında bilginiz varsa, ActionScript uygulamasının sözdizimi ve ayrıntıları XML'dekinden biraz daha farklı olsa da, burada ele alınan konuların çoğu size yabancı gelmeyecektir. Daha önce hiç ad alanlarıyla çalışmadıysanız, kavramın kendisi oldukça anlaşılırdır ancak uygulama için öğrenmeniz gereken belirli terminolojiler vardır.

Ad alanlarının nasıl çalıştığını anlamak için, özellik veya yöntem adının her zaman tanımlayıcı ve ad alanı olmak üzere iki bölüm içerdiğinin bilinmesi yardımcı olacaktır. Tanımlayıcı, genellikle ad olarak düşündüğünüz şeydir. Örneğin, aşağıdaki sınıf tanımında bulunan tanımlayıcılar `sampleGreeting` ve `sampleFunction()` şeklindedir:

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

Tanımların önünde bir ad alanı niteliği olmadığında, tanımların adları varsayılan olarak `internal` ad alanı şeklinde nitelendirilir, başka bir deyişle bunlar yalnızca aynı paketteki çağıranlara görünür. Derleyici katı moda ayarlanırsa, derleyici, ad alanı niteliği olmayan tüm tanımlayıcılara `internal` ad alanının uygulandığına dair bir uyarı yayınlar. Bir tanımlayıcının her yerde kullanılabilir olmasını sağlamak için, tanımlayıcı adının başına özel olarak `public` niteliğini getirmeniz gerekir. Önceki örnek kodda, hem `sampleGreeting` hem de `sampleFunction()`, `internal` ad alanı değerine sahiptir.

Ad alanları kullanılırken izlenmesi gereken üç temel adım vardır. İlk olarak, `namespace` anahtar sözcüğünü kullanarak ad alanını tanımlamanız gerekir. Örneğin, aşağıdaki kod `version1` ad alanını tanımlar:

```
namespace version1;
```

İkinci olarak, ad alanınızı bir özellik veya yöntem bildiriminde erişim denetimi belirticisinin yerine kullanarak uygularsınız. Aşağıdaki örnek, `version1` ad alanına `myFunction()` adında bir işlev yerleştirir:

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

Üçüncü olarak, ad alanını uyguladıktan sonra, `use` direktifiyle veya bir tanımlayıcının adını bir ad alanıyla niteleyerek bu ad alanına başvurabilirsiniz. Aşağıdaki örnek, `use` direktifi yoluyla `myFunction()` işlevine başvurur:

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

Aşağıdaki örnekte gösterildiği gibi, `myFunction()` işlevine başvurmak için nitelendirilmiş bir ad da kullanabilirsiniz:

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

Ad alanlarını tanımlama

Ad alanları, bazen *ad alanı adı* olarak da adlandırılan tek bir değer (Uniform Resource Identifier (URI)) içerir. URI, ad alanı tanımınızın benzersiz olmasını sağlamanıza yardımcı olur.

İki yöntemden birini kullanarak bir ad alanı tanımı bildirip bir ad alanı oluşturursunuz. Bir XML ad alanı tanımlayacağınızdan, açık bir URI ile bir ad alanını tanımlayabilir veya URI'yi atabilirsiniz. Aşağıdaki örnek, URI kullanılarak bir ad alanının nasıl tanımlanabildiğini gösterir:

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

URI, o ad alanı için benzersiz bir kimlik dizesi görevi görür. URI'yi aşağıdaki örnekteki gibi atarsanız, derleyici URI yerine benzersiz bir dahili kimlik dizesi oluşturur. Bu dahili kimlik dizesine erişiminiz olmaz.

```
namespace flash_proxy;
```

Siz URI ile veya URI olmadan bir ad alanını tanımladıktan sonra, o ad alanı aynı kapsamda yeniden tanımlanamaz. Aynı kapsamda önceden tanımlanmış bir ad alanını tanımlama girişimi, bir derleyici hatasına yol açar.

Bir paket veya sınıf içinde bir ad alanı tanımlanırsa, uygun erişim denetimi belirticileri kullanılmadığı sürece, söz konusu ad alanı, o paket veya sınıf dışındaki koda görünmeyebilir. Örneğin, aşağıdaki kod, flash.utils paketinde tanımlanan `flash_proxy` ad alanını gösterir. Aşağıdaki örnekte, erişim denetimi belirticisinin olmaması, `flash_proxy` ad alanının yalnızca flash.utils paketindeki koda görüneceği ve bu paket dışındaki kodlara görünmeyeceği anlamına gelir:

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

Aşağıdaki kod, `flash_proxy` ad alanını paket dışındaki kodlara görünür duruma getirmek için `public` niteliğini kullanır:

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

Ad alanlarını uygulama

Ad alanı uygulanması, bir ad alanına bir tanımın yerleştirilmesi anlamına gelir. Ad alanlarına yerleştirilebilen tanımlar arasında, işlevler, değişkenler ve sabitler yer alır. (Özel bir ad alanına sınıf yerleştiremezsiniz.)

Örneğin, `public` erişim denetimi ad alanı kullanılarak bildirilen bir işlevi göz önünde bulundurun. Bir işlev tanımında `public` niteliği kullanıldığında, o işlev genel ad alanına yerleştirilir ve böylece işlev, tüm kod için kullanılabilir olur. Bir ad alanı tanımladıktan sonra, tanımladığınız ad alanını `public` niteliğiyle aynı şekilde kullanabilirsiniz ve tanım da özel ad alanınıza başvurabilen kod için kullanılabilir hale gelir. Örneğin, bir ad alanını `example1` olarak tanımlarsanız, aşağıdaki örnekte gösterildiği gibi, nitelik olarak `example1` öğesini kullanıp `myFunction()` adında bir yöntemi ekleyebilirsiniz:

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

Nitelik olarak `example1` ad alanı kullanılarak `myFunction()` yönteminin bildirilmesi, yöntemin `example1` ad alanına ait olduğu anlamına gelir.

Ad alanlarını uygularken aşağıdakileri göz önünde bulundurmanız gerekir:

- Her bildirime yalnızca bir ad alanı uygulayabilirsiniz.
- Aynı anda birden çok tanıma bir ad alanı niteliği uygulanamaz. Başka bir deyişle, ad alanınızı on farklı işleve uygulamak istiyorsanız, on işlev tanımının her birine nitelik olarak ad alanınızı eklemeniz gerekir.
- Ad alanları ve erişim denetimi belirticileri birbirini dışladığından, bir ad alanı uygularsanız, bir erişim denetimi belirticisi de belirtemezsiniz. Başka bir deyişle, bir işlevi veya özelliği ad alanınıza uygulamanın yanı sıra, `public`, `private`, `protected` ya da `internal` olarak bildiremezsiniz.

Ad alanlarına başvurma

`public`, `private`, `protected` ve `internal` gibi herhangi bir erişim denetimi ad alanıyla bildirilmiş bir yöntem veya özellik kullandığınızda, ad alanına açıkça başvurulmasına gerek yoktur. Bunun nedeni, bu özel ad alanlarına erişimin bağlama göre denetlenmesidir. Örneğin, `private` ad alanına yerleştirilen tanımlar, aynı sınıf içindeki kod için otomatik olarak kullanılabilir olur. Ancak tanımladığınız ad alanları için böyle bir bağlam duyarlılığı yoktur. Özel bir ad alanına yerleştirdiğiniz bir yöntem veya özelliği kullanmak için, ad alanına başvurmanız gerekir.

`use namespace` direktifiyle ad alanlarına başvurabilir veya ad niteleyicisi (`::`) işaretini kullanarak ad alanıyla adı niteleyebilirsiniz. `use namespace` direktifiyle bir ad alanına başvurulması, ad alanını "açar", böylece ad alanı nitelendirilmiş olmayan herhangi bir tanımlayıcıya uygulanabilir. Örneğin, `example1` ad alanını tanımladıysanız, `use namespace example1` direktifini kullanarak o ad alanındaki adlara erişebilirsiniz:

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

Aynı anda birden çok ad alanı açabilirsiniz. Siz `use namespace` direktifiyle bir ad alanını açtıktan sonra, bu ad alanı, açıldığı kod bloğu boyunca açık kalır. Ad alanını açıkça kapatmanın bir yolu yoktur.

Ancak birden çok ad alanının açılması, ad çakışması olma olasılığını artırır. Bir ad alanını açmamayı tercih derseniz, yöntem veya özellik adını ad alanı ve ad niteleyicisi işaretiyle niteleyerek `use namespace` direktifini önleyebilirsiniz. Örneğin, aşağıdaki kod, `myFunction()` adını `example1` ad alanıyla nasıl niteleyebileceğinizi gösterir:

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

Ad alanlarını kullanma

ActionScript 3.0'ın parçası olan flash.utils.Proxy sınıfında ad çakışmalarını önlemek için kullanılan bir gerçek ad alanı örneğini bulabilirsiniz. ActionScript 2.0'daki `Object.__resolve` özelliğinin yerini alan Proxy sınıfı, bir hata oluşmadan önce tanımsız özellik veya yöntemlere yapılan başvuruları önlemenize yardımcı olur. Ad çakışmalarını önlemek için, Proxy sınıfının tüm yöntemleri, `flash_proxy` ad alanında bulunur.

`flash_proxy` ad alanının nasıl kullanıldığını daha iyi anlamak için, Proxy sınıfının nasıl kullanıldığını anlamanız gerekir. Proxy sınıfının işlevselliği, yalnızca Proxy sınıfından miras alan sınıflar için kullanılabilir durumdadır. Başka bir deyişle, bir nesnede Proxy sınıfının yöntemlerini kullanmak isterseniz, nesnenin sınıf tanımının, Proxy sınıfını genişletmesi gerekir. Örneğin, tanımsız bir yöntemi çağırma girişimlerinin tümünü önlemek istiyorsanız, Proxy sınıfını genişletir ve daha sonra Proxy sınıfının `callProperty()` yöntemini geçersiz kılarsınız.

Ad alanlarının uygulanmasının genellikle ad alanını tanımlama, uygulama ve ad alanına başvurma olmak üzere üç adımlık bir işlem olduğunu hatırlayabilirsiniz. Ancak, Proxy sınıfı yöntemlerinden herhangi birini asla açıkça çağırmadığınızdan, `flash_proxy` ad alanı yalnızca tanımlanır ve uygulanır fakat bu ad alanına asla başvurulmaz. ActionScript 3.0, `flash_proxy` ad alanını tanımlar ve bunu Proxy sınıfında uygular. Kodunuzun yalnızca Proxy sınıfını genişleten sınıflara `flash_proxy` ad alanını uygulaması gerekir.

`flash_proxy` ad alanı, flash.utils paketinde aşağıdakine benzer şekilde tanımlanır:

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

Aşağıdaki Proxy sınıfı alıntısında gösterildiği gibi, ad alanı, Proxy sınıfının yöntemlerine uygulanır:

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

Aşağıdaki kodda gösterildiği gibi, ilk olarak hem Proxy sınıfını hem de `flash_proxy` ad alanını içe aktarmanız gerekir. Daha sonra, Proxy sınıfını genişletecek şekilde sınıfınızı bildirmeniz gerekir. (Katı modda derleme yapıyorsanız, ayrıca `dynamic` niteliğini de eklemeniz gerekir.) `callProperty()` yöntemini geçersiz kıldığınızda, `flash_proxy` ad alanını kullanmanız gerekir.

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

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

MyProxy sınıfının bir örneğini oluşturur ve aşağıdaki örnekte çağrılan `testing()` yöntemi gibi tanımsız bir yöntem çağırırsanız, Proxy nesneniz yöntem çağrısını önler ve geçersiz kılınan `callProperty()` yönteminin içinde deyimleri çalıştırır (bu durumda, basit bir `trace()` deyimi).

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

`flash_proxy` ad alanının içinde Proxy sınıfı yöntemlerinden birinin bulunmasının iki avantajı vardır. İlk olarak, ayrı bir ad alanının bulunması, Proxy sınıfını genişleten herhangi bir sınıfın genel arabiriminde karmaşıklığı azaltır. (Proxy sınıfında, geçersiz kılabileceğiniz yaklaşık bir düzine yöntem vardır ve bunların hiçbiri doğrudan çağrılmak üzere tasarlanmamıştır. Bunların tümünün genel ad alanına yerleştirilmesi kafa karıştırıcı olabilir.) İkinci olarak, `flash_proxy` ad alanının kullanılması, Proxy alt sınıfınızın herhangi bir Proxy sınıfı yöntemiyle eşleşen adlara sahip örnek yöntemler içermesi durumunda ad çakışmalarını önler. Örneğin, kendi yöntemlerinizden birine `callProperty()` adını vermek isteyebilirsiniz. `callProperty()` yöntemi sürümünüz farklı bir ad alanında olduğundan, aşağıdaki kod kabul edilebilir:

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

Dört erişim denetimi belirticisiyle (`public`, `private`, `internal` ve `protected`) gerçekleştirilemeyecek şekilde yöntemlere veya özelliklere erişim sağlamak istediğinizde de ad alanları yardımcı olabilir. Örneğin, birçok pakete yayılmış birkaç yardımcı program yönteminiz olabilir. Bu yöntemlerin, tüm paketleriniz için kullanılabilir olmasını ancak genel olarak herkese açık olmamasını istersiniz. Bu işlemi gerçekleştirmek için, bir ad alanı oluşturabilir ve bunu kendi özel erişim denetimi belirticiniz olarak kullanabilirsiniz.

Aşağıdaki örnekte, farklı paketlerde bulunan iki işlevi birlikte gruplandırmak için kullanıcı tanımlı bir ad alanı kullanılmaktadır. Bunları aynı ad alanında gruplandırarak tek bir `use namespace` deyimini kullanıp her iki işlevi bir sınıf veya paket için görünebilir duruma getirebilirsiniz.

Bu örnekte tekniği göstermek için dört dosya kullanılmaktadır. Tüm dosyaların sınıf yolunuzda yazılması gerekir. Birinci dosya olan myInternal.as, `myInternal` ad alanını tanımlamak için kullanılır. Dosya, example adındaki bir pakette bulunduğundan, dosyayı example adındaki bir klasöre yerleştirmeniz gerekir. Ad alanı, diğer paketlere içe aktarılabilmesi için `public` olarak işaretlenir.

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

İkinci ve üçüncü dosyalar olan Utility.as ve Helper.as, diğer paketler için kullanılabilir olması gereken yöntemlerin bulunduğu sınıfları tanımlar. Utility sınıfı, example.alpha paketindedir, başka bir deyişle, example klasörünün alt klasörü olan alpha adında bir klasörün içine dosyanın yerleştirilmesi gerekir. Helper sınıfı, example.beta paketindedir, başka bir deyişle, example klasörünün alt klasörü olan beta adında bir klasörün içine dosyanın yerleştirilmesi gerekir. Bu her iki example.alpha ve example.beta paketinin de ad alanını kullanmadan önce içe aktarması gerekir.

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

Dördüncü dosya olan NamespaceUseCase.as, ana uygulama sınıfı olup example klasörünün bir eşdüzeyi olması gerekir. Flash Professional'da bu sınıf, FLA'nın belge sınıfı olarak kullanılır. NamespaceUseCase sınıfı ayrıca `myInternal` ad alanını içe aktarır ve diğer paketlerde bulunan iki statik yöntemi çağırmak için bu ad alanını kullanır. Bu örnek yalnızca kodu basitleştirmek için statik yöntemleri kullanır. `myInternal` ad alanına hem statik yöntemler hem de örnek yöntemleri yerleştirilebilir.

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

Değişkenler

Değişkenler, programınızda kullandığınız değerleri saklamanıza olanak sağlar. Bir değişken bildirmek için, değişken adıyla `var` deyimini kullanmanız gerekir. ActionScript 3.0'da ise `var` deyiminin kullanılması her zaman gerekir. Örneğin, aşağıdaki ActionScript satırı, `i` adında bir değişken bildirir:

```
var i;
```

Bir değişkeni bildirirken `var` deyimini atarsanız, sıkı modda bir derleyici hatası ve standart modda bir çalışma zamanı hatası alırsınız. Örneğin, `i` değişkeni önceden tanımlanmadıysa, aşağıdaki kod satırı bir hataya yol açar:

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

Bir değişkeni bir veri türüyle ilişkilendirmek için, değişkeni bildirdiğinizde bunu yapmanız gerekir. Değişken türü belirlenmeden bir değişkenin belirtilmesi geçerlidir ancak bu durum, sıkı modda bir derleyici uyarısına yol açar. Değişkenin adının sonuna iki nokta (:) ve ardından da değişkenin türünü ekleyerek bir değişken türü belirlersiniz. Örneğin, aşağıdaki kod, int türünde bir `i` değişkenini bildirir:

```
var i:int;
```

Atama operatörünü (`=`) kullanarak bir değişkene bir değer atayabilirsiniz. Örneğin, aşağıdaki kod bir `i` değişkenini bildirir ve bu değişkene 20 değerini atar:

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

Aşağıdaki örnekte olduğu gibi, değişkeni bildirdiğiniz anda değişkene bir değer atamak daha kullanışlı olabilir:

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

Bildirildiği anda bir değişkene değer atama tekniği yalnızca tam sayı ve dizeler gibi ilkel değerler atanırken değil, bir dizi oluşturulurken veya bir sınıfın örneği başlatılırken de yaygın olarak kullanılır. Aşağıdaki örnek, bir kod satırı kullanılarak bildirilen ve değer atanan bir diziyi gösterir.

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

`new` operatörünü kullanarak bir sınıfın örneğini oluşturabilirsiniz. Aşağıdaki örnek, `CustomClass` adında bir sınıfın örneğini oluşturur ve yeni oluşturulan sınıf örneğinin başvurusunu `customItem` adındaki değişkene atar:

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

Bildirilecek birden çok değişkeniniz varsa, değişkenleri ayırmak için virgül operatörünü (`,`) kullanarak bunların tümünü tek bir kod satırında bildirebilirsiniz. Örneğin, aşağıdaki kod, tek bir kod satırında üç değişken bildirir:

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

Ayrıca aynı kod satırındaki değişkenlerin her birine değer atayabilirsiniz. Örneğin, aşağıdaki kod üç değişken (`a`, `b` ve `c`) bildirir ve her birine bir değer atar:

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

Değişken bildirimlerini tek bir deyimde gruplandırmak için virgül operatörünü kullanabilseniz de, bunun yapılması kodunuzun okunabilirliğini azaltabilir.

Değişken kapsamını anlama

Bir değişkenin *kapsamı*, sözlü bir başvuruyla değişkene erişilebilen kod alanıdır. *Genel* değişken, kodunuzun tüm alanlarında tanımlı değişkenken, *yerel* değişken ise kodunuzun yalnızca bir bölümünde tanımlı değişkendir. ActionScript 3.0'da değişkenlere her zaman bildirildikleri işlev veya sınıf kapsamı atanır. Genel bir değişken, herhangi bir işlev veya sınıf tanımı dışında tanımladığınız bir değişkendir. Örneğin, aşağıdaki kod, herhangi bir işlevin dışında genel işlevi bildirerek `strGlobal` genel işlevini oluşturur. Bu örnek, genel bir değişkenin, işlev tanımının içinde ve dışında kullanılabilir olduğunu gösterir.

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

Bir işlev tanımı içindeki değişkeni bildirerek yerel bir değişkeni bildirmiş olursunuz. Yerel değişkeni tanımlayabileceğiniz en küçük kod alanı, işlev tanımıdır. Bir işlev içinde bildirilen yerel değişken yalnızca o işlev içinde varolur. Örneğin, `localScope()` adındaki bir işlevin içinde `str2` adında bir değişkeni bildirirseniz, bu değişken söz konusu işlevin dışında kullanılamaz.

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

Yerel değişkeniniz için kullandığınız değişken adı önceden genel değişken olarak bildirilmişse, yerel değişken kapsamdayken, yerel tanım genel tanımı gizler (veya gölgeler). Genel değişken, işlev dışında varolmaya devam eder. Örneğin, aşağıdaki kod, `str1` adında genel bir dize değişkeni oluşturur ve sonra `scopeTest()` işlevinin içinde aynı adda yerel bir değişken oluşturur. İşlevin içindeki `trace` deyimi, değişkenin yerel değerini verirken, işlevin dışındaki `trace` deyimi de değişkenin genel değerini verir.

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

ActionScript değişkenleri, C++ ve Java uygulamalarındaki değişkenlerden farklı olarak, blok düzeyi kapsamına sahip değildir. Kod bloğu, açma küme parantezi ({) ile kapatma küme parantezi (}) arasındaki herhangi bir deyim grubudur. C++ ve Java gibi bazı programlama dillerinde, kod bloğu içinde bildirilen değişkenler o kod bloğunun dışında kullanılamaz. Bu kapsam sınırlamasına blok düzeyi kapsamı denir ve bu kapsam sınırlaması ActionScript'te bulunmaz. Bir kod bloğu içinde bir değişken bildirirseniz, bu değişken yalnızca o kod bloğunda değil, kod bloğunun ait olduğu diğer işlev parçalarında da kullanılabilir. Örneğin, aşağıdaki işlev, çeşitli blok kapsamlarında tanımlanmış değişkenleri içerir. Tüm değişkenler, işlev boyunca kullanılabilir.

```
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"]);
```

Blok düzeyi kapsamı olmamasının işaret ettiği ilginç bir nokta, işlev sona ermeden değişken bildirildiği sürece, bir değişken bildirilmeden o değişkene okuma ve yazma işlemi yapabilmenizdir. Bunun nedeni, derleyicinin tüm değişken bildirimlerini işlevin en üstüne taşıdığını belirten, *kaldırma* adındaki bir tekniktir. Örneğin, `num` değişkeninin ilk `trace()` işlevi, `num` değişkeni bildirilmeden önce olsa da, aşağıdaki kod derleme yapar:

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

Ancak derleyici, atama deyimlerini kaldırmaz. Bu da, `num` değişkeninin ilk `trace()` işlevinin neden Number veri türü için varsayılan değişken değeri olan `NaN` (sayı değil) sonucunu verdiğini açıklar. Başka bir deyişle, aşağıdaki örnekte gösterildiği gibi, değişkenler bildirilmeden değişkenlere değer atayabilirsiniz:

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

Varsayılan değerler

Varsayılan değer, siz değerini ayarlamadan önce bir değişkenin içerdiği değerdir. Bir değişkenin ilk defa değerini ayarladığınızda o değişkeni *başlatırsınız*. Bir değişkeni bildirir ancak değişkenin değerini ayarlamazsanız, o değişken *başlatılmamış* olur. Başlatılmamış değişkenin değeri, veri türüne bağlıdır. Aşağıdaki tabloda, veri türüne göre organize edilmiş şekilde değişkenlerin varsayılan değerleri açıklanmaktadır:

Veri türü	**Varsayılan değer**
Boolean	`false`
int	0
Number	`NaN`
Nesne	`null`
Dize	`null`

Veri türü	Varsayılan değer
uint	0
Bildirilmemiş (tür ek açıklamasına eşit `*`)	`undefined`
Kullanıcı tanımlı sınıflar da dahil olmak üzere diğer tüm sınıflar.	`null`

Number türündeki değişkenler için varsayılan değer `NaN` (sayı değil) olup bu değer, bir değerin sayıyı temsil etmediğini belirtmek için IEEE-754 standardı tarafından tanımlanmış özel bir değerdir.

Bir değişkeni bildirir ancak değişkenin veri türünü bildirmezseniz, gerçekten değişkenin türe sahip olmadığını belirten varsayılan veri türü `*` uygulanır. Türlenmemiş bir değişkeni bir değerle de başlatmazsanız, değişkenin varsayılan değeri `undefined` olur.

Boolean, Number, int ve uint dışındaki veri türleri için, başlatılmamış bir değişkenin varsayılan değeri `null` olur. Bu, ActionScript 3.0 tarafından tanımlanmış tüm sınıfların yanı sıra sizin oluşturduğunuz özel sınıflar için de geçerlidir.

`null` değeri, Boolean, Number, int veya uint türündeki değişkenler için geçerli bir değer değildir. Bu tür bir değişkene `null` değeri atamayı denerseniz, değer o veri türünün varsayılan değerine dönüştürülür. Object türündeki değişkenler için bir `null` değeri atayabilirsiniz. Object türündeki bir değişkene `undefined` değerini atamayı denerseniz, değer `null` değerine dönüştürülür.

Number türündeki değişkenler için, değişken bir sayı olmadığında `true`, aksi takdirde `false` Boolean değerini döndüren `isNaN()` adında özel bir üst düzey işlevi vardır.

Veri türleri

Veri türü, bir değerler kümesini tanımlar. Örneğin, Boolean veri türü tam olarak iki değerden oluşan bir kümedir: `true` ve `false`. ActionScript 3.0, Boolean veri türüne ek olarak, String, Number ve Array gibi birkaç tane daha yaygın kullanılan veri türünü tanımlar. Özel bir değer kümesini tanımlamak için sınıfları veya arabirimleri kullanarak kendi veri türlerinizi tanımlayabilirsiniz. İlkel veya karmaşık olsun, ActionScript 3.0'daki tüm değerler nesnedir.

İlkel değer, şu veri türlerinden birine ait olan bir değerdir: Boolean, int, Number, String ve uint. ActionScript, bellek ve hız eniyileştirmesini olanaklı kılacak şekilde ilkel değerleri sakladığından, ilkel değerlerle çalışılması genellikle karmaşık değerlerle çalışılmasından daha hızlıdır.

Not: *ActionScript, teknik ayrıntılara ilgi duyan okuyucular için ilkel değerleri sabit nesneler şeklinde dahili olarak saklar. Bunların sabit nesneler olarak saklanması, başvuruya göre iletmenin değere göre iletme kadar etkili olduğu anlamına gelir. Başvurular genellikle değerlerin kendisinden çok daha küçük olduğundan bu, bellek kullanımını azaltıp çalıştırma hızını da artırır.*

Karmaşık değer, ilkel olmayan bir değerdir. Karmaşık değerler kümesini tanımlayan veri türleri arasında, Array, Date, Error, Function, RegExp, XML ve XMLList yer alır.

Çoğu programlama dili, ilkel değerler ile bunların sarıcı değerleri arasında ayrım yapar. Örneğin, Java, bir int ilkel değerine ve bunu saran java.lang.Integer sınıfına sahiptir. Java ilkel değerleri nesne değildir ancak bunların sarıcıları nesnedir ve bu da bazı işlemler için ilkel değerleri, bazı işlemler için de sarıcı nesneleri daha uygun hale getirir. ActionScript 3.0'da ilkel değerler ve bunların sarıcı nesneleri pratik amaçlar için birbirinden ayırt edilemez. İlkel değerler de dahil olmak üzere tüm değerler nesnedir. Çalışma zamanı, bu ilkel türleri, nesne gibi davranan ancak nesne oluşturmayla ilişkili normal yükü gerektirmeyen özel durumlar olarak değerlendirir. Başka bir deyişle, aşağıdaki iki kod satırı eşdeğerdir:

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

Yukarıda listelenen tüm ilkel ve karmaşık veri türleri, ActionScript 3.0 çekirdek sınıfları tarafından tanımlanır. Çekirdek sınıflar, `new` operatörü yerine değişmez değerler kullanarak nesneler oluşturmanıza olanak sağlar. Örneğin, aşağıdaki gibi, değişmez bir değer veya Array sınıfı yapıcısını kullanarak bir dizi oluşturabilirsiniz:

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

Tür denetleme

Tür denetleme, derleme zamanında veya çalışma zamanında gerçekleşebilir. C++ ve Java gibi statik olarak türlenmiş diller, tür denetlemesini derleme zamanında yapar. Smalltalk ve Python gibi dinamik olarak türlenmiş diller, tür denetlemesini çalışma zamanında işler. ActionScript 3.0, dinamik olarak türlenmiş bir dil olarak çalışma zamanı tür denetlemesi yapar ancak *katı mod* adı verilen özel bir derleyici moduyla da derleme zamanı tür denetlemesini destekler. Katı modda tür denetlemesi hem derleme zamanında hem de çalışma zamanında gerçekleşir ancak standart modda, tür denetlemesi yalnızca çalışma zamanında gerçekleşir.

Dinamik olarak türlenmiş diller, siz kodunuzu yapılandırırken çok yüksek esneklik sunar ancak bu, çalışma zamanında tür hatalarının verilmesine neden olur. Statik olarak türlenmiş diller, tür hatalarını derleme zamanında bildirir ancak bu da tür bilgilerinin derleme zamanında bilinmesini gerektirir.

Derleme zamanı tür denetlemesi

Proje boyutu arttıkça, olabildiğince erkenden tür hatalarının yakalanmasının önemi artıp veri türü esnekliğinin önemi azaldığından, derleme zamanı tür denetlemesi genellikle daha büyük projelerde kullanışlıdır. Bu nedenle de Flash Professional ve Flash Builder uygulamalarında ActionScript derleyicisi varsayılan olarak sıkı modda çalışacak şekilde ayarlanmıştır.

Adobe Flash Builder

Flash Builder uygulamasında, Proje Özellikleri iletişim kutusunda ActionScript derleyici ayarları üzerinden sıkı modu devre dışı bırakabilirsiniz.

Derleme zamanı tür denetlemesi sağlamak için, derleyicinin kodunuzdaki değişkenlerin veya ifadelerin veri türü bilgilerini bilmesi gerekir. Bir değişkenin veri türünü açıkça bildirmek için, değişken adına son ek olarak iki nokta operatörünü (`:`) ve ardından veri türünü ekleyin. Veri türünü bir parametreyle ilişkilendirmek için, iki nokta operatörünü ve ardından da veri türünü kullanın. Örneğin, aşağıdaki kod, `xParam` parametresine veri türü bilgilerini ekler ve açık bir veri türüyle bir değişkeni `myParam` olarak bildirir:

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

ActionScript derleyicisi, katı modda tür uyuşmazlıklarını derleyici hataları olarak bildirir. Örneğin, aşağıdaki kod, Object türünde bir işlev parametresini `xParam` olarak bildirir, ancak daha sonra bu parametreye String ve Number türünde değerler atamayı dener. Bu da katı modda bir derleyici hatası oluşturur.

```
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);
    }
}
```

Ancak katı modda da, atama deyiminin sağ tarafını türlenmemiş şekilde bırakarak derleme zamanı tür denetlemesini seçerek devre dışı bırakabilirsiniz. Bir tür ek açıklamasını çıkararak veya özel yıldız (`*`) tür ek açıklamasını kullanarak bir değişkeni ya da ifadeyi türlenmemiş olarak işaretleyebilirsiniz. Örneğin, önceki örnekte yer alan `xParam` parametresi, artık tür ek açıklaması içermeyecek şekilde değiştirilirse, kod sıkı modda derlenir:

```
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");
```

Çalışma zamanı tür denetlemesi

Katı modda da, standart modda da derleme yapsanız, ActionScript 3.0'da, çalışma zamanı tür denetlemesi gerçekleşir. Bir dizi bekleyen bir işleve argüman olarak 3 değerinin iletildiği bir durumu göz önünde bulundurun. 3 değeri, Array veri türüyle uyumlu olmadığından, katı modda derleyici bir hata oluşturur. Sıkı modu devre dışı bırakıp standart modda çalışırsanız, derleyici tür uyuşmazlığından şikayetçi olmaz ancak çalışma zamanı tür denetlemesi bir çalışma zamanı hatası verir.

Aşağıdaki örnek, bir Array argümanı bekleyen ancak kendisine 3 değeri iletilen `typeTest()` adındaki bir işlevi gösterir. 3 değeri, parametrenin bildirilen veri türünün (Array) bir üyesi olmadığından bu, standart modda bir çalışma zamanı hatasına yol açar.

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

Katı modda çalıştığınız halde çalışma zamanı tür hatası aldığınız durumlar da olabilir. Katı modu kullanırken türlenmemiş bir değişken kullanarak derleme zamanı tür denetlemesini devre dışı bırakırsanız bu mümkün olur. Türlenmemiş bir değişken kullandığınızda, tür denetlemesini ortadan kaldırmış olmazsınız yalnızca çalışma zamanına ertelemiş olursunuz. Örneğin, önceki örnekte yer alan `myNum` değişkeninin bildirilmiş bir veri türü yoksa, derleyici tür uyuşmazlığını algılayamaz; ancak kod, atama deyiminin sonucu olarak 3 değerine ayarlanan `myNum` çalışma zamanı değerini Array veri türüne ayarlanmış `xParam` türüyle karşılaştırdığından, bir çalışma zamanı hatası oluşturur.

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

Çalışma zamanı tür denetlemesi ayrıca derleme zamanı tür denetlemesine göre mirasın daha esnek kullanılmasını sağlar. Standart mod, tür denetlemesini çalışma zamanına erteleyerek, *yukarı çevrim* de yapsanız bir alt sınıfın özelliklerine başvurmanıza olanak sağlar. Bir sınıf örneğinin türünü belirtmek için bir temel sınıf kullanıp bu sınıf türü örneğini başlatmak için bir alt sınıf kullandığınızda yukarı çevrim gerçekleşir. Örneğin, genişletilebilen ClassBase adında bir sınıf oluşturabilirsiniz (`final` niteliğine sahip sınıflar genişletilemez):

```
class ClassBase
{
}
```

Daha sonra aşağıdaki gibi, ClassExtender adında olan ve `someString` adında bir özellik içeren, ClassBase sınıfının bir alt sınıfını oluşturabilirsiniz:

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

Her iki sınıfı da kullanıp ClassBase veri türü kullanılarak bildirilen ancak ClassExtender yapıcısı kullanılarak başlatılan bir sınıf örneği oluşturabilirsiniz. Temel sınıf, alt sınıfta bulunmayan özellikleri veya yöntemleri içermediğinden, yukarı çevrim güvenli bir işlem olarak değerlendirilir.

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

Ancak bir alt sınıf, temel sınıfının içermediği özellikleri veya yöntemleri içermez. Örneğin, ClassExtender sınıfı, ClassBase sınıfında varolmayan `someString` özelliğini içerir. ActionScript 3.0 standart modunda, aşağıdaki örnekte gösterildiği gibi, bir derleme zamanı hatası oluşturmadan `myClass` örneğini kullanarak bu özelliğe başvurabilirsiniz:

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

is operatörü

`is` operatörü, bir değişkenin veya ifadenin belirli bir veri türünün üyesi olup olmadığını test etmenize olanak tanır. Önceki ActionScript sürümlerinde, `instanceof` operatörü bu işlevselliği sağlamıştır ancak ActionScript 3.0'da `instanceof` operatörü, veri türü üyeliğini test etmek için kullanılmamalıdır. `x instanceof y` ifadesi yalnızca `y` varlığı için `x` prototip zincirini denetlediğinden, elle tür denetleme için `instanceof` operatörü yerine `is` operatörü kullanılmalıdır. (Ayrıca ActionScript 3.0'da prototip zinciri, miras hiyerarşisinin tam resmini sağlamaz.)

`is` operatörü uygun miras hiyerarşisini inceler ve yalnızca bir nesnenin belirli bir sınıfın örneği olup olmadığını denetlemek için değil, bir nesnenin belirli bir arabirimi uygulayan bir sınıf örneği olup olmadığını denetlemek için de kullanılabilir. Aşağıdaki örnek, `mySprite` adında bir Sprite sınıfı örneği oluşturur ve `mySprite` öğesinin Sprite ve DisplayObject sınıflarının bir örneği olup olmadığını ve IEventDispatcher arabirimini uygulayıp uygulamadığını test etmek için `is` operatörünü kullanır:

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

`is` operatörü, miras hiyerarşisini denetler ve `mySprite` örneğinin Sprite ve DisplayObject sınıflarıyla uyumlu olduğunu düzgün şekilde bildirir. (Sprite sınıfı, DisplayObject sınıfının bir alt sınıfıdır.) `is` operatörü ayrıca `mySprite` öğesinin IEventDispatcher arabirimini uygulayan herhangi bir sınıftan miras alıp almadığını da denetler. Sprite sınıfı, IEventDispatcher arabirimini uygulayan IEventDispatcher sınıfından miras aldığından, `is` operatörü, `mySprite` öğesinin aynı arabirimini uyguladığını doğru şekilde bildirir.

Aşağıdaki örnek, `is` operatörü yerine `instanceof` operatörünü kullanarak önceki örnekteki testlerin aynısını gösterir. `instanceof` operatörü, `mySprite` öğesinin Sprite veya DisplayObject öğesinin bir örneği olduğunu doğru şekilde tanımlar ancak `mySprite` öğesinin IEventDispatcher arabirimini uygulayıp uygulamadığını test etmek için kullanıldığında `false` değerini döndürür.

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

as operatörü

`as` operatörü, bir ifadenin belirli bir veri türünün üyesi olup olmadığını denetlemenize olanak tanır. Ancak `is` operatöründen farklı olarak `as` operatörü bir Boolean değeri döndürmez. `as` operatörü, `true` yerine ifadenin değerini ve `false` yerine de `null` değerini döndürür. Aşağıdaki örnek, Sprite örneğinin DisplayObject, IEventDispatcher ve Number veri türlerinden hangisinin üyesi olduğunu denetleme gibi basit bir durumda `is` operatörü yerine `as` operatörü kullanılmasının sonuçlarını gösterir.

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

`as` operatörünü kullandığınızda, sağdaki işlenenin bir veri türü olması gerekir. Sağdaki işlenen olarak veri türü dışında bir ifade kullanma girişimi hataya yol açar.

Dinamik sınıflar

Dinamik sınıf, özellikler ve yöntemler eklenerek veya değiştirilerek çalışma zamanında değiştirilebilen bir nesneyi tanımlar. String sınıfı gibi, dinamik olmayan bir sınıf *mühürlenmiş* bir sınıftır. Mühürlenmiş bir sınıfa çalışma zamanında özellikler veya yöntemler ekleyemezsiniz.

Bir sınıfı bildirirken, `dynamic` niteliğini kullanarak dinamik sınıflar oluşturursunuz. Örneğin, aşağıdaki kod, `Protean` adında dinamik bir sınıf oluşturur:

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

Daha sonra `Protean` sınıfının bir örneğini başlatırsanız, sınıf tanımının dışında buna özellikler veya yöntemler ekleyebilirsiniz. Örneğin, aşağıdaki kod, `Protean` sınıfının bir örneğini oluşturur ve örneğe `aString` adında bir özellik ve `aNumber` adında bir özellik ekler:

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

Dinamik sınıf örneğine eklediğiniz özellikler çalışma zamanı varlıklardır, bu nedenle tüm denetlemeleri çalışma zamanında yapılır. Bu şekilde eklediğiniz bir özelliğe tür ek açıklaması ekleyemezsiniz.

Ayrıca bir işlev tanımlayıp bu işlevi `myProtean` örneğinin bir özelliğine ekleyerek `myProtean` örneğine bir yöntem ekleyebilirsiniz. Aşağıdaki kod, izleme deyimini `traceProtean()` adındaki bir yönteme taşır:

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

Ancak bu şekilde oluşturulan yöntemlerin, Protean sınıfının özel özelliklerine veya yöntemlerine erişimi yoktur. Ayrıca, `Protean` sınıfının genel özelliklerine veya yöntemlerine başvuruların da `this` anahtar sözcüğüyle ya da sınıf adıyla nitelenmesi gerekir. Aşağıdaki örnek, `Protean` sınıfının özel ve genel değişkenlerine erişmeye çalışan `traceProtean()` yöntemini gösterir.

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

Veri türleri açıklamaları

İlkel veri türleri arasında Boolean, int, Null, Number, String, uint ve void yer alır. ActionScript çekirdek sınıfları ayrıca şu karmaşık veri türlerini de tanımlar: Object, Array, Date, Error, Function, RegExp, XML ve XMLList.

Boolean veri türü

Boolean veri türü iki değer içerir: `true` ve `false`. Boolean türündeki değişkenler için diğer değerler geçerli değildir. Bildirilmiş ancak başlatılmamış bir Boolean değişkeninin varsayılan değeri `false` olur.

int veri türü

int veri türü, dahili olarak 32 bit tam sayı şeklinde saklanır ve

-2.147.483.648 (-2^{31}) ile 2.147.483.647 ($2^{31} - 1$) (dahil) arasındaki tam sayılar kümesini kapsar. Önceki ActionScript sürümleri yalnızca hem tam sayı hem de kayan nokta sayıları için kullanılan Number veri türünü sunardı. ActionScript 3.0'da şimdi 32-bit işaretli ve işaretsiz tam sayılar için düşük düzeyli makine türlerine erişiminiz vardır. Değişkeninizin kayan nokta sayılarını kullanmasına gerek yoksa, Number veri türü yerine int veri türünün kullanılması hem daha hızlı hem de daha etkili olur.

Minimum ve maksimum int değerleri aralığı dışında kalan tam sayı değerleri için, pozitif ve negatif 9.007.199.254.740.992 arasındaki değerleri (53-bit tam sayı değerleri) işleyebilen Number veri türünü kullanın. int veri türündeki değişkenler için varsayılan değer 0'dır.

Null veri türü

Null veri türü yalnızca bir değer içerir, `null`. Bu, String veri türü için ve Object sınıfı da dahil olmak üzere karmaşık veri türlerini tanımlayan tüm sınıflar için varsayılan değerdir. Boolean, Number, int ve uint gibi diğer ilkel veri türlerinin hiçbiri `null` değerini içermez. Çalışma zamanında,`null` değeri, Boolean, Number, int, veya uint gibi değişken türlerine `null` atamaya çalışırsanız uygun varsayılan değere dönüştürülür. Bu veri türünü tür ek açıklaması olarak kullanamazsınız.

Number veri türü

ActionScript 3.0'da Number veri türü, tam sayıları, işaretsiz tam sayıları ve kayan nokta sayılarını temsil edebilir. Ancak, performansı en üst düzeye çıkarmak için, Number veri türünü yalnızca 32-bit `int` ve `uint` türlerinin saklayamayacağı kadar büyük tam sayı değerleri için ve kayan nokta sayıları için kullanmanız gerekir. Bir kayan nokta sayısını saklamak için, sayıya bir ondalık işareti dahil edin. Ondalık işaretini çıkarırsanız, sayı bir tam sayı olarak saklanır.

Number veri türü, İkili Kayan Nokta Aritmetiği için IEEE Standardı (IEEE-754) tarafından belirtilen 64-bit çift kesinlikli formatı kullanır. Bu standart, kayan nokta sayılarının mevcut 64 bit kullanılarak nasıl saklanacağını dikte eder Sayının pozitif veya negatif olduğunu belirlemek için tek bir bit kullanılır. Taban 2 olarak saklanan üs için on bir bit kullanılır. Kalan 52 bit, üssün belirttiği kuvvete yükseltilen sayı olan *significand* öğesini (*mantissa* olarak da adlandırılır) saklamak için kullanılır.

Number veri türü, bir üssü saklamak için bit'lerinden bir kısmını kullanarak, significand için tüm bit'leri kullandığında saklayabildiğinden çok daha büyük kayan nokta sayılarını saklayabilir. Örneğin, Number veri türü, significand öğesini saklamak için tüm 64 bit'i de kullansaydı, $2^{65} - 1$ büyüklüğünde bir sayıyı saklayabilirdi. Number veri türü bir üssü saklamak için 11 bit kullanarak, significand öğesini 2^{1023} kuvvetine yükseltebilir.

Number türünün temsil edebildiği maksimum ve minimum değerler, Number sınıfının `Number.MAX_VALUE` ve `Number.MIN_VALUE` adındaki statik özelliklerinde saklanır.

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

Bu sayı aralığı büyük olsa da, bu geniş aralık kesinlik düzeyini azaltır. Number veri türü, significand öğesini saklamak için 52 bit kullanır ve bu da, kesin şekilde temsil edilecek 52 bit'ten fazlasını gerektiren sayıların (örn. 1/3 kesiri) yalnızca yaklaşık değerler olmasına neden olur. Uygulamanız için ondalık sayılarda mutlak kesinlik gerekiyorsa, ikili kayan nokta aritmetiğinin tersine ondalık kayan nokta aritmetiğini uygulayan bir yazılım kullanmanız gerekir.

Number veri türünde tam sayı değerlerini sakladığınızda, significand öğesinin yalnızca 52 bit'i kullanılır. Number veri türü, -9.007.199.254.740.992 (-2^{53}) ile 9.007.199.254.740.992 (2^{53}) arasındaki tam sayıları temsil etmek için bu 52 bit'i ve özel gizlenmiş bit'i kullanır.

`NaN` değeri yalnızca `Number` türündeki değişkenler için varsayılan değer olarak değil, aynı zamanda bir sayı döndürmesi gerektiği halde sayı döndürmeyen herhangi bir işlemin sonucu olarak da kullanır. Örneğin, negatif bir sayının kare kökünü hesaplamayı denerseniz, sonuç `NaN` olur. Diğer özel Number değerleri arasında *pozitif sonsuzluk* ve *negatif sonsuzluk* yer alır.

Not: *`0` değerine bölme işleminin sonucu, yalnızca bölen de `0` olduğunda `NaN` değerini verir. `0` değerine bölme işlemi, bölen pozitif olduğunda `infinity` değerini, bölen negatif olduğunda ise `-infinity` değerini verir.*

String veri türü

String veri türü, 16-bit karakterlerin bir sırasını temsil eder. Dizeler, UTF-16 formatı kullanılarak dahili şekilde Unicode karakterleri olarak saklanır. Dizeler, Java programlama dilinde olduğu gibi, sabit değerlerdir. String değerindeki bir işlem, yeni bir dize örneği döndürür. String veri türüyle bildirilen bir değişkenin varsayılan değeri, `null` şeklindedir. `null` değeri boş dizeyle (`""`) aynı değildir. `null` değeri değişkenin içinde bir değer saklamadığı anlamına gelirken, boş dize, değerin hiçbir karakter içermeyen bir String olan bir değer içerdiği anlamına gelir.

uint veri türü

uint veri türü, dahili olarak 32-bit işaretsiz tam sayı şeklinde saklanır ve 0 ile 4.294.967.295 (2^{32} - 1) arasındaki tam sayıların kümesini kapsar. Negatif olmayan tam sayıları çağıran özel koşullar için uint veri türünü kullanın. Örneğin, int veri türü, renk değerlerinin işlenmesi için uygun olmayan dahili bir işaret bit'i içerdiğinden, piksel rengi değerlerini temsil etmek için uint veri türünü kullanmanız gerekir. Maksimum uint değerinden büyük tam sayı değerleri için, 53-bit tam sayı değerlerini işleyebilen Number veri türünü kullanın. uint veri türündeki değişkenler için varsayılan değer 0'dır.

void veri türü

Void veri türü yalnızca bir değer içerir: `undefined`. Önceki ActionScript sürümlerinde `undefined`, Object sınıfının örnekleri için varsayılan değerdi. ActionScript 3.0'da, Object örneklerinin varsayılan değeri `null` şeklindedir. Object sınıfının bir örneğine `undefined` değerini atamayı denerseniz, değer `null` değerine dönüştürülür. Türlenmemiş değişkenlere yalnızca `undefined` değerini atayabilirsiniz. Türlenmemiş değişkenler, tür ek açıklaması içermeyen veya tür ek açıklaması için yıldız (`*`) sembolünü kullanan değişkenlerdir. `void` öğesini yalnızca döndürme tür ek açıklaması olarak kullanabilirsiniz.

Object veri türü

Object veri türü, Object sınıfı tarafından tanımlanır. Object sınıfı, ActionScript'teki tüm sınıf tanımlamaları için temel sınıf görevi görür. Object veri türünün ActionScript 3.0 sürümü, önceki sürümlerden üç şekilde farklılık gösterir. İlk olarak, Object veri türü artık tür ek açıklaması içermeyen değişkenlere atanan varsayılan veri türü değildir. İkinci olarak, Object veri türü, Object örneklerinin varsayılan değeri olarak kullanılan `undefined` değerini artık içermez. Üçüncü olarak, ActionScript 3.0'da, Object sınıfının örnekleri için varsayılan değer `null` şeklindedir.

Önceki ActionScript sürümlerinde, Object veri türüne tür ek açıklaması içermeyen bir değişken otomatik olarak atanırdı. Artık türlenmemiş değişken kavramını içeren ActionScript 3.0'da bu geçerli değildir. Tür ek açıklaması içermeyen değişkenler artık türlenmemiş olarak değerlendirilir. Kodunuzun okuyucularına, amacınızın bir değişkene tür vermemek olduğunu açıkça belirtmek isterseniz, tür ek açıklamasının çıkarılmasına eşdeğer şekilde, tür ek açıklaması için yıldız (`*`) sembolünü kullanabilirsiniz. Aşağıdaki örnekte, her ikisi de `x` türlenmemiş değişkenini bildiren iki eşdeğer deyim gösterilmektedir:

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

Yalnızca türlenmemiş değişkenler `undefined` değerini barındırabilir. Bir veri türüne sahip değişkene `undefined` değerini atamaya çalışırsanız, çalışma zamanı, `undefined` değerini o veri türünün varsayılan değerine dönüştürür. Object veri türü örnekleri için, varsayılan değer `null` değeridir. Bu durum, `undefined` değerini bir Object örneğine atamaya çalışırsanız, değerin `null` değerine dönüştürüleceği anlamına gelir.

Tür dönüştürmeleri

Bir değer farklı veri türünde bir değere dönüştürüldüğünde, tür dönüştürmesi gerçekleştirilmiş olur. Tür dönüştürmeleri *örtük* veya *açıkça* olabilir. *coercion* olarak da bilinen örtük dönüştürme, bazen çalışma zamanında gerçekleştirilir. Örneğin, Boolean veri türündeki bir değişkene 2 değeri atanırsa, değer değişkene atanmadan önce 2 değeri `true` Boolean değerine dönüştürülür. *Çevrim* olarak da adlandırılan açıkça dönüştürme, kodunuz derleyiciye bir veri türündeki değişkeni farklı bir veri türüne aitmiş gibi değerlendirmesini bildirdiğinde gerçekleşir. İlkel değerler bulunduğunda, çevrim gerçek anlamda değerleri bir veri türünden diğerine dönüştürür. Bir nesneyi farklı bir türe çevirmek için, nesne adını parantez içine alıp ve bunun başına yeni türün adını getirirsiniz. Örneğin, aşağıdaki kod bir Boolean değerini alıp tam sayıya çevirir:

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

Örtük dönüştürmeler

Örtük dönüştürmeler, birçok bağlamda çalışma zamanında gerçekleşir:

- Atama deyimlerinde
- Değerler işlev argümanları olarak iletildiğinde
- Değerler işlevlerden döndürüldüğünde
- Toplama (+) operatörü gibi belirli operatörleri kullanan ifadelerde

  Kullanıcı tanımlı türler için örtük dönüştürmeler, dönüştürülecek değer hedef sınıfın bir örneği veya hedef sınıftan türetilmiş bir sınıf olduğunda gerçekleşir. Örtük dönüştürme başarısız olursa bir hata oluşur. Örneğin, aşağıdaki kod başarılı bir örtük dönüştürme ve başarısız bir örtük dönüştürme içerir:

  ```
  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.
  ```

  İlkel türler için örtük dönüştürmeler, açıkça dönüştürme işlevleri tarafından çağrılan aynı dahili dönüştürme algoritmaları çağrılarak işlenir.

Açıkça dönüştürmeler

Derleme zamanı hatası oluşturacak bir tür uyuşmazlığı istemediğiniz zamanlar olabileceğinden, katı modda derleme yaparken açıkça dönüştürmeleri veya çevrimi kullanmanız yararlı olur. Zorlamanın değerlerinizi çalışma zamanında doğru şekilde dönüştüreceğini bildiğinizde bu durum geçerli olabilir. Örneğin, bir formdan alınan verilerle çalıştığınızda, belirli dize değerlerini sayısal değerlere dönüştürmek için zorlamayı uygulamak isteyebilirsiniz. Aşağıdaki kod, standart modda doğru şekilde de çalışsa, bir derleme zamanı hatası oluşturur:

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

Katı modu kullanmaya devam etmek ancak bir yandan da dizenin bir tam sayıya dönüştürülmesini istiyorsanız, aşağıdaki gibi açıkça dönüştürmeyi kullanabilirsiniz:

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

int, uint ve Number türlerine çevrim

Herhangi bir veri türünü üç sayı türünden birine çevirebilirsiniz: int, uint ve Number. Number herhangi bir sebepten dönüştürülemezse, 0 varsayılan değeri int ve uint veri türleri için atanır ve `NaN` varsayılan değeri Number veri türü için atanır. Bir Boolean değerini bir sayıya dönüştürürseniz, `true` değeri 1 ve `false` değeri de 0 olur.

```
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
```

Yalnızca rakam içeren dize değerleri, sayı türlerinden birine başarıyla dönüştürülebilir. Sayı türleri ayrıca negatif sayı gibi görünen dizeleri veya onaltılık bir değeri (örneğin, `0x1A`) temsil eden dizeleri dönüştürebilir. Dönüştürme işlemi, dize değerinin başındaki ve sonundaki boşluk karakterlerini yoksayar. Ayrıca `Number()` öğesini kullanarak kayan nokta sayısı gibi görünen dizeleri de çevirebilirsiniz. Ondalık işareti eklenmesi, `uint()` ve `int()` öğelerinin, ondalık işaretini ve bu işareti takip eden karakterleri kırparak bir tam sayı döndürmesine neden olur. Örneğin, aşağıdaki dize değerleri sayılara çevrilebilir:

```
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
```

Sayısal olmayan karakterler içeren dize değerleri, `int()` veya `uint()` ile çevrildiğinde 0 değerini; `Number()` ile çevrildiğinde ise `NaN` değerini döndürür. Dönüştürme işlemi, baştaki ve sondaki boşlukları yoksayar ancak dizede iki sayıyı ayıran bir boşluk varsa, 0 veya `NaN` değerini döndürür.

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

ActionScript 3.0'da, `Number()` işlevi artık sekizlik veya 8 tabanlı sayıları desteklemez. ActionScript 2.0 `Number()` işlevine başında sıfır bulunan bir dize sağlarsanız, sayı sekizlik bir sayı olarak yorumlanır ve ondalık eşdeğerine dönüştürülür. ActionScript 3.0'daki `Number()` işlevinde ise bu geçerli değildir, burada baştaki sıfır yoksayılır. Örneğin, aşağıdaki kod, farklı ActionScript sürümleri kullanılarak derlendiğinde farklı çıktı oluşturur:

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

Bir sayısal türdeki değer, farklı bir sayısal türdeki değişkene atandığında çevrim gerekmez. Katı modda da sayısal türler örtük olarak başka sayısal türlere dönüştürülür. Başka bir deyişle, bazı durumlarda bir tür aralığı aşıldığında beklenmeyen değerler ortaya çıkabilir. Aşağıdaki değerlerin bazıları beklenmeyen değerler oluştursa da, tümü sıkı modda derlenir:

```
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
```

Aşağıdaki tabloda, başka veri türlerinden Number, int veya uint veri türüne çevrim sonuçları özetlenmektedir.

Veri türü veya değeri	Number, int veya uint türüne dönüştürme sonucu
Boolean	Değer `true` olursa 1; aksi takdirde 0.
Date	Date nesnesinin dahili temsili; bu, 1 Ocak 1970, gece yarısı evrensel saatinden bu yana geçen milisaniye sayısıdır.
`null`	0
Nesne	Örnek `null` olursa ve Number türüne dönüştürülürse `NaN`; aksi takdirde 0.
Dize	Dize bir number türüne dönüştürülebilirse bir sayı, aksi takdirde Number türüne dönüştürülürse `NaN` veya int ya da uint türüne dönüştürülürse 0.
`undefined`	Number türüne dönüştürülürse `NaN`; int veya uint türüne dönüştürülürse 0.

Boolean değerine çevrim

Herhangi bir sayısal veri türünden (uint, int ve Number) Boolean değerine çevrim, sayısal değer 0 olursa `false`, aksi takdirde `true` değerini verir. Number veri türü için, `NaN` değeri de `false` değerini verir. Aşağıdaki örnek, -1, 0 ve 1 sayılarının çevrim sonuçlarını gösterir:

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

Örnekten elde edilen çıktı, üç sayıdan yalnızca 0 sayısının `false` değeri döndürdüğünü gösterir:

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

Bir String değerinden Boolean değerine çevrim, dize `null` veya boş dize (`""`) olduğunda `false` değerini döndürür. Aksi takdirde, `true` değerini döndürür.

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

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

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

Object sınıfı örneğinden Boolean değerine çevrim, örnek `null` ise `false` değerini; aksi takdirde `true` değerini döndürür:

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

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

Boolean değişkenleri katı modda özel değerlendirmeye tabidir; katı modda çevrim yapmadan herhangi bir veri türündeki değerleri Boolean değişkenine atayabilirsiniz. Tüm veri türlerinden Boolean veri türüne örtük zorlama katı modda da gerçekleşir. Başka bir deyişle, diğer tüm veri türlerinin hemen hemen hepsinden farklı olarak, katı mod hatalarını önlemek için Boolean değerine çevrim gerekmez. Aşağıdaki örneklerin tümü katı modda derleme yapar ve çalışma zamanında beklendiği şekilde davranır:

```
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
```

Aşağıdaki tabloda, başka veri türlerinden Boolean veri türüne çevrim sonuçları özetlenmektedir:

Veri türü veya değeri	**Boolean değerine dönüştürme sonucu**
Dize	Değer `null` veya boş dize (`""`) olursa `false`; aksi takdirde `true`.
`null`	`false`
Number, int veya uint	Değer `NaN` veya 0 olursa `false`; aksi takdirde `true`.
Nesne	Örnek `null` olursa `false`; aksi takdirde `true`.

String türüne çevrim

Herhangi bir sayısal veri türünden String veri türüne çevrim, sayının dize halinde temsilini döndürür. Bir Boolean değerinden String veri türüne çevrim, değer `true` olursa `"true"` dizesini ve değer `false` olursa `"false"` dizesini döndürür.

Bir Object sınıfı örneğinden String veri türüne çevrim, örnek `null` olursa `"null"` dizesini döndürür. Aksi takdirde, Object sınıfından String türüne çevrim, `"[object Object]"` dizesini döndürür.

Array sınıfı örneğinden String türüne çevrim, tüm dizi öğelerinin virgülle ayrılmış bir listesinden oluşan bir dize döndürür. Örneğin, aşağıdaki String veri türüne çevrim işlemi, dizideki üç öğeyi de içeren tek bir dize döndürür:

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

Date sınıfı örneğinden String türüne çevrim, örneğin içerdiği tarihin dize halinde temsilini döndürür. Örneğin, aşağıdaki örnek, Date sınıfı örneğinin dize halinde temsilini döndürür (çıktıda Pasifik Yaz Saati sonucu gösterilmektedir):

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

Aşağıdaki tabloda, başka veri türlerinden String veri türüne çevrim sonuçları özetlenmektedir.

Veri türü veya değeri	**Dizeye dönüştürme sonucu**
Dizi	Tüm dizi öğelerinden oluşmuş bir dize.
Boolean	`"true"` veya `"false"`
Date	Date nesnesinin dize halinde temsili.
`null`	`"null"`
Number, int veya uint	Sayının dize halinde temsili.
Nesne	Örnek null olursa `"null"`; aksi takdirde `"[object Object]"`.

Sözdizimi

Bir dilin sözdizimi, çalıştırılabilir kod yazarken izlenmesi gereken kurallar kümesini tanımlar.

Büyük/küçük harf duyarlılığı

ActionScript 3.0, büyük/küçük harf duyarlı bir dildir. Yalnızca büyük/küçük harf durumu farklı olan tanımlayıcılar, farklı tanımlayıcılar olarak değerlendirilir. Örneğin, aşağıdaki kod iki farklı değişken oluşturur:

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

Nokta sözdizimi

Nokta operatörü (.), bir nesnenin özelliklerine ve yöntemlerine erişme yolu sağlar. Nokta sözdizimini kullanıp sırayla örnek adı, nokta operatörü ve özellik veya yöntem adını kullanarak bir sınıf özelliğini ya da yöntemini ifade edebilirsiniz. Örneğin, şu sınıf tanımını göz önünde bulundurun:

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

Nokta sözdizimini kullanıp aşağıdaki kodda oluşturulan örnek adını kullanarak `prop1` özelliğine ve `method1()` yöntemine erişebilirsiniz:

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

Paketleri tanımlarken nokta sözdizimini kullanabilirsiniz. Yuvalanmış paketleri ifade etmek için nokta operatörünü kullanırsınız. Örneğin, EventDispatcher sınıfı, flash adındaki paket içinde yuvalanmış events adındaki bir pakette bulunur. Aşağıdaki ifadeyi kullanarak events paketini ifade edebilirsiniz:

```
flash.events
```

Bu ifadeyi kullanarak da EventDispatcher sınıfını ifade edebilirsiniz:

```
flash.events.EventDispatcher
```

Eğik çizgi sözdizimi

Eğik çizgi sözdizimi ActionScript 3.0'da desteklenmez. Eğik çizgi sözdizimi, bir film klibinin veya değişkenin yolunu belirtmek için önceki ActionScript sürümlerinde kullanılırdı.

Değişmez değerler

Değişmez değer, doğrudan kodunuzda görüntülenen bir değerdir. Aşağıdaki örneklerin tümü değişmezdir:

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

Değişmez değerler, bileşik değişmez değerler oluşturmak için de gruplandırılabilir. Dizi değişmezleri, köşeli ayraç karakterleri (`[]`) içine alınır ve dizi öğelerini ayırmak için virgül kullanır.

Dizi değişmezi, bir diziyi başlatmak için kullanılabilir. Aşağıdaki örnekler, dizi değişmezleri kullanılarak başlatılan iki diziyi gösterir. `new` deyimini kullanabilir ve bileşik değişmezini parametre olarak Array sınıfı yapıcısına iletebilirsiniz, ancak ayrıca, değişmez değerlerini şu ActionScript çekirdek sınıflarının örneklerini başlatırken doğrudan da atayabilirsiniz: Object, Array, String, Number, int, uint, XML, XMLList ve 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];
```

Değişmez değerler, genel bir nesneyi başlatmak için de kullanılabilir. Genel bir nesne, Object sınıfının bir örneğidir. Nesne değişmezleri küme parantezi (`{}`) içine alınır ve nesne özelliklerini ayırmak için virgül kullanır. Her özellik, özellik adını özelliğin değerinden ayıran iki nokta karakteri (`:`) ile bildirilir.

`new` deyimini kullanarak genel bir nesne oluşturabilir ve nesne değişmezini parametre olarak Object sınıfı yapıcısına iletebilir veya nesne değişmezini, bildirdiğiniz örneğe doğrudan atayabilirsiniz. Aşağıdaki örnek, yeni bir genel nesne oluşturup her biri sırayla 1, 2 ve 3 değerlerine ayarlanmış üç özellikle (`propA`, `propB` ve `propC`) nesneyi başlatmanın iki alternatif yolunu gösterir:

```
// 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};
```

Daha fazla Yardım konusu

Dizelerle çalışma

Normal ifadeler kullanma

XML değişkenlerini başlatma

Noktalı virgüller

Bir deyimi sonlandırmak için noktalı virgül karakterini (`;`) kullanabilirsiniz. Alternatif olarak, noktalı virgül karakterini çıkarırsanız, derleyici, her kod satırının tek bir deyimi temsil ettiğini varsayar. Programcıların çoğu deyim sonunu belirtmek için noktalı virgül kullanmaya alışkın olduğundan, deyimlerinizi sonlandırmak için sürekli olarak noktalı virgül kullanırsanız, kodunuzun okunması daha kolay olabilir.

Bir deyimi sonlandırmak için noktalı virgül kullanmanız, tek bir satıra birden çok deyim yerleştirmenize olanak sağlar ancak bu, kodunuzun okunmasını güçleştirebilir.

Parantezler

ActionScript 3.0'da parantezleri (`()`) üç şekilde kullanabilirsiniz. İlk olarak, bir ifadedeki işlemlerin sırasını değiştirmek için parantezleri kullanabilirsiniz. Parantezler içinde gruplandırılan işlemler her zaman önce çalıştırılır. Örneğin, aşağıdaki kodda işlemlerin sırasını değiştirmek için parantezler kullanılmıştır:

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

İkinci olarak, aşağıdaki örnekte olduğu gibi, bir ifadeler dizisini değerlendirmek ve son ifadenin sonucunu döndürmek için virgül operatörüyle (`,`) birlikte parantezleri kullanabilirsiniz:

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

Üçüncü olarak, aşağıdaki örnekte gösterildiği gibi, işlevlere veya yöntemlere bir ya da daha fazla parametre iletmek için parantezleri kullanabilirsiniz, böylece `trace()` işlevine bir String değeri iletilir:

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

Yorumlar

ActionScript 3.0 kodu, iki tür yorumu destekler: tek satırlı yorumlar ve çok satırlı yorumlar. Bu yorumlama mekanizması, C++ ve Java uygulamalarındaki yorumlama mekanizmalarına benzer. Derleyici, yorum olarak işaretlenen metni yoksayar.

Tek satırlı yorumlar, iki eğik çizgi karakteriyle (`//`) başlar ve satırın sonuna kadar devam eder. Örneğin, aşağıdaki kodda tek satırlı bir yorum bulunmaktadır:

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

Çok satırlı yorumlar, bir eğik çizgi ve yıldız işareti (`/*`) ile başlar ve bir yıldız işareti ve eğik çizgi (`*/`) ile sona erer.

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

Anahtar sözcükler ve ayrılmış sözcükler

Ayrılmış sözcükler, ActionScript tarafından kullanılmak üzere ayrılmış olduğundan, kodunuzda tanımlayıcılar olarak kullanamadığınız sözcüklerdir. Ayrılmış sözcükler arasında, derleyici tarafından programdan kaldırılmayan *sözlü anahtar sözcükler* yer alır. Tanımlayıcı olarak sözlü bir anahtar sözcük kullanırsanız, derleyici bir hata bildirir. Aşağıdaki tabloda, ActionScript 3.0 sözlü anahtar sözcükleri listelenmektedir.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	function	for
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			

Sözdizimi anahtar sözcükleri adı verilen, tanımlayıcı olarak kullanılabilen ancak belirli bağlamlarda özel anlamı olan küçük bir anahtar sözcükleri kümesi vardır. Aşağıdaki tabloda, ActionScript 3.0 sözdizimi anahtar sözcükleri listelenmektedir.

each	get	set	namespace
include	dynamic	final	native
override	static		

Ayrıca bazen *gelecekteki ayrılmış sözcükler* olarak ifade edilen birkaç tanımlayıcı da vardır. Bu tanımlayıcılardan bazıları, ActionScript 3.0 içeren yazılımlar tarafından anahtar sözcük olarak değerlendirilebilse de, bunlar ActionScript 3.0 tarafından ayrılmamıştır. Bu tanımlayıcıların çoğunu kodunuzda kullanabilirsiniz ancak bunlar sonraki dil sürümlerinde anahtar sözcük olarak görünebileceğinden, Adobe, bunları kullanmamanızı önerir.

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

Sabitler

ActionScript 3.0, sabitler oluşturmak için kullanabildiğiniz `const` deyimini destekler. Sabitler, değiştirilemeyen sabit bir değere sahip özelliklerdir. Bir sabite yalnızca bir defa değer atayabilirsiniz ve atamanın, sabitin bildirimine yakın bir yerde gerçekleşmesi gerekir. Örneğin, bir sabit bir sınıfın üyesi olarak bildirilirse, bu sabite yalnızca bildirimin parçası olarak veya sınıf yapıcısının içinde bir değer atayabilirsiniz.

Aşağıdaki kod iki sabit bildirir. Birinci sabit olan `MINIMUM`, bildirim deyiminin parçası olarak atanmış bir değere sahiptir. İkinci sabit olan `MAXIMUM`, yapıcıda atanmış bir değere sahiptir. Katı mod bir sabitin değerinin yalnızca başlatma zamanında atanmasına olanak sağladığından, bu örneğin yalnızca standart modda derleme yaptığını unutmayın.

```
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
```

Bir sabite başka bir şekilde bir başlangıç değeri atamayı denerseniz bir hata oluşur. Örneğin, sınıfın dışında `MAXIMUM` başlangıç değerini ayarlamaya çalışırsanız, bir çalışma zamanı hatası oluşur.

```
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, kullanmanız için çok çeşitli sabitleri tanımlar. Kural gereği, ActionScript'teki sabitlerin tümü, alt çizgi karakteri (_) ile ayrılmış sözcüklerde büyük harf kullanır. Örneğin, MouseEvent sınıfı tanımı, her biri fare girdisiyle ilgili bir olayı temsil eden sabitleri için bu adlandırma kuralını kullanır:

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

Operatörler

Operatörler, bir veya birkaç işleneni alıp bir değer döndüren özel işlevlerdir. *İşlenen*, bir operatörün girdi olarak kullandığı, genellikle değişmez değer, değişken veya ifade olan bir değerdir. Örneğin, aşağıdaki kodda bir değer döndürmek için, üç değişmez işlenenle (2, 3 ve 4) toplama (+) ve çarpma (*) operatörleri kullanılır. Daha sonra, döndürülen değeri (14) `sumNumber` değişkenine atamak için atama (=) operatörü tarafından bu değer kullanılır.

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

Operatörler tekli, ikili veya üçlü olabilir. *Tekli* operatör tek bir işlenen alır. Örneğin, artırma (++) operatörü tek bir işlenen aldığından tekli bir operatördür. *İkili* operatör iki işlenen alır. Örneğin, bölme (/) operatörü iki işlenen alır. *Üçlü* operatör üç işlenen alır. Örneğin, koşul (?:) operatörü üç işlenen alır.

Bazı operatörler *aşırı yüklüdür*, başka bir deyişle, kendilerine iletilen işlenenlerin türüne ve miktarına bağlı olarak farklı şekilde davranır. Toplama (+) operatörü, işlenenlerin veri türüne bağlı olarak farklı şekilde davranan bir aşırı yüklü operatör örneğidir. Her iki işlenen de sayı olursa, toplama operatörü değerlerin toplamını döndürür. Her iki işlenen de dize olursa, toplama operatörü iki işlenenin bitiştirilmiş halini döndürür. Aşağıdaki örnek kod, işlenenlere bağlı olarak operatörün nasıl farklı şekilde davrandığını gösterir:

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

Operatörler de sağlanan işlenenlerin sayısına bağlı olarak farklı şekilde davranabilir. Çıkarma (-) operatörü hem tekli hem de ikili bir operatördür. Yalnızca bir işlenen sağlandığında, çıkarma operatörü işleneni negatif duruma getirip sonucu döndürür. İki işlenen sağlandığında, çıkarma operatörü işlenenlerin farkını döndürür. Aşağıdaki örnek, ilk olarak tekli operatör olarak ve sonra da ikili operatör olarak kullanılan çıkarma operatörünü gösterir.

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

Operatör önceliği ve ilişkilendirilebilirlik

Operatör önceliği ve ilişkilendirilebilirliği, operatörlerin işleneceği sırayı belirler. Derleyicinin, toplama (+) operatöründen önce çarpma (*) operatörünü işlediği aritmetiği bilenler için bu doğal görünse de, derleyicinin ilk olarak hangi operatörlerin işleneceği hakkında açıkça talimatlara ihtiyacı vardır. Bu talimatların hepsi *operatör önceliği* olarak ifade edilir. ActionScript, parantez (()) operatörünü kullanarak değiştirebileceğiniz varsayılan bir operatör önceliğini tanımlar. Örneğin, aşağıdaki kod, derleyiciyi çarpma operatöründen önce toplama operatörünü işlemeye zorlamak için önceki örnekteki varsayılan önceliği değiştirir:

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

Aynı önceliğe sahip iki veya daha fazla operatörün aynı ifadede bulunduğu durumlarla karşılaşabilirsiniz. Bu durumlarda, derleyici, hangi operatörün önce işleneceğini belirlemek için *ilişkilendirilebilirlik* kurallarını kullanır. Atama operatörleri dışındaki tüm ikili operatörler *sola ilişkilendirilebilir*, başka bir deyişle, soldaki operatörler, sağdaki operatörlerden önce işlenir. Atama operatörleri ve koşul (?:) operatörü *sağa ilişkilendirilebilir*, başka bir deyişle, sağdaki operatörler, soldaki operatörlerden önce işlenir.

Örneğin, aynı önceliğe sahip olan küçüktür (<) ve büyüktür (>) operatörlerini göz önünde bulundurun. Aynı ifadede her iki operatör de kullanılırsa, her iki operatör de sola ilişkilendirilebilir olduğundan, soldaki operatör önce işlenir. Başka bir deyişle, aşağıdaki iki deyim aynı çıktıyı oluşturur:

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

Büyüktür operatörü önce işlenir, bu da, 3 işleneni, 2 işleneninden büyük olduğundan `true` değerini verir. Daha sonra `true` değeri, 1 işleneniyle birlikte küçüktür operatörüne iletilir. Aşağıdaki kod, bu ara durumu temsil eder:

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

Küçüktür operatörü, `true` değerini 1 sayısal değerine dönüştürür ve bu sayısal değeri ikinci işlenen olan 1 ile karşılaştırarak `false` değerini döndürür. (1 değeri 1'den küçük değildir.)

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

Parantez operatörüyle, varsayılan sola ilişkilendirilebilirliği değiştirebilirsiniz. Küçüktür operatörünü ve bu operatörün işlenenlerini parantez içine alarak, derleyiciye, ilk önce küçüktür operatörünü işlemesini bildirebilirsiniz. Aşağıdaki örnek, önceki örnekle aynı sayıları kullanarak farklı bir çıktı oluşturmak için parantez operatörünü kullanır:

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

Küçüktür operatörü önce işlenir, bu da, 2 işleneni, 1 işleneninden küçük olmadığından `false` değerini verir. Daha sonra `false` değeri, 3 işleneniyle birlikte büyüktür operatörüne iletilir. Aşağıdaki kod, bu ara durumu temsil eder:

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

Büyüktür operatörü, `false` değerini 0 sayısal değerine dönüştürür ve bu sayısal değeri diğer 3 işleneniyle karşılaştırarak `true` değerini döndürür. (3 değeri 0'dan büyüktür.)

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

Aşağıdaki tabloda, azalan öncelik sırasıyla ActionScript 3.0'ın operatörleri listelenmektedir. Her tablo satırında, aynı önceliğe sahip operatörler bulunmaktadır. Her operatör satırı, tablonun aşağısında görüntülenen satırdan daha yüksek önceliğe sahiptir.

Grup	**Operatörler**
Birincil	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Sonek	`x++ x--`
Tekli	`++x --x + - ~ ! delete typeof void`
Çarpma	`* / %`
Toplama	`+ -`
Bitsel kaydırma	`<< >> >>>`
İlişkili	`< > <= >= as in instanceof is`
Eşitlik	`== != === !==`
Bitsel AND	`&`
Bitsel XOR	`^`
Bitsel OR	`|`
Mantıksal AND	`&&`
Mantıksal OR	`||`
Koşullu	`?:`
Atama	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Virgül	`,`

Birincil operatörler

Birincil operatörler arasında, Array ve Object değişmezleri oluşturmak, ifadeleri gruplandırmak, işlevleri çağırmak, sınıf örneklerini başlatmak ve özelliklere erişmek için kullanılan operatörler yer alır.

Aşağıdaki tabloda listelendiği gibi, tüm birincil operatörler eşit önceliğe sahiptir. E4X belirtiminin parçası olan operatörler, (E4X) notasyonuyla belirtilir.

Operatör	Gerçekleştirilen işlem
`[]`	Bir diziyi başlatır
`{x:y}`	Bir nesneyi başlatır
`()`	İfadeleri gruplandırır
`f(x)`	Bir işlevi çağırır
`new`	Bir yapıcıyı çağırır
`x.y x[y]`	Bir özelliğe erişir
`<></>`	Bir XMLList nesnesini (E4X) başlatır
`@`	Bir niteliğe (E4X) erişir
`::`	Bir adı (E4X) niteler
`..`	Bir alt XML öğesine (E4X) erişir

Sonek operatörleri

Sonek operatörleri bir operatörü alır ve değeri artırır veya azaltır. Bu operatörler tekli operatörler olsa da, yüksek öncelikleri ve özel davranışları nedeniyle diğer tekli operatörlerden ayrı olarak sınıflandırılır. Sonek operatörü büyük bir ifadenin parçası olarak kullanıldığında, sonek operatörü işlenmeden önce ifadenin değeri döndürülür. Örneğin, aşağıdaki kod, değer artırılmadan önce `xNum++` ifadesinin değerinin nasıl döndürüldüğünü gösterir:

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

Aşağıdaki tabloda listelendiği gibi, tüm sonek operatörleri eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`++`	Artırır (sonek)
`--`	Azaltır (sonek)

Tekli operatörler

Tekli operatörler tek bir işlenen alır. Bu gruptaki artırma (`++`) ve azaltma (`--`) operatörleri, *önekoperatörleridir*, başka bir deyişle, bunlar bir ifadede işlenenden önce görüntülenir. Önek operatörleri, tüm ifadenin değeri döndürülmeden önce artırma veya azaltma işleminin tamamlanmasıyla sonek eşlerinden farklılık gösterir. Örneğin, aşağıdaki kod, değer artırıldıktan sonra `++xNum` ifadesinin değerinin nasıl döndürüldüğünü gösterir:

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

Aşağıdaki tabloda listelendiği gibi, tüm tekli operatörler eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`++`	Artırır (önek)
`--`	Azaltır (önek)
`+`	Tekli +
`-`	Tekli - (değilleme)
`!`	Mantıksal NOT
`~`	Bitsel NOT
`delete`	Bir özelliği siler
`typeof`	Tür bilgilerini döndürür
`void`	Tanımsız değer döndürür

Çarpma operatörleri

Çarpma operatörleri iki işlenen alır ve çarpma, bölme veya modulo hesaplamaları yapar.

Aşağıdaki tabloda listelendiği gibi, tüm çarpma operatörleri eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`*`	Çarpma
`/`	Bölme
`%`	Modulo

Toplama operatörleri

Toplama operatörleri iki işlenen alır ve toplama veya çıkarma hesaplamaları yapar: Aşağıdaki tabloda listelendiği gibi, tüm toplama operatörleri eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`+`	Toplama
`-`	Çıkarma

Bitsel kaydırma operatörleri

Bitsel kaydırma operatörleri iki işlenen alır ve birinci işlenenin bit'lerini ikinci işlenen tarafından belirtilen ölçüde kaydırır. Aşağıdaki tabloda listelendiği gibi, tüm bitsel kaydırma operatörleri eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`<<`	Bitsel sola kaydırma
`>>`	Bitsel sağa kaydırma
`>>>`	Bitsel işaretsiz sağa kaydırma

İlgili operatörler

İlgili operatörler iki işlenen alır, bunların değerlerini karşılaştırır ve bir Boolean değeri döndürür. Aşağıdaki tabloda listelendiği gibi, tüm ilgili operatörler eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`<`	Küçüktür
`>`	Büyüktür
`<=`	Küçüktür veya eşittir
`>=`	Büyüktür veya eşittir
`as`	Veri türünü kontrol eder
`in`	Nesne özelliklerini kontrol eder
`instanceof`	Prototip zincirini kontrol eder
`is`	Veri türünü kontrol eder

Eşitlik operatörleri

Eşitlik operatörleri iki işlenen alır, bunların değerlerini karşılaştırır ve bir Boolean değeri döndürür. Aşağıdaki tabloda listelendiği gibi, tüm eşitlik operatörleri eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`==`	Eşitlik
`!=`	Eşitsizlik
`===`	Katı eşitlik
`!==`	Katı eşitsizlik

Bitsel mantıksal operatörler

Bitsel mantıksal operatörler iki işlenen alır ve bit düzeyinde mantıksal işlemler gerçekleştirir. Bitsel mantıksal operatörler, öncelikleri konusunda farklılık gösterir ve azalan öncelik sırasıyla aşağıdaki tabloda listelenmektedir:

Operatör	Gerçekleştirilen işlem
`&`	Bitsel AND
`^`	Bitsel XOR
`|`	Bitsel OR

Mantıksal operatörler

Mantıksal operatörler iki işlenen alır ve bir Boolean sonucu döndürür. Mantıksal operatörler, öncelikleri konusunda farklılık gösterir ve azalan öncelik sırasıyla aşağıdaki tabloda listelenmektedir:

Operatör	Gerçekleştirilen işlem
`&&`	Mantıksal AND
`||`	Mantıksal OR

Koşul operatörü

Koşul operatörü üçlü operatördür, başka bir deyişle, üç işlenen alır. Koşul operatörü, `if..else` koşul deyimini uygulamanın kısayol yöntemidir.

Operatör	Gerçekleştirilen işlem
`?:`	Koşul

Atama operatörleri

Atama operatörleri iki işlenen alır ve bir işlenene diğer işlenenin değerini esas alarak bir değer atar. Aşağıdaki tabloda listelendiği gibi, tüm atama operatörleri eşit önceliğe sahiptir:

Operatör	Gerçekleştirilen işlem
`=`	Atama
`*=`	Çarpma ataması
`/=`	Bölme ataması
`%=`	Modülleme ataması
`+=`	Toplama ataması
`-=`	Çıkarma ataması
`<<=`	Bitsel sola kaydırma ataması
`>>=`	Bitsel sağa kaydırma ataması
`>>>=`	Bitsel işaretsiz sağa kaydırma ataması
`&=`	Bitsel AND ataması
`^=`	Bitsel XOR ataması
`|=`	Bitsel OR ataması

Koşullar

ActionScript 3.0, program akışını denetlemek için kullanabileceğiniz üç temel koşul deyimi sağlar.

if..else

`if..else` koşul deyimi, bir koşulu test etmenize ve bu koşul varsa bir kod bloğu çalıştırmanıza veya koşul yoksa alternatif bir kod bloğu çalıştırmanıza olanak sağlar. Örneğin, aşağıdaki kod, `x` değerinin 20 değerini aşıp aşmadığını test eder, aşıyorsa bir `trace()` işlevi oluşturur veya aşmıyorsa farklı bir `trace()` işlevi oluşturur:

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

Alternatif kod bloğu çalıştırmak istemiyorsanız, `if` deyimini `else` deyimi olmadan kullanabilirsiniz.

if..else if

`if..else if` koşul deyimini kullanarak, birden çok koşul için test yapabilirsiniz. Örneğin, aşağıdaki kod yalnızca `x` değerinin 20 değerini aşıp aşmadığını değil, aynı zamanda `x` değerinin negatif olup olmadığını da test eder:

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

Bir `if` veya `else` deyiminden sonra yalnızca tek bir deyim geliyorsa, deyimin küme parantezine alınması gerekmez. Örneğin, aşağıdaki kod küme parantezi kullanmaz:

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

Ancak küme parantezi bulunmayan bir koşul deyimine daha sonra if deyimleri eklenirse beklenmedik davranış oluşabileceğinden Adobe, her zaman küme parantezleri kullanmanızı önerir. Örneğin, aşağıdaki kodda koşul `true` olarak değerlendirilse de değerlendirilmese de, `positiveNums` değeri 1 artar:

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

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

trace(positiveNums); // 1
```

switch

Aynı koşul ifadesine bağlı birden çok çalıştırma yolunuz varsa `switch` deyimi kullanışlıdır. Bu, uzun `if..else if` deyimleri dizisine benzer şekilde işlevsellik sağlar, ancak daha kolay okunabilir. `switch` deyimi, bir Boolean değerinin koşulunu test etmek yerine, bir ifade olarak değerlendirilir ve hangi kod bloğunun çalıştırılacağını belirlemek için sonucu kullanır. Kod blokları bir `case` deyimiyle başlar ve bir `break` deyimiyle sona erer. Örneğin, aşağıdaki `switch` deyimi, `Date.getDay()` yönteminin döndürdüğü gün sayısını esas alarak haftanın gününü yazdırır:

```
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;
}
```

Döngü

Döngü deyimleri, bir değerler veya değişkenler dizisi kullanarak art arda belirli bir kod bloğu gerçekleştirmenize olanak sağlar. Adobe, kod bloğunu her zaman küme parantezlerinin ({}) arasına almanızı önerir. Kod bloğu yalnızca bir deyim içeriyorsa küme parantezlerini çıkarabilseniz de, koşullar için geçerli olan aynı nedenden dolayı bu uygulama önerilmez: bu, daha sonra eklenen deyimlerin yanlışlıkla kod bloğundan hariç tutulması olasılığını artırır. Daha sonra, kod bloğuna dahil etmek istediğiniz bir deyimi eklerseniz ancak gerekli küme parantezlerini koymayı unutursanız, deyim döngünün bir parçası olarak çalıştırılmaz.

function

`for` döngüsü, belirli bir değer aralığı için bir değişkeni yinelemenize olanak sağlar. `for` deyiminde üç ifade sağlamanız gerekir: başlangıç değerine ayarlı bir değişken, döngünün ne zaman sona ereceğini belirleyen bir koşul deyimi ve her döngüyle değişkenin değerini değiştiren bir ifade. Örneğin, aşağıdaki kod beş defa döngü sağlar. `i` değişkeninin değeri 0'da başlar ve 4'te sona erer, çıktı da her biri kendi satırında olan 0 ile 4 arasındaki sayılar olur.

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

for..in

`for..in` döngüsü, bir nesnenin özelliklerini veya dizideki öğeleri yineler. Örneğin, genel bir nesnenin özelliklerini yinelemek için bir `for..in` döngüsünü kullanabilirsiniz (nesne özellikleri belirli bir sırada tutulmaz, bu nedende özellikler rastgele sırada görüntüleniyor gibi gelebilir):

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

Bir dizinin öğelerini de yineleyebilirsiniz:

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

Mühürlü bir sınıfın (yerleşik sınıflar ve kullanıcı tanımlı sınıflar da dahil) örneğiyse bir nesnenin özellikleri üzerinden yineleme yapamazsınız. Yalnızca dinamik bir sınıfın özellikleri üzerinden yineleme yapabilirsiniz. Dinamik sınıfların örnekleriyle bile, ancak dinamik olarak eklenmiş özellikler üzerinden yineleme yapabilirsiniz.

for each..in

`for each..in` döngüsü, bir koleksiyonun öğelerini yineler, bu öğeler bir XML veya XMLList nesnesindeki etiketler, nesne özellikleri tarafından tutulan değerler veya bir dizinin öğeleri olabilir. Örneğin, aşağıdaki alıntıda da gösterildiği gibi, genel bir nesnenin özelliklerini yinelemek için `for each..in` döngüsünü kullanabilirsiniz ancak `for..in` döngüsünden farklı olarak, `for each..in` döngüsündeki yineleyici değişken, özelliğin adı yerine özelliğin kendisi tarafından tutulan değeri içerir:

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

Aşağıdaki örnekte gösterildiği gibi, bir XML veya XMLList nesnesini yineleyebilirsiniz:

```
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
*/
```

Bu örnekte gösterildiği gibi, bir dizinin öğelerini de yineleyebilirsiniz:

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

Bir nesne mühürlenmiş bir sınıf örneğiyse, o nesnenin özelliklerini yineleyemezsiniz. Dinamik sınıf örnekleri için de, sınıf tanımının bölümü olarak tanımlanan özellikler olan sabit özellikleri yineleyemezsiniz.

while

`while` döngüsü, koşul `true` olduğu sürece yinelenen `if` deyimine benzer. Örneğin, aşağıdaki kod, `for` döngüsü örneğiyle aynı çıktıyı oluşturur:

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

`for` döngüsü yerine `while` döngüsü kullanılmasının bir dezavantajı, sonsuz döngülerin `while` döngüleriyle daha kolay yazılmasıdır. Sayaç değişkenini artıran ifadeyi çıkarırsanız, `for` döngüsü örneği derleme yapmaz, ancak bu adımı çıkarırsanız `while` döngüsü örneği derleme yapar. `i` değerini artıran ifade olmadan döngü sonsuz döngü olur.

do..while

`do..while` döngüsü, kod bloğu çalıştırıldıktan sonra koşul denetlendiğinden, kod bloğunun en az bir defa çalıştırılmasını garantileyen bir `while` döngüsüdür. Aşağıdaki kod, koşul karşılanmasa da çıktı oluşturan basit bir `do..while` döngüsü örneğini göstermektedir:

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

İşlevler

İşlevler, belirli görevleri gerçekleştirdiğiniz ve programınızda yeniden kullanılabilen kod bloklarıdır. ActionScript 3.0'da iki tür işlev vardır: *yöntemler* ve *işlev kapanışları*. Bir işlevin yöntem veya işlev kapanışı olarak adlandırılması, işlevin tanımlandığı bağlama bağlıdır. Bir işlevi sınıf tanımının parçası olarak tanımlarsanız veya bunu bir nesne örneğine eklerseniz bu işlev yöntem olarak adlandırılır. Bir işlev başka bir şekilde tanımlanırsa, işlev kapanışı olarak adlandırılır.

İşlevler ActionScript'te her zaman çok önemli olmuştur. Örneğin, ActionScript 1.0'da `class` anahtar sözcüğü yoktu, bu nedenle "sınıflar" yapıcı işlevleri tarafından tanımlanırdı. Bu nedenle dile `class` anahtar sözcüğü eklenmiş olsa da, dilin sunması gereken şeylerden tam anlamıyla yararlanmak istiyorsanız, işlevlerin düzgün bir şekilde anlaşılması hala çok önemlidir. ActionScript işlevlerinin C++ veya Java gibi dillerdeki işlevlere benzer şekilde davranmasını bekleyen programcılar için bu güçlük yaratabilir. Temel işlev tanımı ve çağırma işlemi deneyimli programcılar için güçlük oluşturmasa da, ActionScript işlevlerinin daha gelişmiş olan bazı özelliklerinin açıklanması gerekir.

Temel işlev kavramları

İşlevleri çağırma

Bir işlevin, ardından parantez operatörü (`()`) gelen tanımlayıcısını kullanarak o işlevi çağırabilirsiniz. İşleve göndermek istediğiniz herhangi bir işlev parametresini kapsamak için parantez operatörünü kullanırsınız. Örneğin, `trace()` işlevi ActionScript 3.0'da üst düzeyli bir işlevdir:

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

Herhangi bir parametre içermeyen bir işlevi çağırıyorsanız, boş bir parantez çifti kullanmanız gerekir. Örneğin, rastgele bir sayı oluşturmak için, herhangi bir parametre almayan `Math.random()` yöntemini kullanabilirsiniz:

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

Kendi işlevlerinizi tanımlama

ActionScript 3.0'da bir işlevi tanımlamanın iki yolu vardır: bir işlev deyimini veya işlev ifadesini kullanabilirsiniz. Seçtiğiniz teknik, daha statik veya daha dinamik bir programlama stili seçmenize bağlıdır. Statik veya katı mod programlamayı tercih ediyorsanız işlevlerinizi işlev deyimleriyle tanımlayın. Aksini yapmanız gerekiyorsa, işlevlerinizi işlev ifadeleriyle tanımlayın. İşlev ifadeleri, dinamik veya standart mod programlamada daha sık kullanılır.

İşlev deyimleri

İşlev deyimleri, katı modda işlevleri tanımlamak için tercih edilen tekniktir. Bir işlev deyimi, `function` anahtar sözcüğüyle başlar ve şunlarla devam eder:

- İşlev adı
- Parantez içindeki virgül sınırlı bir listede yer alan parametreler
- İşlev gövdesi, başka bir deyişle, işlev çağrıldığında çalıştırılacak olan, küme parantezi içine alınmış ActionScript kodu

Örneğin, aşağıdaki kod, bir parametreyi tanımlayan bir işlev oluşturur ve ardından parametre değeri olarak " `hello"` dizesini kullanarak işlevi çağırır:

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

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

İşlev ifadeleri

Bir işlev bildirmenin ikinci yolu, aynı zamanda bazen bir işlev değişmezini veya adsız işlevi çağıran işlev ifadesiyle bir atama deyiminin kullanılmasıdır. Bu, önceki ActionScript sürümlerinde yaygın olarak kullanılan daha ayrıntılı bir yöntemdir.

İşlev ifadesi içeren bir atama deyimi, `var` anahtar sözcüğüyle başlar ve şunlarla devam eder:

- İşlev adı
- İki nokta operatörü (`:`)
- Veri türünü belirtecek `Function` sınıfı
- Atama operatörü (`=`)
- `function` anahtar sözcüğü
- Parantez içindeki virgül sınırlı bir listede yer alan parametreler
- İşlev gövdesi, başka bir deyişle, işlev çağrıldığında çalıştırılacak olan, küme parantezi içine alınmış ActionScript kodu

  Örneğin, aşağıdaki kod, bir işlev ifadesi kullanarak `traceParameter` işlevini bildirir:

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

  İşlev deyiminde yaptığınız gibi bir işlev adı belirtmediğinize dikkat edin. İşlev ifadeleri ile işlev deyimleri arasındaki başka bir önemli fark, işlev ifadesinin deyim yerine bir ifade olmasıdır. Başka bir deyişle, bir işlev ifadesi, işlev deyimi gibi tek başına duramaz. İşlev ifadesi yalnızca bir deyimin parçası olarak kullanılabilir ve bu genellikle bir atama deyimi olur. Aşağıdaki örnek, bir dizi örneğine atanmış işlev ifadesini gösterir:

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

Deyimler ile ifadeler arasında tercih yapma

Genel bir kural olarak, belirli koşullar bir ifade kullanımını gerektirmediği sürece, işlev deyimini kullanın. İşlev deyimleri daha az ayrıntılıdır ve katı mod ile standart mod arasında işlev ifadelerine göre daha tutarlı bir deneyim sağlar.

İşlev deyimlerinin okunması, işlev ifadelerini içeren atama deyimlerinden daha kolaydır. İşlev deyimleri kodunuzu daha kısa hale getirir; hem `var` hem de `function` anahtar sözcüklerini kullanmanızı gerektiren işlev ifadelerinden daha az karmaşıktır.

İşlev deyimleri, bir işlev deyimi kullanılarak bildirilmiş bir yöntemi çağırmak için nokta sözdizimini hem sıkı hem de standart modda kullanabilmenize olanak sağladığından, iki derleyici modu arasında daha tutarlı bir deneyim sağlar. Bu bir işlev ifadesiyle bildirilmiş yöntemler için her zaman geçerli değildir. Örneğin, aşağıdaki kod, iki yöntemle Example adında bir sınıfı tanımlar: bir işlev ifadesiyle bildirilen `methodExpression()` yöntemi ve bir işlev deyimiyle çağrılan `methodStatement()` yöntemi. Sıkı modda, `methodExpression()` yöntemini çağırmak için nokta sözdizimini kullanamazsınız.

```
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
```

İşlev deyimleri, çalışma zamanını veya dinamik davranışı esas alan programlamalar için daha uygun olarak değerlendirilir. Katı modu kullanmayı tercih ediyorsanız ancak diğer yandan bir işlev ifadesiyle bildirilmiş bir yöntemi çağırmanız gerekiyorsa, iki teknikten herhangi birini kullanabilirsiniz. İlk olarak, köşeli ayraçları (`[]`) nokta (`.`) operatörü gibi bilindik operatörlerle işlersiniz. Aşağıdaki yöntem çağrısı hem katı modda hem de standart modda başarılı olur:

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

İkinci olarak, sınıfın tamamını dinamik sınıf olarak bildirebilirsiniz. Bu, nokta operatörünü kullanarak yöntemi çağırmanıza olanak sağlasa da, bunun dezavantajı, söz konusu sınıfın tüm örnekleri için bazı katı mod işlevselliğinden taviz vermenizdir. Örneğin, bir dinamik sınıf örneğinde tanımsız bir özelliğe erişmeyi denerseniz, derleyici bir hata oluşturmaz.

İşlev ifadelerinin kullanışlı olduğu bazı koşullar vardır. İşlev ifadelerinin yaygın olarak kullanıldığı koşullardan biri, yalnızca bir defa kullanılan ve sonra atılan işlevlerdir. Daha az yaygın olarak da bir işlevin bir prototip özelliğine eklenmesi için kullanılabilir. Daha fazla bilgi için, bkz. Prototip nesnesi.

İşlev deyimleri ile işlev ifadeleri arasında, kullanılacak tekniği seçerken dikkate almanız gereken iki küçük fark vardır. Birinci fark, işlev ifadelerinin bellek yönetimi ve çöp toplamaya göre nesneler olarak bağımsız şekilde bulunmamasıdır. Başka bir deyişle, dizi öğesi veya nesne özelliği gibi başka bir nesneye bir işlev ifadesi atadığınızda, kodunuzda yalnızca o işlev ifadesine başvuru oluşturursunuz. İşlev ifadenizin eklendiği dizi veya nesne kapsam dışına çıkarsa ya da artık kullanılamazsa, işlev ifadesine daha fazla erişemezsiniz. Dizi veya nesne silinirse, işlev ifadesinin kullandığı bellek, çöp toplama için uygun olur; başka bir deyişle, bellek başka amaçlar için geri istenmeye ve yeniden kullanılmaya uygun olur.

Aşağıdaki örnek, bir işlev ifadesi için, ifadenin atandığı özellik silindikten sonra işlevin artık kullanılamadığını gösterir. Test sınıfı dinamiktir, başka bir deyişle, bir işlev ifadesi içeren `functionExp` adında bir özellik ekleyebilirsiniz. `functionExp()` işlevi nokta operatörüyle çağrılabilir, ancak `functionExp` özelliği silindikten sonra artık işleve erişilemez.

```
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
```

Diğer bir yandan, işlev ilk olarak bir işlev deyimiyle tanımlanırsa, kendi nesnesi olarak varolur ve siz işlevin eklendiği özelliği sildikten sonra da işlev varolmaya devam eder. `delete` operatörü yalnızca nesnelerin özelliklerinde çalışır, bu nedenle, `stateFunc()` işlevini silme çağrısı çalışmaz.

```
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
```

İşlev deyimleri ile işlev ifadeleri arasındaki ikinci bir fark, işlev deyimlerinin, işlev deyiminden önce görüntülenen deyimler de dahil olmak üzere, tanımlandıkları kapsamda varolmalarıdır. İşlev ifadeleri, bunun tersine yalnızca sonraki deyimler için tanımlanır. Örneğin, aşağıdaki kod tanımlanmadan önce `scopeTest()` işlevini başarıyla çağırırsa:

```
statementTest(); // statementTest

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

İşlev ifadeleri, tanımlanmadan önce kullanılamaz, bu nedenle de aşağıdaki kod bir çalışma zamanı hatası oluşturur:

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

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

İşlevlerden değerleri döndürme

İşlevinizden bir değer döndürmek için, ardından döndürmek istediğiniz ifade veya değişmez değer gelecek şekilde `return` deyimini kullanın. Örneğin, aşağıdaki kod, parametreyi temsil eden bir ifade döndürür:

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

`return` deyiminin işlevi sonlandırdığına dikkat edin, böylece aşağıdaki gibi, `return` deyiminin altındaki deyimler çalıştırılmaz:

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

Katı modda, bir döndürme türü belirtmeyi seçerseniz, ilgili türde bir değer döndürmeniz gerekir. Örneğin, aşağıdaki kod geçerli bir değer döndürmediğinden, katı modda bir hata oluşturur:

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

Yuvalanmış işlevler

İşlevleri yuvalayabilirsiniz, başka bir deyişle, işlevler diğer işlevler içinde bildirilebilir. Yuvalanmış işlevin başvurusu harici koda iletilmediği sürece, yuvalanmış bir işlev yalnızca üst işlevi içinde kullanılabilir. Örneğin, aşağıdaki kod, `getNameAndVersion()` işlevi içinde iki yuvalanmış işlev bildirir:

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

Yuvalanmış işlevler harici koda iletildiğinde, işlev kapanışları olarak iletilir; başka bir deyişle, işlev tanımlandığında kapsamda olan tüm tanımlar işlevde saklanır. Daha fazla bilgi için, bkz. İşlev kapsamı.

İşlev parametreleri

ActionScript 3.0, dil kullanımında tecrübesiz olan programcılar için yeni gibi görünen bazı işlev parametreleri işlevleri sağlar. Değere veya başvuruya göre parametre iletme kavramı çoğu programcılara tanıdık gelse de, `arguments` nesnesi ve ... (rest)parametresi birçoğunuz için yeni olabilir.

Değere veya başvuruya göre argümanları iletme

Çoğu programlama dilinde, değere veya başvuruya göre argümanları iletme arasındaki ayrımın anlaşılması önemlidir; bu ayrım kodun tasarlanma şeklini etkileyebilir.

Değere göre iletilme, argüman değerinin, işlev içinde kullanılmak üzere yerel bir değişkene kopyalanması anlamına gelir. Başvuruya göre iletilme ise gerçek değerin değil, yalnızca argümanın bir başvurusunun iletilmesi anlamına gelir. Gerçek argümanın herhangi bir kopyası oluşturulmaz. Bunun yerine, argüman olarak iletilen değişkenin başvurusu oluşturulur ve işlev içinde kullanılmak üzere yerel değişkene atanır. Yerel değişken, işlev dışındaki bir değişkenin başvurusu olarak, size orijinal değişkenin değerini değiştirme yeteneği sağlar.

ActionScript 3.0'da, tüm değerler nesneler olarak saklandığından, tüm argümanlar başvuruya göre iletilir. Ancak, Boolean, Number, int, uint ve String gibi ilkel veri türlerine ait olan nesneler, değere göre iletilmiş gibi davranmasını sağlayan özel operatörlere sahiptir. Örneğin, aşağıdaki kod, her ikisi de int türünde olan `xParam` ve `yParam` adında iki parametreyi tanımlayan `passPrimitives()` adında bir işlev oluşturur. Bu parametreler, `passPrimitives()` işlevinin gövdesinde bildirilen yerel değişkenlere benzer. İşlev, `xValue` ve `yValue` argümanlarıyla çağrılırsa, `xParam` ve `yParam` parametreleri, `xValue` ve `yValue` tarafından temsil edilen int nesnelerinin başvurularıyla başlatılır. Argümanlar ilkel olduğundan, bunlar değere göre iletilmiş gibi davranır. `xParam` ve `yParam` öğeleri başlangıçta `xValue` ve `yValue` nesnelerini içerse de, işlev gövdesi içinde değişkenler üzerinde yapılan tüm değişiklikler bellekte değerlerin yeni kopyalarını oluşturur.

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

`passPrimitives()` işlevi içinde, `xParam` ve `yParam` değerleri artırılır ancak bu, son `trace` deyiminde gösterildiği gibi, `xValue` ve `yValue` değerlerini etkilemez. İşlevin içindeki `xValue` ve `yValue` öğeleri, bellekte, işlev dışında aynı addaki değişkenlerden ayrı olarak varolan yeni konumları işaret ettiğinden, parametreler `xValue` ve `yValue` değişkenleriyle aynı şekilde adlandırılsaydı da bu durum geçerli olurdu.

Diğer tüm nesneler; başka bir deyişle, ilkel veri türünde olmayan nesneler, her zaman başvuruya göre iletilir ve bu da size orijinal değişkenin değerini değiştirme yeteneği sağlar. Örneğin, aşağıdaki kod, `x` ve `y` olmak üzere iki özellikle `objVar` adında bir nesne oluşturur. `passByRef()` işlevine argüman olarak iletilen nesne. Nesne ilkel türde olmadığından, yalnızca başvuruya göre iletilmekle kalmaz aynı zamanda başvuru olmaya devam eder. Başka bir deyişle, işlev içindeki parametreler üzerinde yapılan değişiklikler, işlev dışındaki nesne özelliklerini etkiler.

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

`objParam` parametresi, genel `objVar` değişkeniyle aynı nesneye başvurur. Örnekteki `trace` deyimlerinde de görebileceğiniz gibi, `objParam` nesnesinin `x` ve `y` özellikleri üzerinde yapılan değişiklikler, `objVar` nesnesinde yansıtılır.

Varsayılan parametre değerleri

ActionScript'te, bir işlev için *varsayılan parametre değerleri* bildirebilirsiniz. Varsayılan parametre değerleri içeren bir işleve yapılan çağrı, varsayılan değerleri içeren bir parametreyi çıkarırsa, o parametre için işlev tanımında belirtilen değer kullanılır. Varsayılan değerlere sahip tüm parametrelerin parametre listesinin sonuna yerleştirilmesi gerekir. Varsayılan değer olarak atanan değerlerin derleme zamanı sabitleri olması gerekir. Bir parametre için varsayılan bir değerin olması, o parametreyi etkili şekilde *isteğe bağlı parametre* yapar. Varsayılan değer içermeyen bir parametre, *zorunlu parametre* olarak değerlendirilir.

Örneğin, aşağıdaki kod üç parametre içeren bir işlev oluşturur, bu parametrelerin ikisi varsayılan değerleri içerir. Yalnızca bir parametreyle işlev çağrıldığında, parametrelerin varsayılan değerleri kullanılır.

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

arguments nesnesi

Bir işleve parametreler iletildiğinde, işlevinize iletilen parametreler hakkındaki bilgilere erişmek için `arguments` nesnesini kullanabilirsiniz. `arguments` nesnesinin önemli yönlerinden bazıları şunlardır:

- `arguments` nesnesi, işleve iletilen tüm parametreleri içeren bir dizidir.
- `arguments.length` özelliği, işleve iletilen parametrelerin sayısını bildirir.
- `arguments.callee` özelliği, işlevin kendisine bir başvuru sağlar, bu da işlev ifadelerine yapılan yinelemeli çağrılar için kullanışlıdır.

  Not: *Herhangi bir parametre `arguments` olarak adlandırılırsa veya ... (rest) parametresini kullanırsanız, `arguments` nesnesi kullanılamaz.*

  İşlev gövdesinde `arguments` nesnesine başvurulursa, ActionScript 3.0, işlev çağrılarının, işlev tanımında tanımlananlardan daha fazla parametre içermesine olanak tanır, ancak parametre sayısı, zorunlu parametre (ve isteğe bağlı olarak isteğe bağlı parametre) sayısıyla eşleşmezse, sıkı modda bir derleyici hatası oluşturur. İşlev tanımında tanımlansa da tanımlanmasa da, işleve iletilen herhangi bir parametreye erişmek için `arguments` nesnesinin dizi yönünü kullanabilirsiniz. Yalnızca standart modda derleme yapan aşağıdaki örnek, `traceArgArray()` işlevine iletilen tüm parametreleri izlemek için `arguments.length` özelliğiyle birlikte `arguments` dizisini kullanır:

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

traceArgArray(1, 2, 3);

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

`arguments.callee` özelliği genellikle yineleme oluşturmak için adsız işlevlerde kullanılır. Kodunuza esneklik katmak için bunu kullanabilirsiniz. Yinelemeli işlevin adı, geliştirme döngünüzde değişirse, işlev adı yerine `arguments.callee` öğesini kullanmanız durumunda, işlev gövdenizde yinelemeli çağrıyı değiştirmekle ilgili endişe duymanız gerekmez. Yinelemeyi etkinleştirmek için, aşağıdaki işlev ifadesinde `arguments.callee` özelliği kullanılır:

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

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

İşlev bildiriminizde ... (rest) parametresini kullanırsanız, `arguments` nesnesini kullanamazsınız. Bunun yerine, parametreler için bildirdiğiniz parametre adlarını kullanarak parametrelere erişmeniz gerekir.

Ayrıca parametre adı olarak `"arguments"` dizesini kullanmaktan kaçınmalısınız, aksi takdirde bu, `arguments` nesnesini gölgeler. Örneğin, bir `arguments` parametresi eklenecek şekilde `traceArgArray()` işlevi yeniden yazılırsa, işlev gövdesinde `arguments` öğesine başvurular, `arguments` nesnesini değil, parametreyi ifade eder. Aşağıdaki kod herhangi bir çıktı oluşturmaz:

```
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
```

Önceki ActionScript sürümlerinde bulunan `arguments` nesnesi de geçerli işlevi çağıran işlevin başvurusu niteliğindeki `caller` adında bir özellik içerirdi. `caller` özelliği ActionScript 3.0'da yoktur ancak çağıran işleve başvuru gerekiyorsa, çağıran işlevi, başvurunun kendisi olan fazladan bir parametreyi iletecek şekilde değiştirebilirsiniz.

... (rest) parametresi

ActionScript 3.0, ... (rest) parametresi adında yeni bir parametre içerir. Bu parametre, virgül sınırlı herhangi bir sayıda argümanı kabul eden bir dizi parametresi belirtmenize olanak sağlar. Parametreler, ayrılmış bir sözcük olmayan herhangi bir ada sahip olabilir. Bu parametre bildiriminin belirtilen son parametre olması gerekir. Bu parametrenin kullanılması, `arguments` nesnesini kullanılamaz hale getirir. ... (rest) parametresi, `arguments` dizisi ve `arguments.length` özelliğiyle aynı işlevselliği verse de, bu, `arguments.callee` tarafından sağlanan işlevselliğe benzer bir işlevsellik sağlamaz. ... (rest) parametresini kullanmadan önce `arguments.callee` öğesini kullanmadığınızdan emin olmanız gerekir.

Aşağıdaki örnek, `arguments` nesnesi yerine ... (rest) parametresini kullanarak `traceArgArray()` işlevini yeniden yazar:

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

traceArgArray(1, 2, 3);

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

... (rest) parametresi ayrıca listedeki son parametre olduğu sürece diğer parametrelerle de kullanılabilir. Aşağıdaki örnek, işlevin birinci parametresi (`x`) int türünde olacak ve ikinci parametre de ... (rest) parametresini kullanacak şekilde `traceArgArray()` işlevini değiştirir. Birinci parametre artık ... (rest) parametresi tarafından oluşturulan dizinin bölümü olmadığından, çıktı birinci değeri atlar.

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

traceArgArray(1, 2, 3);

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

Nesne olarak işlevler

ActionScript 3.0'daki işlevler nesnelerdir. Bir işlev oluşturduğunuzda, yalnızca başka bir işleve parametre olarak iletilmekle kalmayan aynı zamanda kendisine eklenmiş özellik ve yöntemlerin de bulunduğu bir nesne oluşturursunuz.

Başka bir işleve argümanlar olarak iletilen işlevler, değere göre değil, başvuruya göre iletilir. Bir işlevi argüman olarak ilettiğinizde, yöntemi çağırmak için parantez operatörünü değil yalnızca tanımlayıcıyı kullanırsınız. Örneğin, aşağıdaki kod, `addEventListener()` yöntemine argüman olarak `clickListener()` adında bir işlev iletir:

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

ActionScript'i ilk defa kullanan programcılar için bu garip görünse de, işlevler tıpkı diğer nesneler gibi özelliklere ve yöntemlere sahip olabilir. Aslında, her işlev, kendisi için tanımlı parametrelerin sayısını saklayan `length` adında salt okunur bir özelliğe sahiptir. Bu, işleve gönderilen argümanların sayısını bildiren `arguments.length` özelliğinden farklıdır. ActionScript'te, bir işleve gönderilen argüman sayısının, o işlev için tanımlanmış parametre sayısını aşabileceğini unutmayın. Katı mod, iletilen argümanların sayısı ile tanımlanan parametrelerin sayısı arasında tam eşleşme gerektirdiğinden yalnızca standart modda derleme yapan aşağıdaki örnek, iki özellik arasındaki farkı gösterir:

```
// 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 */
```

Standart modda, kendi işlev özelliklerinizi işlev gövdesinin dışında tanımlayabilirsiniz. İşlev özellikleri, işlevle ilgili bir değişkenin durumunu kaydetmenize olanak sağlayan yarı durağan özellikler görevi görebilir. Örneğin, belirli bir işlevin kaç defa çağrıldığını izlemek isteyebilirsiniz. Bir oyun yazıyorsanız ve bir kullanıcının belirli bir komutu kaç defa kullandığını izlemek istiyorsanız, statik sınıf özelliği kullanabilseniz de, bu işlevsellik kullanışlı olabilir. Katı mod, işlevlere dinamik özellikler eklemenize olanak sağlamadığından yalnızca standart modda derleme yapan aşağıdaki örnek, işlev bildirimi dışında bir işlev özelliği oluşturur ve işlev her çağrıldığında özelliği artırır:

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

someFunction.counter = 0;

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

İşlev kapsamı

Bir işlevin kapsamı, yalnızca programın neresinde işlevin çağrılabileceğini değil, işlevin hangi tanımlara erişebildiğini de belirler. Değişken tanımlayıcıları için geçerli olan aynı kapsam kuralları, işlev tanımlayıcıları için de geçerlidir. Genel kapsamda bildirilen bir işlev, tüm kodunuzda kullanılabilir. Örneğin, ActionScript 3.0, kodunuzun herhangi bir yerinde kullanılabilir olan `isNaN()` ve `parseInt()` gibi genel işlevler içerir. Yuvalanmış bir işlev—başka bir işlev içinde bildirilen bir işlev—bildirildiği işlevin herhangi bir yerinde kullanılabilir.

Kapsam zinciri

Bir işlev her çalıştırılmaya başladığında, birçok nesne ve özellik oluşturulur. İlk olarak, işlev gövdesinde bildirilen parametreleri ve yerel değişkenleri veya işlevleri saklayan *etkinleştirme nesnesi* adında özel bir nesne oluşturulur. Etkinleştirme nesnesi dahili bir mekanizma olduğundan bu nesneye doğrudan erişemezsiniz. İkinci olarak, çalışma zamanının tanımlayıcı bildirimleri için denetleyeceği nesnelerin sıralanmış bir listesini içeren bir *kapsam zinciri* oluşturulur. Çalıştırılan her işlev, dahili bir özellikte saklanan bir kapsam zincirine sahiptir. Yuvalanmış bir işlev için, kapsam zinciri kendi etkinleştirme nesnesiyle başlar ve üst işlevinin etkinleştirme nesnesiyle devam eder. Zincir, genel nesneye ulaşıncaya kadar bu şekilde devam eder. Bir ActionScript programı başlatıldığında genel bir nesne oluşturulur ve bu nesne tüm genel değişkenleri ve işlevleri içerir.

İşlev kapanışları

İşlev kapanışı, işlevin anlık görüntüsünü ve *sözlü ortamını* içeren bir nesnedir. İşlevin sözlü ortamı, işlevin değerleriyle birlikte kapsam zincirinde bulunan tüm değişkenleri, özellikleri, yöntemleri ve nesneleri içerir. Nesne veya sınıftan ayrı olarak bir işlev her çalıştırıldığında işlev kapanışları oluşturulur. İşlev kapanışlarının tanımlandıkları kapsamda bulunması, bir işlev farklı bir kapsama bir argüman veya bir döndürme değeri olarak iletildiğinde ilginç sonuçlar oluşturur.

Örneğin, aşağıdaki kod iki işlev oluşturur: bir dikdörtgenin alanını hesaplayan `rectArea()` adında yuvalanmış bir işlevi döndüren `foo()` ve `foo()` öğesini çağırıp döndürülen işlev kapanışını `myProduct` adında bir değişkende saklayan `bar().bar()` işlevi, kendi `x` yerel değişkenini (2 değeriyle) tanımlasa da, `myProduct()` işlev kapanışı çağrıldığında bu, `foo() i levinde tanımlı x de i kenini (40 de eriyle) içerir.` Bu nedenle de `bar()` işlevi, `8` değerini değil, `160` değerini döndürür.

```
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
```

Yöntemler, oluşturuldukları sözlü ortamla ilgili bilgi içermeleri yönünden benzer şekilde davranır. Bir yöntem, örneğinden ayıklanıp bağımlı bir yöntem oluşturduğunda bu özellik en belirgin durumda olur. İşlev kapanışı ile bağımlı yöntem arasındaki ana fark, bir işlev kapanışında `this`anahtar sözcüğünün değeri değişebilirken, bağımlı yöntemdeki `this` anahtar sözcüğünün değerinin her zaman başlangıçta eklendiği örneği ifade etmesidir.

Bölüm 4: ActionScript'te nesne tabanlı programlama

Nesne tabanlı programlamaya giriş

Nesne tabanlı programlama (OOP) bir programdaki kodu nesne gruplarında toplayarak organize etmenin bir yoludur. Bu bağlamda *object* terimi, bilgi (veri değerleri) ve işlevsellik içeren ayrı bir öğe anlamına gelir. Bir program organize ederken nesne tabanlı yaklaşım kullandığınızda, yaygın işlevsellik veya o bilgiyle ilişkili eylemlerle birlikte belirli bilgileri gruplarsınız. Örneğin, albüm başlığı, parça başlığı veya sanatçı adı gibi müzik bilgilerini, "parçayı çalma listesine ekle" veya "bu sanatçının tüm şarkılarını oynat/çal" gibi işlevselliklerle gruplayabilirsiniz. Bu parçalar, nesne adı verilen tek bir öğede (örneğin, bir “Album” veya “MusicTrack”) birleştirilir. Paketleme değerleri ve işlevleri birlikte bazı avantajlar sağlar. Önemli bir avantaj ise, birden fazla değişken yerine bir değişken kullanmaya ihtiyacınız olmasıdır. Ayrıca, ilgili işlevsellikleri bir arada tutar. Son olarak, bilgi ve işlevselliği birleştirmek programları gerçek hayata daha yakın bir şekilde uyacak biçimde yapılandırmanıza izin verir.

Sınıflar

Sınıf, bir nesnenin soyut temsilidir. Sınıfta, bir nesnenin barındırabileceği veri türleri ve nesnenin sergileyebileceği davranışlar hakkında bilgiler yer alır. Böyle bir soyutlamanın faydası, yalnızca birbiriyle etkileşim kuran birkaç nesnenin yer aldığı küçük komut dosyaları yazdığınızda belirgin olmayabilir. Ancak, programın kapsamı genişledikçe yönetilmesi gereken nesnelerin sayısı artar. Bu durumda sınıflar nesnelerin nasıl oluşturulduğunu ve birbirleriyle nasıl etkileşimde bulunduklarını daha rahat kontrol etmenize izin verir.

ActionScript 1.0'a kadar ActionScript programcıları, sınıflara benzeyen yapılar oluşturmak için Function nesnelerini kullanabilirdi. ActionScript 2.0 ise `class` ve `extends` gibi anahtar sözcüklerle sınıflar için biçimsel destek ekledi. ActionScript 3.0 ActionScript 2.0'daki anahtar sözcükleri desteklemekle kalmaz, aynı zamanda yeni özellikler de ekler: Örneğin, ActionScript 3.0 `protected` ve `internal` özellikleriyle geliştirilmiş erişim denetimi içerir. `final` ve `override` anahtar sözcükleriyle miras üzerinde de daha iyi denetim sağlar.

ActionScript, Java, C++, veya C# gibi programlama dillerinde sınıf oluşturmuş geliştiriciler için bilindik bir deneyim sağlar. ActionScript `class`, `extends` ve `public` gibi aynı özellik ve anahtar sözcüklerinin birçoğunu paylaşır.

Not: *Adobe ActionScript belgesinde, özellik terimiyle değişkenler, sabitler ve yöntemler gibi herhangi bir nesne veya sınıf üyesi ifade edilmektedir. Ayrıca sınıf ve statik terimleri sık sık birbirinin yerine kullanılsa da, burada bu terimler birbirinden farklı şekilde ele alınmıştır. Örneğin, burada “sınıfı özellikleri” deyimi, yalnızca statik üyeleri değil, bir sınıfın tüm üyelerini ifade etmek için kullanılmıştır.*

Sınıf tanımları

ActionScript 3.0 sınıf tanımları, ActionScript 2.0 sınıf tanımlarında kullanılana benzer bir sözdizimi kullanır. Sınıf tanımının düzgün sözdizimi, ardından sınıf adının geldiği `class` anahtar sözcüğünü çağırır. Sınıf adından sonra da, küme parantezi (`{}`) içine alınmış sınıf gövdesi gelir. Örneğin, aşağıdaki kod, `visible` adında tek bir değişken içeren Shape adında bir sınıf oluşturur:

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

Önemli sözdizimi değişikliklerinden biri, paket içindeki sınıf tanımlarında gerçekleşmiştir. ActionScript 2.0'da, bir sınıf bir paketin içindeyse, sınıf bildirimine paket adının da dahil edilmesi gerekir. `package` deyimini ilk defa sunan ActionScript 3.0'da, paket adının, sınıf bildirimine değil, paket bildirimine dahil edilmesi gerekir. Örneğin, aşağıdaki sınıf bildirimleri, flash.display paketinin bir parçası olan BitmapData sınıfının ActionScript 2.0 ve ActionScript 3.0'da nasıl tanımlandığını gösterir:

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

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

Sınıf nitelikleri

ActionScript 3.0, aşağıdaki dört nitelikten birini kullanarak sınıf tanımlarını değiştirmenize olanak sağlar:

Nitelik	Tanım
`dynamic`	Çalışma zamanında örneklere özellik eklenmesine olanak sağlar.
`final`	Başka bir sınıf tarafından genişletilmemelidir.
`internal` (varsayılan)	Geçerli paketin içindeki başvurular tarafından görülebilir.
`public`	Her yerdeki başvurular tarafından görülebilir

`internal` dışında bu niteliklerin her biri için, ilişkilendirilmiş davranışı elde etmek üzere açıkça niteliği dahil etmeniz gerekir. Örneğin, bir sınıfı tanımlarken `dynamic` niteliğini dahil etmezseniz, çalışma zamanında sınıfa özellikler ekleyemezsiniz. Aşağıdaki kodun gösterdiği gibi, sınıf tanımının başına niteliği yerleştirerek açıkça o niteliği atamış olursunuz:

```
dynamic class Shape {}
```

Listede `abstract` adında bir nitelik bulunmadığına dikkat edin. Abstract sınıfları ActionScript 3.0'da desteklenmez. Ayrıca listede `private` ve `protected` adındaki niteliklerin de bulunmadığına dikkat edin. Bu nitelikler yalnızca bir sınıf tanımı içinde anlam içerir ve tek başlarına sınıflara uygulanamaz. Bir sınıfın paket dışında herkes tarafından görülebilir olmasını istemiyorsanız, sınıfı bir paketin içine yerleştirin ve `internal` niteliğiyle işaretleyin. Alternatif olarak, hem `internal` hem de `public` niteliklerini çıkarabilirsiniz, böylece derleyici `internal` niteliğini otomatik olarak sizin için ekler. Ayrıca bir sınıfı, yalnızca tanımlandığı kaynak dosyası içinde görünür olacak şekilde tanımlayabilirsiniz. Sınıfı, paket tanımının kapatma küme parantezinin altına, kaynak dosyanızın alt tarafına yerleştirin.

Sınıf gövdesi

Sınıf gövdesi küme parantezlerinin arasında kalır. Sınıfınızın değişkenlerini, sabitlerini ve yöntemlerini tanımlar. Aşağıdaki örnek ActionScript 3.0'daki Accessibility sınıfına ilişkin bildirimi gösterir:

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

Bir sınıf gövdesinin içinde bir ad alanını da tanımlayabilirsiniz. Aşağıdaki örnek, bir ad alanının, nasıl sınıf gövdesi içinde tanımlanabildiğini ve o sınıftaki yöntemin bir niteliği olarak kullanılabildiğini gösterir:

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

ActionScript 3.0, tanımları yalnızca sınıf gövdesine değil, deyimlere de dahil etmenize olanak sağlar. Sınıf gövdesinin içinde olup yöntem tanımının dışında olan deyimler, yalnızca bir kez çalıştırılır. Bu çalıştırma, sınıf tanımıyla ilk karşılaşıldığında ve ilişkili sınıf nesnesi oluşturulduğunda gerçekleşir. Aşağıdaki örnek, `hello()` adındaki bir harici işleve yapılan çağrıyı ve sınıf tanımlandığında onaylama mesajı veren bir `trace` deyimini içerir:

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

ActionScript 3.0'da, aynı sınıf gövdesinde aynı ada sahip bir statik özelliğin ve örnek özelliğinin tanımlanmasına izin verilir. Örneğin, aşağıdaki kod, `message` adında bir statik değişkeni ve aynı ada sahip bir örnek değişkenini bildirir:

```
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
```

Sınıf özelliği nitelikleri

ActionScript nesne modeli ele alınırken, *property* terimi, değişkenler, sabitler ve yöntemler gibi, sınıf öğesi olabilen herhangi bir şeyi ifade eder. Ancak, Adobe Flash Platform'a ilişkin Adobe ActionScript 3.0 Referansı'nda bu terim daha az kullanılır. Bu bağlamda özellik terimi yalnızca değişken olan veya alıcı ya da ayarlayıcı yöntemiyle tanımlanan sınıf üyelerini içerir. ActionScript 3.0'da, bir sınıfın herhangi bir özelliğiyle kullanılan bir nitelik kümesi vardır. Aşağıdaki tabloda bu nitelik kümesi listelenmektedir.

Nitelik	Tanım
`internal` (varsayılan)	Aynı paketin içindeki başvurular tarafından görülebilir.
`private`	Aynı sınıftaki başvurular tarafından görülebilir.
`protected`	Aynı sınıftaki ve türetilmiş sınıflardaki başvurular tarafından görülebilir.
`public`	Her yerdeki başvurular tarafından görülebilir
`static`	Sınıf örneklerinin aksine, bir özelliğin bir sınıfa ait olduğunu belirtir.
`UserDefinedNamespace`	Kullanıcı tarafından tanımlanmış özel ad alanı adı

Erişim denetimi ad alanı nitelikleri

ActionScript 3.0, bir sınıf içinde tanımlanmış özelliklere erişimi denetleyen dört özel nitelik sağlar: `public`, `private`, `protected` ve `internal`.

`public` niteliği, bir özelliği komut dosyanızın her yerinde görülebilir duruma getirir. Örneğin, bir yöntemi kendi paketinin dışındaki kodlar için kullanılabilir duruma getirmek üzere, yöntemi `public` niteliğiyle bildirmeniz gerekir. `var`, `const` veya `function` anahtar sözcükleriyle bildirilmiş tüm özellikler için bu geçerlidir.

`private` niteliği, bir özelliğin yalnızca özelliğin tanımlayan sınıfı içindeki çağıranlar tarafından görülebilmesini sağlar. Bu davranış, alt sınıfın bir üst sınıftaki özel özelliğine erişmesine olanak sağlayan, ActionScript 2.0'daki `private` niteliğinin davranışından farklıdır. Davranıştaki başka bir önemli değişiklik de çalışma zamanı erişimiyle ilgilidir. ActionScript 2.0'da, `private` anahtar sözcüğü yalnızca derleme zamanında erişimi yasaklarken çalışma zamanında kolayca atlatılabilirdi. ActionScript 3.0'da ise artık bu durum geçerli değildir. `private` olarak işaretlenmiş özellikler, derleme zamanında da çalışma zamanında da kullanılamaz.

Örneğin, aşağıdaki kod, tek bir değişkenle PrivateExample adında basit bir sınıf oluşturur ve sonra sınıfın dışından özel değişkene erişmeyi dener.

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

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

ActionScript 3.0'da ise, katı mod kullanıyorsanız, nokta operatörü (`myExample.privVar`) kullanılarak bir özel özelliğe erişme girişimi, derleme zamanı hatasına yol açar. Aksi takdirde, özellik erişimi operatörü (`myExample["privVar"]`) kullandığınızda olduğu gibi, çalışma zamanında hata bildirilir.

Aşağıdaki tabloda, mühürlenmiş (dinamik olmayan) bir sınıfa ait özel özelliğe erişme girişiminin sonuçları özetlenmektedir:

	Katı mod	Standart mod
nokta operatörü (`.`)	derleme zamanı hatası	çalışma zamanı hatası
ayraç operatörü (`[]`)	çalışma zamanı hatası	çalışma zamanı hatası

`dynamic` niteliğiyle bildirilen sınıflarda, özel bir değişkene erişme girişimi, çalışma zamanı hatasına yol açmaz. Bunun yerine, değişken görünür değildir, bu yüzden `undefined` değeri geri döndürülür. Ancak katı modda nokta operatörünü kullanırsanız derleme zamanı hatası oluşur. Aşağıdaki örnek, önceki örnekle aynıdır; tek farkı, PrivateExample sınıfının bir dinamik sınıf olarak bildirilmesidir:

```
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
```

Dinamik sınıflar genellikle, bir sınıf için harici olan bir kod, özel bir özelliğe erişme girişiminde bulunduğunda hata oluşturmak yerine `undefined` değerini döndürür. Aşağıdaki tabloda, yalnızca katı modda özel bir özelliğe erişmek için nokta operatörü kullanıldığında bir hata oluşturulduğu gösterilir:

	Katı mod	**Standart mod**
nokta operatörü (`.`)	derleme zamanı hatası	`undefined`
ayraç operatörü (`[]`)	`undefined`	`undefined`

ActionScript 3.0'da ilk defa sunulan `protected` niteliği, bir özelliği kendi sınıfı veya alt sınıfı içindeki çağıranlar tarafından görülebilir duruma getirir. Başka bir deyişle, protected özelliği kendi sınıfı içinde kullanılabilir veya miras hiyerarşisinde kendisinin aşağısında bulunan sınıflar için kullanılabilir durumdadır. Alt sınıf aynı pakette veya farklı bir pakette de olsa bu durum geçerlidir.

ActionSctipt 2.0'ı bilenler için bu işlevsellik, ActionScript 2.0'daki `private` niteliğine benzerdir. ActionScript 3.0 `protected` niteliği de Java'daki `protected` niteliğine benzerdir. Java sürümünün ayrıca aynı paket içindeki çağıranlara izin vermesi açısından fark gösterir. `protected` niteliği, alt sınıfınız için gerekli olan ancak miras zincirinin dışındaki kodlardan gizlemek istediğiniz bir değişken veya yönteminiz olduğunda kullanışlıdır.

ActionScript 3.0'da yeni olan `internal` niteliği, bir özelliğin kendi paketi içindeki çağıranlar tarafından görülebilir olmasını sağlar. Bu, bir paket içindeki kod için varsayılan nitelik olup aşağıdaki niteliklerden herhangi birine sahip olmayan tüm özellikler için geçerlidir:

- `public`
- `private`
- `protected`
- kullanıcı tanımlı bir ad alanı

Java'da bu erişim düzeyi için açıkça bir ad olmayıp yalnızca başka bir erişim değiştiricisinin çıkarılmasıyla bu erişim düzeyi elde edilebilse de, `internal` niteliği, Java'daki varsayılan erişim denetimine benzer. `internal` niteliği, özelliği yalnızca kendi paketi içindeki çağıranlar tarafından görülebilir duruma getirme amacınızı açıkça belirtme seçeneğini size sunar.

static niteliği

`var`, `const` veya `function` anahtar sözcükleriyle bildirilmiş özelliklerle kullanılabilen `static` niteliği, bir özelliği sınıfın örneklerine değil, sınıfa eklemenize olanak sağlar. Sınıf için harici olan kodun, örnek adı yerine sınıf adını kullanarak statik özellikleri çağırması gerekir.

Statik özellikler alt sınıflar tarafından miras alınmaz ancak özellikler, alt sınıfın kapsam zincirinin parçasıdır. Başka bir deyişle, alt sınıfın gövdesi içinde, statik bir değişken veya yöntem tanımlanmış olduğu sınıfa başvurulmadan kullanılabilir.

Kullanıcı tanımlı ad alanı nitelikleri

Önceden tanımlanmış erişim denetimi niteliklerine alternatif olarak, bir nitelik olarak kullanılmak üzere özel bir ad alanı oluşturabilirsiniz. Her tanım için yalnızca bir ad alanı niteliği kullanılabilir ve ad alanı niteliğini, erişim denetimi niteliklerinden herhangi biriyle (`public`, `private`, `protected`, `internal`) birlikte kullanamazsınız.

Değişkenler

Değişkenler, `var` veya `const` anahtar sözcükleriyle bildirilebilir. `var` anahtar sözcüğüyle bildirilmiş değişkenlerin değerleri, komut dosyasının çalıştırılması sırasında birden çok defa değişebilir `const` anahtar sözcükleriyle bildirilen değişkenler *sabitler* olarak adlandırılır ve kendilerine yalnızca bir defa atanmış değerlere sahip olabilir. Başlatılmış bir sabite yeni bir değer atama girişimi hataya yol açar.

Statik değişkenler

Statik değişkenler, `static` anahtar sözcüğü ile `var` veya `const` deyiminin birleşimi kullanılarak bildirilir. Bir sınıf örneğine değil, sınıfa eklenen statik değişkenler, nesne sınıfının tamamı için geçerli olan bilgilerin saklanıp paylaşılmasında kullanışlıdır. Örneğin, bir sınıfın başlatılma sayısının hesabını tutmak istiyorsanız veya izin verilen maksimum sınıf örneği sayısını saklamak istiyorsanız statik değişken uygundur.

Aşağıdaki örnek, sınıf başlatma sayısının izlenmesi için `totalCount` değişkenini ve maksimum başlatma sayısını saklamak için `MAX_NUM` sabitini oluşturur. `totalCount` ve `MAX_NUM` değişkenleri, belirli bir örneğe değil, bir bütün olarak sınıfa uygulanan değerleri içerdiğinden statiktir.

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

StaticVars sınıfına ve bu sınıfın alt sınıflarına harici olan kod, yalnızca sınıfın kendisi üzerinden `totalCount` ve `MAX_NUM` özelliklerine başvurabilir. Örneğin, aşağıdaki kod çalışır:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

Sınıfın bir örneği üzerinden statik değişkenlere erişemezsiniz, bu nedenle de aşağıdaki kod hata döndürür:

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

StaticVars sınıfının `MAX_NUM` için yaptığı gibi, hem `static` hem de `const` anahtar sözcükleriyle bildirilen değişkenlerin, siz sabiti bildirdiğiniz anda başlatılması gerekir. Yapıcı veya bir örnek yöntemi içinde `MAX_NUM` öğesine bir değer atayamazsınız. Aşağıdaki kod, statik sabit başlatmanın geçerli bir yolu olmadığından bir hata oluşturur:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Örnek değişkenleri

Örnek değişkenleri arasında, `var` ve `const` anahtar sözcükleri ile ancak `static` anahtar sözcüğü olmadan bildirilen özellikler yer alır. Sınıfın tamamı yerine sınıf örneklerine eklenen örnek değişkenler, bir örneğe özgü değerlerin saklanmasında kullanışlıdır. Örneğin, Array sınıfı, belirli bir Array sınıfının barındırdığı dizi öğelerinin sayısını saklayan `length` adında bir örnek özelliğine sahiptir.

`var` veya `const` olarak bildirilen örnek değişkenleri, bir alt sınıfta geçersiz kılınamaz. Ancak, alıcı ve ayarlayıcı yöntemlerini geçersiz kılarak değişkenlerin geçersiz kılınmasına benzer işlevselliği gerçekleştirebilirsiniz.

Yöntemler

Yöntemler, bir sınıf tanımının parçası olan işlevlerdir. Sınıfın bir örneği oluşturulduktan sonra, bir yöntem bu örneğe bağımlı olur. Sınıf dışında bildirilen bir işlevden farklı olarak yöntem, eklendiği örnekten ayrı şekilde kullanılamaz.

Yöntemler, `function` anahtar sözcüğü kullanılarak tanımlanır. Herhangi bir sınıf özelliğiyle, özel, korunan, genel, dahili, statik veya özel ad alanını da içeren sınıf özelliği niteliklerinden herhangi birini yöntemlere uygulayabilirsiniz. Bir işlev deneyimini aşağıdaki gibi kullanabilirsiniz:

```
public function sampleFunction():String {}
```

Veya aşağıdaki gibi bir işlev ifadesi atadığınız bir değişkeni kullanabilirsiniz:

```
public var sampleFunction:Function = function () {}
```

Çoğu durumda, aşağıdaki nedenlerden dolayı işlev ifadesi yerine işlev deyimi kullanın:

- İşlev deyimleri daha kısa ve okunması daha kolaydır.
- İşlev deyimleri, `override` ve `final` anahtar sözcüklerini kullanmanıza olanak sağlar.
- İşlev deyimleri, tanımlayıcı (işlevin adı) ile yöntem gövdesinin içindeki kod arasında daha güçlü bir bağ oluşturur. Değişkenin değeri bir atama deyimiyle değiştirilebildiğinden, değişken ile işlev ifadesi arasındaki bağlantı herhangi bir zamanda kesilebilir. Değişkeni `var` yerine `const` ile bildirerek bu sorunu geçici olarak çözebilseniz de, bu teknik, en iyi uygulama olarak değerlendirilmez. Kodun okunmasını güçleştirip `override` ve `final` anahtar sözcüklerinin kullanılmasını önler.

Prototip nesnesine bir işlev eklemeyi seçtiğinizde işlev ifadesi kullanmanız gerekir.

Yapıcı yöntemleri

Bazen *yapıcılar* olarak da adlandırılan yapıcı yöntemleri, tanımlandıkları sınıfla aynı adı paylaşan işlevlerdir. `new` anahtar sözcüğüyle sınıfın bir örneği her oluşturulduğunda, yapıcı yöntemine dahil ettiğiniz kodlar çalıştırılır. Örneğin, aşağıdaki kod, `status` adında tek bir özellik içeren Example adında basit bir sınıfı tanımlar. `status` değişkeninin başlangıç değeri, yapıcı işlevinin içinde ayarlanır.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Yapıcı yöntemleri yalnızca genel olabilir ancak `public` niteliğinin kullanılması isteğe bağlıdır Bir yapıcıda `private`, `protected` veya `internal` gibi diğer erişim denetimi belirticilerinden herhangi birini kullanamazsınız. Ayrıca yapıcı yöntemiyle kullanıcı tanımlı bir ad alanı da kullanamazsınız.

Yapıcı, `super()` deyimini kullanarak doğrudan üst sınıfının yapıcısına açıkça bir çağrı yapabilir. Üst sınıf yapıcısı açıkça çağrılmazsa, derleyici otomatik olarak yapıcı gövdesindeki birinci deyimin önüne bir çağrı ekler. Üst sınıfın başvurusu olarak `super` önekini kullanarak da üst sınıf yöntemlerini çağırabilirsiniz. Aynı yapıcı gövdesinde hem `super()` hem de `super` öğesini kullanmaya karar verirseniz, ilk olarak `super()` öğesini çağırdığınızdan emin olun. Aksi takdirde, `super` başvurusu beklendiği gibi davranmaz. `super()` yapıcısının ayrıca `throw` veya `return` deyiminden önce çağrılması gerekir.

Aşağıdaki örnek, `super()` yapıcısını çağırmadan önce `super` başvurusunu kullanmaya çalışmanız durumunda ne olacağını gösterir. Yeni bir sınıf olan ExampleEx, Example sınıfını genişletir. ExampleEx yapıcısı, üst sınıfında tanımlanmış durum değişkenine erişmeye çalışır ancak bunu `super()` öğesini çağırmadan önce yapar. `super()` yapıcısı çalıştırılıncaya kadar `status` değişkeni kullanılamadığından, ExampleEx yapıcısının içindeki `trace()` deyimi, `null` değerini üretir.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Yapıcı içinde `return` deyiminin kullanılması geçerli bir durum olsa da, bir değer döndürülmesine izin verilmez. Başka bir deyişle, `return` deyimlerinin ilişkilendirilmiş ifadeler veya değerler içermemesi gerekir. Aynı şekilde, yapıcı yöntemlerinin değer döndürmesine izin verilmez, başka bir deyişle, herhangi bir döndürme türü belirtilemez.

Sınıfınızda bir yapıcı yöntemi tanımlamazsanız, derleyici otomatik olarak sizin için boş bir yapıcı oluşturur. Sınıfınız başka bir sınıfı genişletirse, derleyici oluşturduğu yapıcıya bir `super()` çağrısı ekler.

Statik yöntemler

Sınıf yöntemleri olarak da adlandırılan statik yöntemler, `static` anahtar sözcüğüyle bildirilen yöntemlerdir. Sınıfın örneğine değil, sınıfa eklenen statik yöntemler, tek bir örneğin durumu dışındaki şeyleri etkileyen işlevlerin kapsüllenmesinde kullanışlıdır. Statik yöntemler bir bütün olarak sınıfa eklendiğinden, statik yöntemlere sınıfın örneği üzerinden değil yalnızca sınıf üzerinden erişilebilir.

Statik yöntemler, sınıf örneklerinin durumunu etkilemekle sınırlı olmayan işlevlerin kapsüllenmesinde kullanışlıdır. Başka bir deyişle, bir yöntem sınıf örneğinin değerini doğrudan etkilemeyen işlevler sağlıyorsa statik olmalıdır. Örneğin, Date sınıfı, bir dizeyi alıp sayıya dönüştüren `parse()` adındaki bir statik yönteme sahiptir. Bu yöntem sınıfın tek bir örneğini etkilemediğinden statiktir. `parse()` yöntemi bir tarih değerini temsil eden dizeyi alır, dizeyi ayrıştırır ve Date nesnesinin dahili temsiliyle uyumlu bir biçimde sayıyı döndürür. Date sınıfının bir örneğine yöntemin uygulanması mantıklı olmadığından, bu yöntem bir örnek yöntemi değildir.

Statik `parse()` yöntemini, `getMonth()` gibi Date sınıfının örnek yöntemlerinden biriyle karşılaştırın. `getMonth()` yöntemi, Date örneğinin belirli bir bileşenini (ay) alarak doğrudan örneğin değeri üzerinde çalıştığından, bir örnek yöntemidir.

Statik yöntemler tek tek örneklere bağımlı olmadığından, statik yöntemin gövdesinde `this` veya `super` anahtar sözcüklerini kullanamazsınız. `this` başvurusu ve `super` başvurusu yalnızca bir örnek yönteminin bağlamında anlam içerir.

Bazı sınıf tabanlı programlama dillerinin aksine, ActionScript 3.0'da statik yöntemler miras alınmaz.

Örnek yöntemleri

Örnek yöntemleri, `static` anahtar sözcüğü olmadan bildirilen yöntemlerdir. Bütün olarak sınıfa değil, sınıfın örneklerine eklenen örnek yöntemleri, sınıfın tek tek örneklerini etkileyen işlevlerin uygulanmasında kullanışlıdır. Örneğin, Array sınıfı, doğrudan Array örneklerinde çalışan `sort()` adında bir örnek yöntemi içerir.

Örnek yönteminin gövdesinde statik değişkenler ve örnek değişkenleri kapsam içinde bulunur, başka bir deyişle, aynı sınıfta tanımlanmış değişkenlere, basit bir tanımlayıcı kullanılarak başvurulabilir. Örneğin, şu sınıf (CustomArray) Array sınıfını genişletir. CustomArray sınıfı, sınıf örneklerinin toplam sayısını izlemek için `arrayCountTotal` adındaki bir statik değişkeni, örneklerin oluşturulma sırasını izleyen `arrayNumber` adındaki bir örnek değişkenini ve bu değişkenlerin değerlerini döndüren `getPosition()` adındaki bir örnek yöntemini tanımlar.

```
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);
    }
}
```

Sınıfa harici olan kodun `CustomArray.arrayCountTotal` öğesini kullanarak sınıf nesnesi üzerinden `arrayCountTotal` statik değişkenine erişmesi gerekse de, `getPosition()` yönteminin gövdesinde bulunan kod, doğrudan `arrayCountTotal` değişkenini ifade edebilir. Üst sınıflardaki statik değişkenler için de bu geçerlidir. Statik özellikler ActionScript 3.0'da miras alınmasa da, üst sınıflardaki statik özellikler kapsam içindedir. Örneğin, Array sınıfı, bir tanesi `DESCENDING` olarak adlandırılan birkaç statik değişkene sahiptir. Array alt sınıfında bulunan kod, basit bir tanımlayıcı kullanarak `DESCENDING` statik sabitine erişebilir:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

Bir örnek yönteminin gövdesindeki `this` başvurusunun değeri, yöntemin eklendiği örneğin başvurusudur. Aşağıdaki kod, `this` başvurusunun yöntemi içeren örneği işaret ettiğini gösterir:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Örnek yöntemlerinin mirası, `override` ve `final` anahtar sözcükleriyle denetlenebilir. Miras alınan bir yöntemi yeniden tanımlamak için `override` niteliğini ve alt sınıfların bir yöntemi geçersiz kılmasını önlemek için `final` niteliğini kullanabilirsiniz.

Erişimci yöntemlerini alma ve ayarlama

Alıcılar ve *ayarlayıcılar* olarak da adlandırılan alma ve ayarlama erişimci işlevleri, oluşturduğunuz sınıflar için kullanımı kolay bir programlama arabirimi sağlarken, bilgi gizleme ve kapsüllemeye yönelik programlama ilkelerine da bağlı kalmanıza olanak sağlar. Alma ve ayarlama işlevleri, sınıf özelliklerinizin sınıf için özel olmasını sürdürmenizi ancak sınıfınızın kullanıcılarının bir sınıf yöntemi çağırmak yerine bir sınıf değişkenine erişiyormuş gibi bu özelliklere erişmesine olanak sağlar.

Bu yaklaşımın avantajı, `getPropertyName()` ve `setPropertyName()` gibi kullanılması güç adlara sahip geleneksel erişimci işlevlerinden kaçınmanıza olanak sağlamasıdır. Alıcı ve ayarlayıcıların başka bir avantajı da, hem okuma hem de yazma erişimine izin veren her özellik için genele açık iki işlevden kaçınabilmenizi sağlamasıdır.

GetSet adındaki şu örnek sınıf, `privateProperty` adındaki özel değişkene erişim sağlayan `publicAccess()` adında alma ve ayarlama erişimci işlevlerini içerir:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

`privateProperty` özelliğine doğrudan erişmeyi denerseniz, aşağıdaki gibi bir hata oluşur:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Bunun yerine, GetSet sınıfının kullanıcısı, `publicAccess` adında bir özellik olarak görünen ancak gerçekte `privateProperty` adındaki özel özellikte çalışan bir alma ve ayarlama erişimcisi işlevleri çifti olan bir öğeyi kullanır. Aşağıdaki örnek, GetSet sınıfını başlatır ve sonra `publicAccess` adındaki genel erişimciyi kullanarak `privateProperty` değerini ayarlar:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Alıcı ve ayarlayıcı işlevleri, normal sınıf üyesi değişkenleri kullandığınız zaman mümkün olmayacak şekilde, bir üst sınıftan miras alınan özellikleri geçersiz kılmanıza olanak sağlar. `var` anahtar sözcüğü kullanılarak bildirilen sınıf üyesi değişkenleri, bir alt sınıfta geçersiz kılınamaz. Alıcı ve ayarlayıcı işlevleri kullanılarak oluşturulan özelliklerse bu kısıtlamaya sahip değildir. Bir üst sınıftan miras alınan alıcı ve ayarlayıcı işlevlerinde `override` niteliğini kullanabilirsiniz.

Bağımlı yöntemler

Bazen *yöntem kapanışı* olarak da adlandırılan bir bağımlı yöntem, yalnızca örneğinden ayıklanmış bir yöntemdir. Bağımlı yöntem örnekleri arasında, bir işleve argüman olarak iletilen veya bir işlevden değer olarak döndürülen yöntemler yer alır. ActionScript 3.0'da yeni bir özellik olan bağımlı yöntem, kendi örneğinden ayıklandığında da sözlü ortamını koruduğundan bir işlev kapanışına benzer. Bağımlı yöntem ile işlev kapanışı arasındaki en önemli fark, bağımlı yöntemin `this` başvurusunun, yöntemi uygulayan örneğe bağlı veya bağımlı kalmaya devam etmesidir. Başka bir deyişle, bağımlı yöntemdeki `this` başvurusu her zaman yöntemi uygulayan orijinal nesneyi işaret eder. İşlev kapanışları için, `this` başvurusu geneldir, başka bir deyişle, çağrıldığı zaman işlevin ilişkilendirilmiş olduğu nesneyi işaret eder.

`this` anahtar sözcüğünü kullanıyorsanız, bağımlı yöntemleri anlamanız önemlidir. `this` anahtar sözcüğünün yöntemin üst nesnesine bir başvuru sağladığını unutmayın. ActionScript programcılarının çoğu, `this` anahtar sözcüğünün her zaman yöntemin tanımını içeren nesneyi veya sınıfı temsil etmesini bekler. Ancak yöntem bağlama olmadan bu her zaman geçerli olmayabilir. Örneğin, önceki ActionScript sürümlerinde `this` başvurusu her zaman yöntemi uygulayan örneği ifade etmiyordu. ActionScript 2.0'da bir örnekten yöntemler ayıklandığında, `this` başvurusu orijinal örneğe bağımlı olmamakla kalmaz aynı zamanda örneğin sınıfının üye değişkenleri ve yöntemleri de kullanılamaz. Bir yöntemi parametre olarak ilettiğinizde bağımlı yöntemler otomatik olarak oluşturulduğundan ActionScript 3.0'da bu bir sorun yaratmaz. Bağımlı yöntemler, `this` anahtar sözcüğünün her zaman yöntemin tanımlanmış olduğu nesne veya sınıfa başvurmasını sağlar.

Aşağıdaki kod, bağımlı yöntemi tanımlayan `foo()` adındaki bir yöntemi ve bağımlı yöntemi döndüren `bar()` adındaki bir yöntemi içeren ThisTest adındaki bir sınıfı tanımlar. Sınıfa harici olan kod, ThisTest sınıfının bir örneğini oluşturur, `bar()` yöntemini çağırır ve döndürme değerini `myFunc` adındaki bir değişkende saklar.

```
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 */
```

Kodun son iki satırı, kendisinden bir önceki satırsa bulunan `this` başvurusu genel nesneyi işaret etse de, `foo()` bağımlı yöntemindeki `this` başvurusunun halen ThisTest sınıfının bir örneğini işaret ettiğini gösterir. Üstelik, `myFunc` değişkeninde saklanan bağımlı yöntem, ThisTest sınıfının üye değişkenlerine erişmeye devam eder. Bu kodun aynısı ActionScript 2.0'da çalıştırılsa, `this` başvuruları eşleşir ve `num` değişkeni `undefined` olurdu.

`addEventListener()` yöntemi, argüman olarak bir işlev veya yöntem iletmenizi gerektirdiğinden, bağımlı yöntemlerin eklenmesinin en belirgin olduğu alanlardan biri olay işleyicileridir.

Sınıflarla numaralandırma

Numaralandırmalar, küçük bir değer kümesini kapsüllemek için oluşturduğunuz özel veri türleridir. ActionScript 3.0, `enum` anahtar sözcüğüne sahip C++ veya Numaralandırma arabirimine sahip Java uygulamalarından farklı olarak belirli bir numaralandırma hizmetini desteklemez. Ancak sınıfları ve statik sabitleri kullanarak numaralandırmalar oluşturabilirsiniz. Örneğin, ActionScript 3.0'daki PrintJob sınıfı, aşağıdaki kodda gösterildiği gibi, `"landscape"` ve `"portrait"` değerlerini saklamak için PrintJobOrientation adında bir numaralandırma kullanır:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Kural gereği, sınıfı genişletmeye gerek olmadığından, numaralandırma sınıfı `final` niteliğiyle bildirilir. Sınıf yalnızca statik üyeleri içerir, başka bir deyişle, sınıfın örneklerini oluşturmazsınız. Bunun yerine, aşağıdaki kod alıntısında gösterildiği gibi, doğrudan sınıf nesnesi üzerinden numaralandırma değerlerine erişirsiniz:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

ActionScript 3.0'daki numaralandırma sınıflarının tümü yalnızca String, int veya uint türündeki değişkenleri içerir. Değişmez dizeler veya sayı değerleri yerine numaralandırmaları kullanmanın avantajı, yazım hatalarının numaralandırmada daha kolay bulunabilmesidir. Bir numaralandırmanın adını yanlış yazarsanız, ActionScript derleyicisi bir hata oluşturur. Değişmez değerleri kullanırsanız, bir sözcüğü yanlış yazdığınızda veya yanlış sayıyı kullandığınızda derleyici şikayette bulunmaz. Önceki örnekte, aşağıdaki alıntının gösterdiği gibi, numaralandırma sabitinin adı hatalıysa, derleyici bir hata oluşturur:

```
if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Ancak, aşağıdaki gibi, bir dize değişmez değerini yanlış yazarsanız, derleyici bir hata oluşturmaz:

```
if (pj.orientation == "portrai") // no compiler error
```

Numaralandırma oluşturmaya yönelik ikinci bir teknik de, numaralandırma için statik özelliklere sahip ayrı bir sınıf oluşturulmasıdır. Ancak her statik özellik, bir dize veya tam sayı değerini değil, sınıfın bir örneğini içerdiğinden, bu teknik farklılık gösterir. Örneğin, aşağıdaki kod, haftanın günleri için bir numaralandırma sınıfı oluşturur:

```
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();
}
```

Bu teknik ActionScript 3.0 tarafından kullanılmasa da, tekniğin sağladığı gelişmiş tür denetlemesini tercih eden birçok geliştirici tarafından kullanılır. Örneğin, numaralandırma değeri döndüren bir yöntem, döndürme değerini numaralandırma veri türüyle sınırlandırabilir. Aşağıdaki kod, yalnızca haftanın gününü döndüren bir işlevi değil, aynı zamanda tür ek açıklaması olarak numaralandırma türünü kullanan bir işlev çağrısını da gösterir:

```
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();
```

Ayrıca, haftanın günlerinin her biriyle bir tam sayıyı ilişkilendirip günün dize halinde temsilini döndüren bir `toString()` yöntemi sağlayacak şekilde Day sınıfını geliştirebilirsiniz.

Gömülü varlık sınıfları

ActionScript 3.0, gömülü varlıkları temsil etmek için *gömülü varlık sınıfları* adı verilen özel sınıfları kullanır. *Gömülü varlık*, derleme zamanında bir SWF dosyasına dahil edilen, ses, görüntü, font gibi bir varlıktır. Bir varlığın dinamik olarak yüklenmek yerine gömülmesi, o varlığın çalışma zamanında da kullanılabilir olmasını sağlarken SWF dosyasının boyutunun artmasına neden olur.

Flash Professional'da gömülü varlık sınıflarını kullanma

Bir varlığı gömmek için, ilk olarak varlığı bir FLA dosyasının kütüphanesine yerleştirin. Ardından varlığın gömülü varlık sınıfı için bir ad sağlamak üzere varlığın bağlantı özelliğini kullanın. Sınıf yolunda bu ada sahip bir sınıf bulunmazsa, sizin için otomatik olarak bir sınıf oluşturulur. Daha sonra, gömülü varlık sınıfının bir örneğini oluşturabilir ve o sınıf tarafından tanımlanmış veya miras alınmış özellikleri ve yöntemleri kullanabilirsiniz. Örneğin, aşağıdaki kod, PianoMusic adındaki gömülü bir varlık sınıfına bağlı gömülü bir sesi çalmak için kullanılabilir:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Alternatif olarak, `[Embed]` meta veri etiketini bir sonraki bölümde açıklandığı gibi, Flash Professional projesinde varlık gömmek için kullanabilirsiniz. `[Embed]` meta veri etiketini kodunuzda kullanırsanız, Flash Professional, projenizi derlemek için Flash Professional derleyicisi yerine Flex derleyicisi kullanır.

Flex derleyicisi kullanarak gömülü varlık sınıfları kullanma

Kodunuzu Flex derleyicisi kullanarak derliyorsanız, ActionScript kodunda bir varlık gömmek için `[Embed]` meta veri etiketini kullanın. Varlığı ana kaynak klasörüne veya projenizin oluşturma yolundaki başka bir klasöre yerleştirin. Flex derleyicisi bir Embed meta veri etiketiyle karşılaşırsa, sizin için gömülü varlık sınıfını oluşturur. `[Embed]` meta veri etiketinden hemen sonra bildirdiğiniz bir Class veri türündeki değişken üzerinden sınıfa erişebilirsiniz. Örneğin, aşağıdaki kod, sound1.mp3 adındaki bir sesi gömer ve o sesle ilişkilendirilmiş gömülü varlık sınıfının bir başvurusunu saklamak için `soundCls` adındaki bir değişkeni kullanır. Daha sonra örnek, gömülü varlık sınıfının bir örneğini oluşturur ve bu örnekte `play()` yöntemini çağırır:

```
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

Bir Flash Builder ActionScript projesinde `[Embed]` meta veri etiketini kullanmak için, Flex çerçevesinden gerekli sınıfları içe aktarın. Örneğin, sesleri gömmek için mx.core.SoundAsset sınıfını içe aktarın. Flex çerçevesini kullanmak için, ActionScript oluşturma yolunuza framework.swc dosyasını dahil edin. Bunun sonucunda SWF dosyanızın boyutu artar.

Adobe Flex

Alternatif olarak, Flex'te bir MXML etiket tanımında `@Embed()` direktifiyle bir varlığı gömebilirsiniz.

Arabirimler

Arabirim, ilgisiz nesnelerin birbiriyle iletişim kurmasına olanak sağlayan bir yöntem bildirimleri koleksiyonudur. Örneğin, ActionScript 3.0, bir sınıfın olay nesnelerini işlemek için kullanabileceği yöntem bildirimlerini içeren IEventDispatcher arabirimini tanımlar. IEventDispatcher arabirimi, nesnelerin birbirine olay nesnelerini iletmesi için standart bir yol oluşturur. Aşağıdaki kod, IEventDispatcher arabiriminin tanımını gösterir:

```
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;
}
```

Arabirimler, bir yöntemin arabirimi ile uygulaması arasındaki ayrımı esas alır. Bir yöntemin arabirimi, yöntemin adı, tüm parametreleri ve döndürme türü gibi yöntemi çağırmak için gerekli olan tüm bilgileri içerir. Yöntemin uygulaması, yalnızca arabirim bilgilerini değil, yöntemin davranışını yürüten çalıştırılabilir deyimleri de içerir. Arabirim tanımı yalnızca yöntem arabirimlerini içerir ve arabirimi uygulayan tüm sınıflar, yöntem uygulamalarını tanımlamaktan sorumludur.

ActionScript 3.0'da, EventDispatcher sınıfı, tüm IEventDispatcher arabirim yöntemlerini tanımlayıp yöntemlerin her birine yöntem gövdeleri ekleyerek IEventDispatcher arabirimini uygular. Aşağıdaki kod, EventDispatcher sınıfı tanımından bir alıntıdır:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

IEventDispatcher arabirimi, olay nesnelerini işlemek ve IEventDispatcher arabirimini uygulayan diğer nesnelere de bu olay nesnelerini iletmek için EventDispatcher örneklerinin kullandığı bir protokol görevi görür.

Arabirimin başka bir açıklaması da, veri türünü tıpkı sınıfın tanımladığı gibi tanımlamasıdır. Aynı şekilde, tıpkı sınıf gibi, arabirim de tür ek açıklaması olarak kullanılabilir. Bir veri türü olarak arabirim, veri türü gerektiren `is` ve `as` operatörleri gibi operatörlerle de kullanılabilir. Ancak sınıfın aksine, bir arabirim başlatılamaz. Bu ayrım da birçok programcının arabirimleri soyut veri türleri olarak ve sınıfları da somut veri türleri olarak değerlendirmesine neden olmuştur.

Arabirimi tanımlama

Bir arabirim tanımının yapısı, sınıf tanımının yapısına benzer; tek fark, arabirimin yalnızca herhangi bir yöntem gövdesi içermeyen yöntemleri içerebilmesidir. Arabirimler, değişken veya sabitleri içeremez ancak alıcı ve ayarlayıcıları içerebilir. Bir arabirimi tanımlamak için, `interface` anahtar sözcüğünü kullanın. Örneğin, şu IExternalizable arabirimi, ActionScript 3.0'da flash.utils paketinin parçasıdır. IExternalizable arabirimi, bir nesnenin serileştirilmesi, başka bir deyişle nesnenin bir aygıtta depolanmaya veya ağda taşınmaya uygun bir biçime dönüştürülmesine yönelik bir protokolü tanımlar.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

IExternalizable arabirimi `public` erişim denetimi değiştiricisi ile bildirilir. Arabirim tanımları yalnızca `public` ve `internal` erişim denetimi belirticileri tarafından değiştirilebilir. Bir arabirim tanımının içindeki yöntem bildirimleri, herhangi bir erişim denetimi belirticisi içeremez.

ActionScript 3.0, arabirim adlarının `I` büyük harfiyle başladığı bir kuralı izler ancak arabirim adı olarak herhangi bir geçerli tanımlayıcıyı kullanabilirsiniz. Arabirim tanımları genellikle paketin üst düzeyine yerleştirilir. Arabirim tanımları bir sınıf tanımının içine veya başka bir arabirim tanımının içine yerleştirilemez.

Arabirimler başka bir veya birkaç arabirimi genişletebilir. Örneğin, aşağıdaki IExample arabirimi, IExternalizable arabirimini genişletir:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

IExample arabirimini uygulayan tüm sınıfların yalnızca `extra()` yöntemi için değil, aynı zamanda IExternalizable arabiriminden miras alınan `writeExternal()` ve `readExternal()` yöntemleri için de uygulamalar içermesi gerekir.

Bir sınıfta arabirim uygulama

Sınıf, bir arabirim uygulayabilen ActionScript 3.0 dil öğesidir. Bir veya daha fazla arabirim uygulamak için sınıf bildiriminde `implements` anahtar sözcüğünü kullanın. Aşağıdaki örnek, IAlpha ve IBeta adında iki arabirimi ve bunların her ikisini uygulayan `Alpha` adında bir sınıfı tanımlar:

```
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 {}
}
```

Arabirim uygulayan bir sınıfta, uygulanan yöntemlerin şunları yapması gerekir:

- `public` erişim denetimi tanımlayıcısını kullanma.
- Arabirim yöntemiyle aynı adı kullanma.
- Veri türlerinin her biri arabirim yöntemi parametresi veri türüyle eşleşen, aynı sayıda parametreye sahip olma.
- Aynı döndürme türünü kullanma.

  ```
  public function foo(param:String):String {}
  ```

Ancak uyguladığınız yöntemlerin parametrelerini adlandırma şeklinizde esnekliğe sahip olursunuz. Uygulanan yöntemdeki parametre sayısının ve her parametrenin veri türünün, arabirim yöntemininkilerle eşleşmesi gerekse de, parametre adlarının eşleşmesi gerekmez. Örneğin, önceki örnekte, `Alpha.foo()` yönteminin parametresi `param` olarak adlandırılır:

Ancak parametre, `IAlpha.foo()` arabirim yönteminde `str` olarak adlandırılır:

```
function foo(str:String):String;
```

Ayrıca varsayılan parametre değerlerinde de esnekliğe sahip olursunuz. Bir arabirim tanımı, varsayılan parametre değerlerine sahip işlev bildirimleri içerebilir. Böyle bir işlev bildirimini uygulayan yöntemin, arabirim tanımında belirtilen değerle aynı veri türünün üyesi olan bir varsayılan parametre değerine sahip olması gerekir ancak gerçek değerin eşleşmesi gerekmez. Örneğin, aşağıdaki kod, 3 varsayılan parametre değerine sahip bir yöntemi içeren arabirimi tanımlar:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

Aşağıdaki sınıf tanımı, Igamma arabirimini uygular ancak farklı bir varsayılan parametre değeri kullanır:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Bu esnekliğin nedeni, arabirim uygulama kurallarının özellikle veri türü uyumluluğunu sağlamak üzere tasarlanmış olması ve aynı parametre adları ve varsayılan parametre değerlerinin zorunlu tutulmasının, bu hedefin elde edilmesinde gerekli olmamasıdır.

Miras

Miras, programcıların varolan sınıfları esas alarak yeni sınıflar geliştirmesine olanak sağlayan bir kod yeniden kullanım şeklidir. Varolan sınıflar genellikle *temel sınıflar* veya *üst sınıflar* olarak adlandırılırken, yeni sınıflar genellikle *alt sınıflar* olarak adlandırılır. Mirasın en büyük avantajı, siz bir temel sınıftaki kodu yeniden kullanırken varolan kodu değiştirmeden bırakmanıza olanak sağlamasıdır. Üstelik miras, diğer sınıfların temel sınıfla etkileşim kurma şekli üzerinde herhangi bir değişiklik gerektirmez. Tamamen test edilmiş veya hala kullanımda olabilecek varolan bir sınıfı değiştirmek yerine, mirası kullanarak sınıfı, ek özellik ve yöntemlerle genişletebileceğiniz tümleşik bir modül olarak değerlendirebilirsiniz. Aynı şekilde, bir sınıfın başka bir sınıftan miras aldığını belirtmek için `extends` anahtar sözcüğünü kullanırsınız.

Miras aynı zamanda kodunuzda *çok biçimlilikten* yararlanmanıza olanak sağlar. Çok biçimlilik, farklı veri türlerine uygulandığında farklı şekilde davranan bir yöntem için tek bir yöntem adı kullanma yeteneğidir. Bunun basit bir örneği, Circle ve Square adında iki alt sınıf içeren Shape adındaki bir temel sınıftır. Shape sınıfı, şeklin alanını döndüren, `area()` adında bir yöntemi tanımlar. Çok biçimlilik uygulanıyorsa, Circle ve Square türündeki nesnelerde `area()` yöntemini çağırıp sizin için doğru hesaplamaların yapılmasını sağlayabilirsiniz. Miras, alt sınıfların temel sınıflardaki yöntemleri miras almasına, yeniden tanımlamasına veya *geçersiz kılmasına* olanak sağlayarak çok biçimliliği etkinleştirir. Aşağıdaki örnekte, `area()` yöntemi Circle ve Square sınıfları tarafından yeniden tanımlanır:

```
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
```

Her sınıf bir veri türünü tanımladığından, miras kullanılması, temel sınıf ile temel sınıfı genişleten sınıf arasında özel bir ilişki oluşturur. Bir alt sınıfın, temel sınıfın tüm özelliklerine sahip olması garantilenir, başka bir deyişle, alt sınıfın bir örneği her zaman temel sınıfın bir örneği yerine geçebilir. Örneğin, bir yöntem Shape türünde bir parametreyi tanımlarsa, aşağıdaki gibi, Circle öğesi Shape öğesini genişlettiğinden Circle türünde bir argüman iletilmesi geçerli bir durumdur:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Örnek özellikleri ve miras

`function`, `var` veya `const` anahtar sözcükleriyle tanımlanmış bir örnek özelliği, özellik temel sınıfta `private` niteliğiyle bildirilmediği sürece, tüm alt sınıflar tarafından miras alınır. Örneğin, ActionScript 3.0'daki Event sınıfı, tüm olay nesneleri için ortak olan özellikleri miras alan çok sayıda alt sınıfa sahiptir.

Bazı olay türleri için, Event sınıfı, olayın tanımlanması için gerekli olan tüm özellikleri içerir. Bu olay türleri, Event sınıfında tanımlı olanlar dışında bir örnek özelliği gerektirmez. Bu olaylara örnek olarak, veri başarıyla yüklendiğinde gerçekleşen `complete` olayı ve bir ağ bağlantısı kurulduğunda gerçekleşen `connect` olayı verilebilir.

Aşağıdaki örnek, alt sınıflar tarafından miras alınan özellik ve yöntemlerden bazılarını gösteren bir Event sınıfı alıntısıdır. Özellikler miras alındığından, tüm alt sınıflar bu özelliklere erişebilir.

```
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 {}
    ...
}
```

Diğer olay türleri, Event sınıfında kullanılamayan benzersiz özellikleri gerektirir. Bu olaylar, Event sınıfının alt sınıfları kullanılarak tanımlanır, böylece Event sınıfında tanımlanan özelliklere yeni özellikler eklenebilir. Böyle bir alt sınıfa örnek olarak, `mouseMove` ve `click` olayları gibi fare hareketiyle veya fare tıklatmalarıyla ilişkilendirilmiş olaylara özgü özellikler ekleyen MouseEvent sınıfı verilebilir. Aşağıdaki örnek, alt sınıfta bulunan ancak temel sınıfta bulunmayan özelliklerin tanımını gösteren bir MouseEvent alıntısıdır:

```
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 {}
    ...
}
```

Erişim denetimi belirticileri ve miras

Bir özellik `public` anahtar sözcüğüyle bildirilirse, özellik her yerdeki kodlar tarafından görülebilir. Başka bir deyişle, `private`, `protected` ve `internal` anahtar sözcüklerinin aksine, `public` anahtar sözcüğü özellik mirasına herhangi bir kısıtlama koymaz.

Bir özellik `private` anahtar sözcüğüyle bildirilirse, yalnızca kendisini tanımlayan sınıfta görülebilir, başka bir deyişle, alt sınıflar tarafından miras alınmaz. Bu davranış, `private` anahtar sözcüğünün daha çok ActionScript 3.0 `protected` anahtar sözcüğü gibi davrandığı önceki ActionScript sürümlerinden farklıdır.

`protected` anahtar sözcüğü, bir özelliğin yalnızca kendisini tanımlayan sınıfta değil, tüm alt sınıflar için de görülebilir olduğunu belirtir. Java programlama dilindeki `protected` anahtar sözcüğünün aksine, ActionScript 3.0'daki `protected` anahtar sözcüğü, bir özelliği aynı paketteki diğer tüm sınıflar tarafından görülebilir duruma getirmez. ActionScript 3.0'da, yalnızca alt sınıflar `protected` anahtar sözcüğüyle bildirilen bir özelliğe erişebilir. Üstelik, alt sınıf temel sınıfla aynı pakette de olsa farklı pakette de olsa, korumalı özelliği o alt sınıf tarafından görülebilir.

Bir özelliğin görünebilirliğini tanımlandığı paketle sınırlandırmak için, `internal` anahtar sözcüğünü kullanın veya herhangi bir erişim denetimi belirticisi kullanmayın. `internal` erişim denetimi belirticisi, herhangi bir erişim denetimi belirticisi belirtilmediğinde geçerli olan varsayılandır. `internal` olarak işaretlenmiş bir özellik yalnızca aynı pakette bulunan bir alt sınıf tarafından miras alınır.

Erişim denetimi belirticilerinin her birinin, paket sınırları boyunca mirası nasıl etkilediğini görmek için aşağıdaki örneği kullanabilirsiniz. Aşağıdaki örnek, AccessControl adında bir ana uygulama sınıfını ve Base ve Extender adında başka iki sınıfı tanımlar. Base sınıfı, foo adındaki bir pakette ve Base sınıfının alt sınıfı olan Extender sınıfı da bar adındaki bir pakettedir. AccessControl sınıfı yalnızca Extender sınıfını içe aktarır ve Base sınıfında tanımlanmış `str` adındaki bir değişkene erişmeye çalışan Extender örneğini oluşturur. `str` değişkeni, `public` olarak bildirilir, böylece aşağıdaki alıntıda gösterildiği gibi kod derleme yapar ve çalıştırılır:

```
// 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
        }
    }
}
```

Diğer erişim denetimi belirticilerinin, önceki örnekte derlemeyi ve çalıştırmayı nasıl etkilediğini görmek için, `AccessControl` sınıfından aşağıdaki satırı sildikten veya aşağıdaki satırın yorumunu kaldırdıktan sonra, `str` değişkeninin erişim denetimi belirticisini `private`, `protected` ya da `internal` olarak değiştirin:

```
trace(myExt.str);// error if str is not public
```

Değişkenleri geçersiz kılmaya izin verilmez

`var` veya `const` anahtar sözcükleriyle bildirilen özellikler miras alınır ancak geçersiz kılınamaz. Bir özelliğin geçersiz kılınması, bir alt sınıfta özelliğin yeniden tanımlanması anlamına gelir. Geçersiz kılınabilen tek özellik türü al ve ayarla erişimcileridir (`function` anahtar sözcüğüyle bildirilmiş özellikler). Bir örnek değişkenini geçersiz kılamasanız da, örnek değişkeni için alıcı ve ayarlayıcı yöntemleri oluşturup yöntemleri geçersiz kılarak benzer bir işlevi elde edebilirsiniz.

Yöntemleri geçersiz kılma

Bir yöntemin geçersiz kılınması, miras alınan bir yöntemin davranışının yeniden tanımlanması anlamına gelir. Statik yöntemler miras alınmaz ve geçersiz kılınamaz. Ancak, örnek yöntemleri, alt sınıflar tarafından miras alınır ve şu iki kriter karşılandığı sürece geçersiz kılınabilir:

- Örnek yöntemi temel sınıfta `final` anahtar sözcüğüyle bildirilmez. `final` anahtar sözcüğü bir örnek yöntemiyle kullanıldığında, programcının, alt sınıfların yöntemi geçersiz kılmasını önleme amacında olduğunu belirtir.
- Örnek yöntemi temel sınıfta `private` erişim denetimi belirticisiyle bildirilmez. Bir yöntem temel sınıfta `private` olarak işaretlenmişse, temel sınıf yöntemi alt sınıf tarafından görülemeyeceğinden, alt sınıfta aynı şekilde adlandırılmış yöntemi tanımlarken `override` anahtar sözcüğünün kullanılması gerekmez.

  Bu kriterleri karşılayan bir örnek yöntemini geçersiz kılmak için, aşağıdaki şekilde alt sınıftaki yöntem tanımının `override` anahtar sözcüğünü kullanması ve yöntemin üst sınıf sürümüyle eşleşmesi gerekir:

- Geçersiz kılma yönteminin, temel sınıf yöntemiyle aynı erişim denetimi düzeyine sahip olması gerekir. Dahili olarak işaretlenmiş yöntemler, herhangi bir erişim denetimi belirticisi içermeyen yöntemlerle aynı erişim denetimi düzeyine sahiptir.
- Geçersiz kılma yönteminin, temel sınıf yöntemiyle aynı sayıda parametreye sahip olması gerekir.
- Geçersiz kılma yöntemi parametrelerinin, temel sınıf yöntemindeki parametrelerle aynı veri türü ek açıklamalarına sahip olması gerekir.
- Geçersiz kılma yönteminin, temel sınıf yöntemiyle aynı döndürme türüne sahip olması gerekir.

Her ikisinin de parametre sayısı ve parametrelerinin veri türü eşleştiği sürece, geçersiz kılma yöntemindeki parametrelerin adları ile temel sınıftaki parametrelerin adlarının eşleşmesi gerekmez.

super deyimi

Programcılar bir yöntemi geçersiz kılarken genellikle davranışı tamamen değiştirmek yerine geçersiz kıldıkları üst sınıf yönteminin davranışına ekleme yapmak ister. Bunun için de, bir alt sınıftaki yöntemin kendi üst sınıf sürümünü çağırmasına olanak sağlayan bir mekanizma gerekir. `super` deyimi, anında üst sınıfa başvuru içererek bu mekanizmayı sağlar. Aşağıdaki örnek, `thanks()` adındaki bir yöntemi içeren Base adında bir sınıfı ve `thanks()` yöntemini geçersiz kılan Extender adındaki bir Base sınıfının alt sınıfını tanımlar. `Extender.thanks()` yöntemi, `Base.thanks()` öğesini çağırmak için `super` deyimini kullanır.

```
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";
    }
}
```

Alıcıları ve ayarlayıcıları geçersiz kılma

Bir üst sınıfta tanımlanmış değişkenleri geçersiz kılamasanız da, alıcıları ve ayarlayıcıları geçersiz kılabilirsiniz. Örneğin, aşağıdaki kod, ActionScript 3.0'daki MovieClip sınıfında tanımlanmış `currentLabel` adındaki bir alıcıyı geçersiz kılar:

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

OverrideExample sınıf yapıcısında `trace()` çıktısı `Override: null` olup bu, örneğin miras alınan `currentLabel` özelliğini geçersiz kılabildiğini gösterir.

Statik özellikler miras alınmaz

Statik özellikler, alt sınıflar tarafından miras alınmaz. Başka bir deyişle, statik özelliklere bir alt sınıfın örneği üzerinden erişilemez. Statik özelliğe yalnızca tanımlanmış olduğu sınıf nesnesi üzerinden erişilebilir. Örneğin, aşağıdaki kod, Base adında bir temel sınıfı ve Base sınıfını genişleten Extender adında bir alt sınıfı tanımlar. `test` adındaki bir statik değişken Base sınıfında tanımlanır. Aşağıdaki alıntıda yazılı kod katı modda derleme yapmaz ve standart modda bir çalışma zamanı hatası oluşturur.

```
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 { }
```

Aşağıdaki kodda gösterildiği gibi, `test` statik değişkenine yalnızca sınıf nesnesi üzerinden erişilebilir:

```
Base.test;
```

Ancak aynı adı statik özellik olarak kullanıp bir örnek özelliğinin tanımlanmasına izin verilir. Böyle bir örnek özelliği, statik özellikle aynı sınıfta veya bir alt sınıfta tanımlanabilir. Örneğin, önceki örnekte yer alan Base sınıfı, `test` adında bir örnek özelliğine sahip olabilirdi. Örnek özelliği Extender sınıfı tarafından miras alındığından, aşağıdaki kod derleme yapar ve çalıştırılır. Test örneği değişkeninin tanımı Extender sınıfına kopyalanmayıp taşınırsa da kod derleme yapar ve çalıştırılır.

```
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 {}
```

Statik özellikler ve kapsam zinciri

Statik özellikler miras alınmasa da, bunlar kendilerini tanımlayan sınıfın ve o sınıfın alt sınıflarının kapsam zincirinde bulunur. Bu nedenle de, statik özelliklerin hem tanımlandıkları sınıfın hem de alt sınıfların *kapsamında* olduğu söylenebilir. Başka bir deyişle, bir statik özellik yalnızca statik özelliği tanımlayan sınıfın ve o sınıfın alt sınıflarının gövdesinde doğrudan erişilebilir durumdadır.

Aşağıdaki örnek, Base sınıfında tanımlanmış statik `test` değişkeninin, Extender sınıfının kapsamında olduğunu göstermek için, önceki örnekte tanımlanan sınıfları değiştirir. Başka bir deyişle, Extender sınıfı, `test` öğesini tanımlayan sınıfın adını değişkene önek olarak eklemeden statik `test` değişkenine erişebilir.

```
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
    }

}
```

Bir örnek özelliği, aynı sınıfta veya bir üst sınıfta statik özellik olarak aynı adı kullanacak şekilde tanımlanırsa, örnek özelliği kapsam zincirinde daha yüksek bir öncelik elde eder. Örnek özelliğinin statik özelliği *gölgelediği*, başka bir deyişle, statik özelliğin değeri yerine örnek özelliğinin değerinin kullanıldığı söylenebilir. Örneğin, aşağıdaki kod, Extender sınıfı `test` adında bir örnek değişkeni tanımlarsa, `trace()` deyiminin, statik değişkenin değerini değil, örnek değişkeninin değerini kullandığını gösterir:

```
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
    }

}
```

Gelişmiş başlıklar

ActionScript OOP desteğinin geçmişi

ActionScript 3.0, önceki ActionScript sürümleri esas alınarak oluşturulduğundan, ActionScript nesne modelinin nasıl geliştiğinin anlaşılması yardımcı olabilir. ActionScript, önceki Flash Professional sürümleri için basit bir komut dosyası yazma mekanizması olarak kullanılmaya başlamıştır. Daha sonra programcılar ActionScript ile gittikçe daha karmaşık uygulamalar oluşturmaya başladı. Bu programcıların gereksinimlerine yanıt vermek için, sonraki her sürüme, karmaşık uygulamaların oluşturulmasını kolaylaştıran dil özellikleri eklendi.

ActionScript 1.0

ActionScript 1.0, Flash Player 6 ve öncesinde kullanılan dil sürümüdür. Bu geliştirme aşamasında bile ActionScript nesne modeli, temel veri türü olarak nesne kavramını esas almıştı. ActionScript nesnesi bir grup *özellik* içeren birleşik bir veri türüdür. Nesne modeli ele alınırken *özellik* terimi, değişkenler, işlevler veya yöntemler gibi bir nesneye eklenen her şeyi içerir.

Bu birinci nesil ActionScript, `class` anahtar sözcüğüyle sınıfların tanımlanmasını desteklemese de, prototip nesne adı verilen özel bir nesne türünü kullanarak bir sınıfı tanımlayabilirsiniz. Java ve C++ gibi sınıf tabanlı dillerde yaptığınız gibi, somut nesneler olarak başlatacağınız soyut bir sınıf tanımı oluşturmak için `class` anahtar sözcüğünü kullanmak yerine, ActionScript 1.0 gibi prototip tabanlı diller, başka nesneler için bir model (veya prototip) olarak varolan bir nesneyi kullanır. Sınıf tabanlı bir dilde nesneler, o nesnenin şablonu görevini gören bir sınıfa işaret edebilse de, prototip tabanlı bir dildeki nesneler, bunun yerine nesnenin şablonu görevini gören başka bir nesneye (prototip) işaret eder.

ActionScript 1.0'da bir sınıf oluşturmak üzere o sınıf için bir yapıcı işlevi tanımlarsınız. ActionScript'te işlevler yalnızca soyut tanımlar değil, gerçek nesnelerdir. Oluşturduğunuz yapıcı işlevi, o sınıfın örnekleri için prototip nesne görevi görür. Aşağıdaki kod, Shape adında bir sınıf oluşturur ve varsayılan olarak `true` değerine ayarlanmış `visible` adında tek bir özelliği tanımlar:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Bu yapıcı işlevi, aşağıdaki gibi `new` operatörüyle başlatabileceğiniz bir Shape sınıfını tanımlar:

```
myShape = new Shape();
```

`Shape()` yapıcı işlevi nesnesi, Shape sınıfının örnekleri için prototip görevi gördüğü gibi, Shape sınıfının alt sınıfları (başka bir deyişle, Shape sınıfını genişleten diğer sınıflar) için de prototip görevi görebilir.

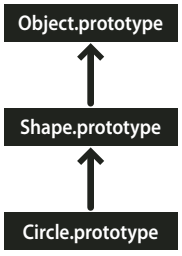
Shape sınıfının alt sınıfı olan bir sınıfın oluşturulması iki adımda gerçekleşir. İlk olarak, aşağıdaki gibi, sınıf için bir yapıcı işlevi tanımlayarak sınıfı oluşturun:

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

İkinci olarak, Shape sınıfının, Circle sınıfının prototipi olduğunu bildirmek için `new` operatörünü kullanın. Varsayılan olarak, oluşturduğunuz tüm sınıflar kendi prototipi olarak Object sınıfını kullanır, başka bir deyişle, `Circle.prototype` öğesi geçerli olarak bir genel nesne (Object sınıfının bir örneği) içerir. Circle öğesinin prototipinin Object değil Shape olduğunu belirtmek için, aşağıdaki kodu kullanarak genel bir nesneyi değil, Shape nesnesini içerecek şekilde `Circle.prototype` öğesinin değerini değiştirin:

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

Şimdi Shape sınıfı ve Circle sınıfı, *prototip zinciri* olarak bilinen bir miras ilişkisiyle birbirine bağlanır. Diyagramda, bir prototip zincirindeki ilişkiler gösterilmektedir:



Her prototip zincirinin sonundaki temel sınıf, Object sınıfıdır. Object sınıfı, ActionScript 1.0'da oluşturulmuş tüm nesneler için temel prototip nesnesine işaret eden `Object.prototype` adında statik bir özellik içerir. Örnek prototip zincirimizdeki bir sonraki nesne Shape nesnesidir. Bunun nedeni, `Shape.prototype` özelliğinin asla açıkça ayarlanmaması ve bu nedenle de genel nesne (Object sınıfının bir örneği) içermeye devam etmesidir. Bu zincirdeki son halka, prototipine (Shape sınıfına) bağlı Circle sınıfıdır. (`Circle.prototype` özelliği bir Shape nesnesi içerir.)

Aşağıdaki örnekte olduğu gibi Circle sınıfının bir örneğini oluşturursanız, örnek Circle sınıfının prototip zincirini miras alır:

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Shape sınıfının bir üyesi olarak `visible` adında bir özellik içerdiğini hatırlayın. Örneğimizde, `visible` özelliği, `myCircle` nesnesinin bir parçası olarak değil, yalnızca Shape nesnesinin bir üyesi olarak bulunur ancak aşağıdaki kod satırı `true` değerini verir:

```
trace(myCircle.visible); // output: true
```

Çalışma zamanı, prototip zincirinde gezinerek `myCircle` nesnesinin `visible` özelliğini miras aldığını doğrulayabilir. Bu kodu çalıştırırken, çalışma zamanı ilk olarak `myCircle` nesnesinin özelliklerinde `visible` adındaki bir özelliği arar, ancak bu özelliği bulamaz. Daha sonra `Circle.prototype` nesnesine bakar ancak yine `visible` adındaki özelliği bulamaz. Prototip zincirinde devam ederek en sonunda `Shape.prototype` nesnesinde tanımlanmış `visible` özelliğini bulur ve o özelliğin değerini verir.

Kolaylık olması açısından, prototip zincirinin ayrıntıları ve karmaşıklığı dahil edilmemiştir. Bunun yerine amaç ActionScript 3.0 nesne modelini anlayabilmeniz için yeterli bilgiyi sağlamaktır.

ActionScript 2.0

ActionScript 2.0, Java ve C++ gibi sınıf tabanlı dillerle çalışan kişilere tanıdık gelecek şekilde sınıflar tanımlamanıza olanak sağlayan `class`, `extends`, `public` ve `private` gibi yeni anahtar sözcükler sunmuştur. ActionScript 1.0 ile ActionScript 2.0 arasında temel alınan miras mekanizmasının değişmediğinin anlaşılması önemlidir. ActionScript 2.0 yalnızca sınıfların tanımlanması için yeni bir sözdizimi eklemiştir. Prototip zinciri, her iki dil sürümünde de aynı şekilde çalışır.

Aşağıdaki alıntıda gösterildiği gibi, ActionScript 2.0 tarafından ilk defa sunulan yeni sözdizimi, birçok programcının daha sezgisel bulduğu bir şekilde sınıfları tanımlamanıza olanak sağlar:

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

ActionScript 2.0'ın ayrıca derleme zamanı tür denetlemesiyle kullanılmak üzere tür ek açıklamaları da sunduğunu unutmayın. Bu, önceki örnekte bulunan `visible` özelliğinin yalnızca bir Boolean değeri içermesi gerektiğini bildirmenize olanak sağlar. Yeni `extends` anahtar sözcüğü, alt sınıf oluşturma işlemini de basitleştirir. Aşağıdaki örnekte, ActionScript 1.0'da iki adımda gerçekleştirilen işlem, `extends` anahtar sözcüğüyle tek adımda gerçekleştirilir:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

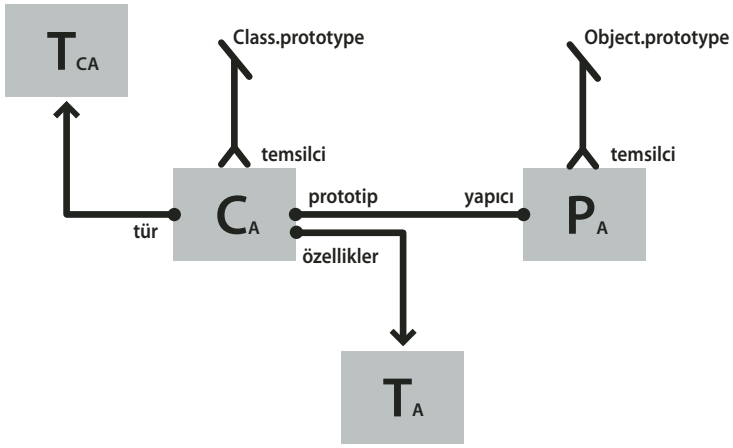
Şimdi yapıcı, sınıf tanımının parçası olarak bildirilir ve `id` ve `radius` sınıf özelliklerinin de açıkça bildirilmesi gerekir.

ActionScript 2.0 ayrıca arabirimlerin tanımı için de destek eklemiştir, bu sayede nesne tabanlı programlarınızı nesneler arası iletişim için biçimsel olarak tanımlanmış protokollerle daha düzgün hale getirebilirsiniz.

ActionScript 3.0 sınıf nesnesi

Daha çok Java ve C++ ile ilişkilendirilmiş olan yaygın bir nesne tabanlı programlama paradigması, nesne türlerini tanımlamak için sınıfları kullanır. Bu paradigmayı kullanan programlama dilleri, sınıfın tanımladığı veri türü örneklerini oluşturmak için de sınıfları kullanabilir. ActionScript, sınıfları bu iki amaç için de kullanır ancak ActionScript'in prototip tabanlı bir dil olması da ilginç bir özellik katar. ActionScript, her sınıf tanımı için, hem davranışın hem de durumun paylaşılmasına olanak sağlayan özel bir sınıf nesnesi oluşturur. Ancak birçok ActionScript programcısı için bu ayrım pratik bir kodlama anlamına gelmeyebilir. ActionScript 3.0, bu özel sınıf nesnelerini kullanmadan hatta anlamadan karmaşık nesne tabanlı ActionScript uygulamaları oluşturabileceğiniz şekilde tasarlanmıştır.

Aşağıdaki diyagram, A adındaki basit bir sınıfı temsil eden bir sınıf nesnesi yapısını göstermektedir. A sınıfı, `class A { }` deyimiyle tanımlanmıştır:



Diyagramdaki her dikdörtgen bir nesneyi temsil eder. Diyagramdaki her nesne, A sınıfına ait olduğunu temsil eden bir A alt simge karakterine sahiptir. Sınıf nesnesi (CA), birçok başka önemli nesneye başvuruları içerir. Örnek nitelikleri nesnesi (TA), bir sınıf tanımı içinde tanımlanan örnek özelliklerini saklar. Sınıf nitelikleri nesnesi (TCA), dahili sınıf türünü temsil eder ve sınıf tarafından tanımlanan statik özellikleri saklar. (C alt simge karakteri "sınıfı" ifade eder.) Prototip nesnesi (PA) her zaman başlangıçta `constructor` özelliği yoluyla eklendiği sınıf nesnesini ifade eder.

Nitelikler nesnesi

ActionScript 3.0'da yeni bir özellik olan nitelikler nesnesi, performans göz önünde tutularak uygulanmıştır. Önceki ActionScript sürümlerinde, Flash prototip zincirinde dolandığı için ad arama zaman alıcı bir işlem olabiliyordu. ActionScript 3.0'da, miras alınan özellikler, üst sınıflardan alt sınıfların nitelikler nesnesine kopyalandığından, ad arama çok daha etkili olup daha az zaman alır.

Nitelikler nesnesine programcı kodu ile doğrudan erişilemez ancak performans artışı ve bellek kullanımındaki iyileşme ile bu nesnenin varlığı hissedilebilir. Nitelikler nesnesi, bir sınıfın mizanpajı ve içerikleri hakkında ayrıntılı bilgi içeren AVM2'yi sağlar. Bu bilgiler sayesinde AVM2, zaman alıcı bir işlem olan ad aramasına gerek kalmadan özelliklere erişmeye veya yöntemler çağırmaya yönelik doğrudan makine talimatları oluşturabildiğinden, çalıştırma süresini büyük ölçüde azaltabilir.

Nitelikler nesnesi sayesinde nesnenin bellek izi, önceki ActionScript sürümlerindeki benzer bir nesnenin bellek izinden çok daha küçük olabilir. Örneğin, bir sınıf mühürlenmişse (başka bir deyişle, sınıfın dinamik olduğu bildirilmemişse), o sınıfın örneği, dinamik olarak eklenmiş özelliklerin karma tablosunu gerektirmez ve nitelikler nesnelerinin işaretçilerinden ve sınıfta tanımlanmış sabit özelliklere yönelik birkaç yuvadan biraz daha fazlasını barındırabilir. Sonuç olarak, ActionScript 2.0'da 100 bayt bellek gerektiren bir nesne, ActionScript 3.0'da 20 bayt kadar düşük bellek gerektirebilir.

Not: *Nitelikler nesnesi dahili bir uygulama olup, gelecek ActionScript sürümlerinde bu uygulamanın değişmeyeceğine veya tamamen kaldırılmayacağına dair bir garanti yoktur.*

Prototip nesnesi

Her ActionScript sınıf nesnesi, sınıfın prototip nesnesine başvuru niteliğinde olan `prototype` adında bir özelliğe sahiptir. Prototip nesnesi, ActionScript'in prototip tabanlı dil olması nedeniyle eski uygulamasının devamı niteliğindedir. Daha fazla bilgi için, bkz. ActionScript OOP desteğinin geçmişi.

`prototype` özelliği salt okunur özelliktedir, başka bir deyişle, farklı nesnelere işaret edecek şekilde değiştirilemez. Bu da, prototipin farklı bir sınıfa işaret edecek şekilde yeniden atanabildiği önceki ActionScript sürümlerindeki `prototype` sınıfından farklılık gösterir. `prototype` özelliği salt okunur olsa da, bu özelliğin başvurduğu prototip nesnesi salt okunur değildir. Başka bir deyişle, prototip nesnesine yeni özellikler eklenebilir. Prototip nesnesine eklenen özellikler, sınıfın tüm örnekleri arasında paylaşılır.

Önceki ActionScript sürümlerinde tek miras mekanizması olan prototip zinciri, ActionScript 3.0'da yalnızca ikincil rol oynar. Birincil miras mekanizması olan sabit özellik mirası, nitelikler nesnesi tarafından dahili olarak işlenir. Sabit özellik, sınıf tanımının bir parçası olarak tanımlanan bir değişken veya yöntemdir. Sabit özellik mirası, `class`, `extends` ve `override` gibi anahtar sözcüklerle ilişkilendirilmiş miras mekanizması olduğundan, sınıf mirası olarak da adlandırılır.

Prototip zinciri, sabit özellik mirasından daha dinamik olan alternatif bir miras mekanizması sağlar. Yalnızca sınıf tanımının bir parçası olarak değil, aynı zamanda sınıf nesnesinin `prototype` özelliği üzerinden çalışma zamanında da sınıfın prototip nesnesine özellikler ekleyebilirsiniz. Ancak, derleyiciyi katı moda ayarlamanız durumunda, `dynamic` anahtar sözcüğüyle bir sınıf bildirmediğiniz sürece bir prototip nesnesine eklenmiş özelliklere erişemeyebileceğinizi unutmayın.

Prototip nesnesine birçok özellik eklenmiş olan sınıfa güzel bir örnek Object sınıfıdır. Object sınıfının `toString()` ve `valueOf()` yöntemleri, gerçekten Object sınıfının prototip nesnesinin özelliklerine atanmış işlevlerdir. Aşağıda, bu yöntemlerin bildirilmesinin teoride nasıl göründüğünü gösterir (uygulama ayrıntıları nedeniyle gerçek uygulama biraz daha farklıdır):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Daha önceden belirtildiği gibi, sınıf tanımının dışında bir sınıfın prototip nesnesine bir özellik ekleyebilirsiniz. Örneğin, `toString()` yöntemi aşağıdaki gibi Object sınıfının dışında da tanımlanabilir:

```
Object.prototype.toString = function()
{
    // statements
};
```

Ancak sabit özellik mirasının aksine, prototip mirası, bir alt sınıfta yöntemi yeniden tanımlamak istediğinizde `override` anahtar sözcüğünü gerektirmez. Örneğin. Object sınıfının bir alt sınıfında `valueOf()` yöntemini yeniden tanımlamak istiyorsanız, üç seçeneğiniz vardır. İlk olarak, sınıf tanımının içinde alt sınıfın prototip nesnesinde bir `valueOf()` yöntemini tanımlayabilirsiniz. Aşağıdaki kod, Object sınıfının Foo adında bir alt sınıfını oluşturur ve sınıf tanımının bir parçası olarak Foo alt sınıfının prototip nesnesinde `valueOf()` yöntemini yeniden tanımlar. Her sınıf Object öğesinden miras aldığı için, `extends` anahtar sözcüğünün kullanılması gerekmez.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

İkinci olarak, aşağıdaki kodda gösterildiği gibi, sınıf tanımının dışında Foo alt sınıfının prototip nesnesinde bir `valueOf()` yöntemini tanımlayabilirsiniz:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Üçüncü olarak, Foo sınıfının parçası olarak `valueOf()` adında bir sabit özellik tanımlayabilirsiniz. Bu teknik, sabit özellik mirası ile prototip mirasını karma olarak kullandığından diğer tekniklerden farklıdır. `valueOf()` öğesini yeniden tanımlamak isteyen tüm Foo alt sınıflarının `override` anahtar sözcüğünü kullanması gerekir. Aşağıdaki kod, Foo'da sabit özellik olarak tanımlanan `valueOf()` öğesini gösterir:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

AS3 ad alanı

İki ayrı miras mekanizması, sabit özellik mirası ve prototip mirasının olması, çekirdek sınıfların özellikleri ve yöntemleriyle ilgili ilginç bir uyumluluk zorluğu oluşturur. ActionScript'in esas aldığı ECMAScript dil belirtimiyle uyumluluk için prototip mirasının kullanılması gerekir, başka bir deyişle, çekirdek sınıfın özellikleri ve yöntemleri o sınıfın prototip nesnesinde tanımlanır. Diğer yandan, ActionScript 3.0 ile uyumluluk için sabit özellik mirasının kullanılması gerekir, başka bir deyişle, çekirdek sınıfın özellikleri ve yöntemleri, `const`, `var` ve `function` anahtar sözcükleri kullanılarak sınıf tanımında tanımlanır. Üstelik, prototip sürümleri yerine sabit özelliklerin kullanılması, çalışma zamanı performansında önemli ölçüde artış sağlayabilir.

ActionScript 3.0, çekirdek sınıfları için hem prototip mirasını hem de sabit özellik mirasını kullanarak bu sorunu çözer. Çekirdek sınıfların her biri iki özellik ve yöntem kümesi içerir. Kümelerden biri, ECMAScript belirtimiyle uyumluluk sağlamak için prototip nesnesinde tanımlanırken, diğeri de ActionScript 3.0 ile uyumluluk sağlamak üzere sabit özellikler ve AS3 ad alanıyla tanımlanır.

AS3 ad alanı, iki özellik ve yöntem kümesi arasında seçim yapılmasına yönelik kullanışlı bir mekanizma sağlar. AS3 ad alanını kullanmazsanız, çekirdek sınıfın bir örneği, çekirdek sınıfın prototip nesnesinde tanımlanan özellikleri ve yöntemleri miras alır. Sabit özellikler her zaman prototip özelliklerden daha çok tercih edildiği için, AS3 ad alanını kullanmaya karar verirseniz, çekirdek sınıfın bir örneği AS3 sürümlerini miras alır. Başka bir deyişle, kullanılabilir bir sabit özellik olduğunda, aynı ada sahip olan prototip özelliği yerine her zaman bu sabit özellik kullanılır.

Bir özelliğin veya yöntemin AS3 ad alanı sürümünü AS3 ad alanıyla niteleyerek kullanabilirsiniz. Örneğin, aşağıdaki kod, `Array.pop()` yönteminin AS3 sürümünü kullanır:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Alternatif olarak, bir kod bloğu içindeki tüm tanımlar için AS3 ad alanını açmak üzere `use namespace` direktifini kullanabilirsiniz. Örneğin, aşağıdaki kod, `pop()` ve `push()` yöntemleri için AS3 ad alanını açmak üzere `use namespace` direktifini kullanır:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 ayrıca programınızın tamamına AS3 ad alanı uygulayabilmenizi sağlamak üzere her özellik kümesi için derleyici seçenekleri sağlar. `-as3` derleyici seçeneği, AS3 ad alanını temsil ederken, `-es` derleyici seçeneği de prototip mirası seçeneğini (`es`, ECMAScript'i ifade eder) temsil eder. Programınızın tamamı için AS3 ad alanını açmak üzere, `-as3` derleyici seçeneğini `true` değerine ve `-es` derleyici seçeneğini de `false` değerine ayarlayın. Prototip sürümlerini kullanmak için, derleyici seçeneklerini karşıt değerlere ayarlayın. Flash Builder ve Flash Professional için varsayılan derleyici ayarları şunlardır: `-as3 = true` ve `-es = false`.

Herhangi bir çekirdek sınıfı genişletmeyi ve herhangi bir yöntemi geçersiz kılmayı planlıyorsanız, geçersiz kılınmış bir yöntemi nasıl bildirmeniz gerektiğini AS3 ad alanının nasıl etkileyebildiğini anlamanız gerekir. AS3 ad alanını kullanıyorsanız, çekirdek sınıf yönteminin herhangi bir yöntem geçersiz kılması, `override` niteliğiyle birlikte AS3 ad alanını da kullanmalıdır. AS3 ad alanını kullanmıyor ve bir alt sınıfta çekirdek sınıf yöntemini yeniden tanımlamak istiyorsanız, AS3 ad alanını veya `override` anahtar sözcüğünü kullanmamanız gerekir.

Örnek: GeometricShapes

GeometricShapes örnek uygulaması, ActionScript 3.0 kullanılarak aşağıda örnekleri verilen çok sayıda nesne tabanlı kavram ve özelliklerin nasıl uygulanabildiğini gösterir:

- Sınıfları tanımlama
- Sınıfları genişletme
- Çok biçimlilik ve `override` anahtar sözcüğü
- Arabirimleri tanımlama, genişletme ve uygulama

Bu aynı zamanda, sınıf örnekleri oluşturan ve böylece bir arabirimin örneği olarak döndürme değerinin nasıl bildirildiğini ve bu döndürülen nesnenin genel olarak nasıl kullanıldığını gösteren bir "fabrika yöntemini" içerir.

Bu örneğin uygulama dosyalarını edinmek için bkz. www.adobe.com/go/learn_programmingAS3samples_flash_tr. GeometricShapes uygulama dosyalarını Samples/GeometricShapes klasöründe bulabilirsiniz. Uygulama aşağıdaki dosyaları içerir:

File	Açıklama
GeometricShapes.mxml veya GeometricShapes.fla	Flash (FLA) veya Flex (MXML) içindeki ana uygulama dosyası.
com/example/programmingas3/geometricshapes/IGeometricShape.as	Tüm GeometricShapes uygulama sınıfları tarafından uygulanacak yöntemleri tanımlayan temel arabirim.
com/example/programmingas3/geometricshapes/IPolygon.as	Birden çok kenarı olan GeometricShapes uygulama sınıfları tarafından uygulanacak yöntemleri tanımlayan bir arabirim.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Şeklin merkezi etrafında simetrik olarak konumlandırılmış eşit uzunlukta kenarlara sahip bir geometrik şekil türü.
com/example/programmingas3/geometricshapes/Circle.as	Bir daireyi tanımlayan geometrik şekil türü.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Eşit kenar bir üçgeni tanımlayan bir RegularPolygon alt sınıfı.
com/example/programmingas3/geometricshapes/Square.as	Kareyi tanımlayan bir RegularPolygon alt sınıfı.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Belirli bir tür ve boyutta şekiller oluşturulması için fabrika yöntemini içeren bir sınıf.

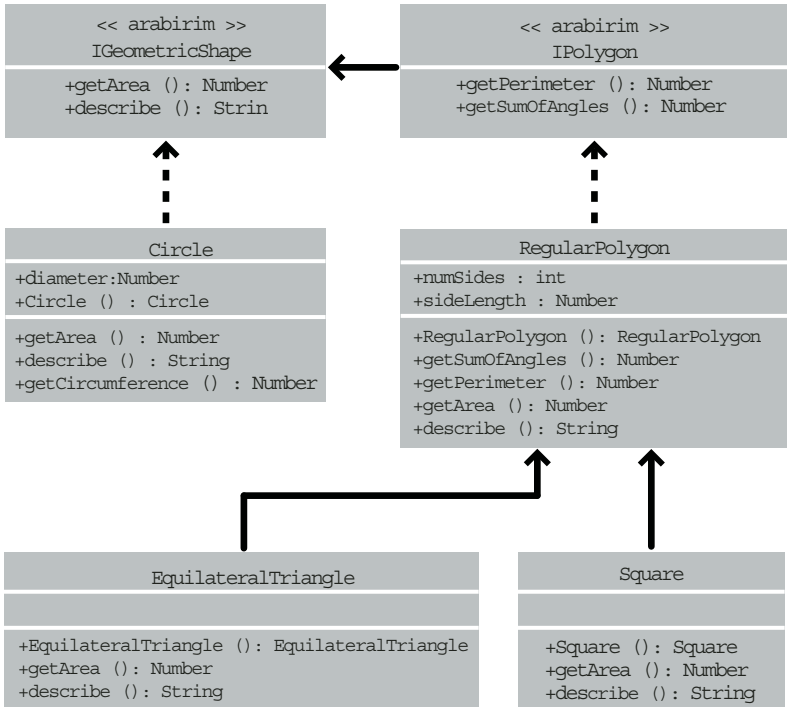
GeometricShapes sınıflarını tanımlama

GeometricShapes uygulaması, kullanıcının bir geometrik şekil türünü ve boyutunu belirtmesine olanak sağlar. Bu daha sonra şeklin açıklamasını, alanını ve çevre uzunluğunu içeren bir yanıt verir.

Uygulama kullanıcı arabirimi, şeklin türünün seçilmesi, boyutun ayarlanması ve açıklamanın görüntülenmesine yönelik birkaç denetim içerecek şekilde küçük ve basittir. Bu uygulamanın en ilginç bölümü, yüzeyin altında, sınıf yapısında ve arabirimin kendisinde yer alır.

Bu uygulama geometrik şekillerle ilgilidir ancak bunları grafiksel olarak görüntülemez.

Bu örnekte geometrik şekilleri tanımlayan sınıflar ve arabirimler, aşağıdaki diyagramda Unified Modeling Language (UML) notasyonu kullanılarak gösterilmektedir:



GeometricShapes Örnek Sınıflar

Arabirimlerle ortak davranışı tanımlama

Bu GeometricShapes uygulaması, üç tür şekli ele alır: daireler, kareler ve eşkenar üçgenler. GeometricShapes sınıf yapısı, üç şekil türü için de ortak olan yöntemleri listeleyen IGeometricShape adındaki çok basit bir arabirimle başlar:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

Bu arabirim iki yöntemi tanımlar: şeklin alanını hesaplayıp döndüren `getArea()` yöntemi ve şeklin özelliklerinin metin açıklamasını bir araya getiren `describe()` yöntemi.

Ayrıca her şeklin çevre uzunluğunu da bilmek isteyebilirsiniz. Ancak dairenin çevresi benzersiz bir şekilde hesaplanır ve bu nedenle de davranış bir üçgenin veya kareninkinden farklıdır. Yine de üçgenler, kareler ve diğer çokgenler arasında yeterince benzerlik olduğundan, yalnızca bunlara yönelik yeni bir arabirim sınıfının tanımlanması mantıklıdır: IPolygon. Ayrıca burada gösterildiği gibi IPolygon arabirimi daha basittir:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Bu arabirim, tüm çokgenler için ortak olan iki yöntemi tanımlar: tüm kenarların birleşik uzunluğunu hesaplayan `getPerimeter()` yöntemi ve tüm iç açıları toplayan `getSumOfAngles()` yöntemi.

IPolygon arabirimi, IGeometricShape arabirimini genişletir, başka bir deyişle, IPolygon arabirimini uygulayan tüm sınıfların, IGeometricShape arabiriminden iki tane ve IPolygon arabiriminden iki tane olmak üzere dört yöntemin hepsini bildirmesi gerekir.

Şekil sınıflarını tanımlama

Tüm şekil türleri için ortak olan yöntemler hakkında iyice fikir edindikten sonra, şekil sınıflarını tanımlayabilirsiniz. Uygulamanız gereken yöntem sayısı açısından en basit şekil, burada gösterildiği gibi Circle sınıfıdır:

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

Circle sınıfı, IGeometricShape arabirimini uygular, bu nedenle hem `getArea()` yöntemi hem de `describe()` yöntemi için kod sağlamalıdır. Ayrıca, Circle sınıfı için benzersiz olan `getCircumference()` yöntemini tanımlar. Circle sınıfı da diğer çokgen sınıflarında bulunmayan bir özellik olarak `diameter` özelliğini bildirir.

Diğer iki şekil türü olan kareler ve eşkenar üçgenler, ortak başka şeylere sahiptir: bunların her biri eşit uzunlukta kenarlara sahiptir ve her ikisi için de çevre uzunluğunu ve iç açıları toplamını hesaplamakta kullanabileceğiniz ortak formüller vardır. Aslında bu ortak formüller, gelecekte de tanımlayacağınız diğer normal çokgenler için de geçerlidir.

RegularPolygon sınıfı hem Square sınıfı hem de EquilateralTriangle sınıfı için üst sınıftır. Üst sınıf, ortak yöntemleri tek bir yerde tanımlamanıza olanak sağlar, böylece alt sınıfların her birinde bunları ayrı ayrı tanımlamanız gerekmez. RegularPolygon sınıfının kodu şöyledir:

```
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;
        }
    }
}
```

İlk olarak, RegularPolygon sınıfı, tüm normal çokgenler için ortak olan iki özelliği bildirir: kenarların her birinin uzunluğu (`sideLength` özelliği) ve kenar sayısı (`numSides` özelliği).

RegularPolygon sınıfı IPolygon arabirimini uygular ve IPolygon arabirim yöntemleri için dört yöntemi de bildirir. Ortak formülleri kullanarak bunlardan ikisini (`getPerimeter()` ve `getSumOfAngles()` yöntemleri) uygular.

`getArea()` yönteminin formülü şekilden şekle göre değiştiği için, yöntemin temel sınıf sürümü, alt sınıf yöntemleri tarafından miras alınabilen ortak mantığı içeremez. Bunun yerine, alanın hesaplanmadığını belirtmek için 0 varsayılan değerini döndürür. Her şeklin alanını doğru şekilde hesaplamak için, RegularPolygon sınıfının alt sınıfları, `getArea()` yöntemini geçersiz kılmalıdır.

Aşağıdaki EquilateralTriangle sınıfının kodu, `getArea()` yönteminin nasıl geçersiz kılındığını gösterir:

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

`override` anahtar sözcüğü, `EquilateralTriangle.getArea()` yönteminin RegularPolygon üst sınıfından `getArea()` yöntemini kasıtlı olarak geçersiz kıldığını belirtir. `EquilateralTriangle.getArea()` yöntemi çağrıldığında, önceki kodda bulunan formülü kullanarak alanı hesaplar ve `RegularPolygon.getArea()` yöntemindeki kod asla çalıştırılmaz.

Buna karşılık, EquilateralTriangle sınıfı, kendi `getPerimeter()` yöntemi sürümünü tanımlamaz. `EquilateralTriangle.getPerimeter()` yöntemi çağrıldığında, çağrı miras zincirinde yukarı gider ve RegularPolygon üst sınıfının `getPerimeter()` yönteminde kodu çalıştırır.

`EquilateralTriangle()` yapıcısı, üst sınıfının `RegularPolygon()` yapıcısını açıkça çağırmak için `super()` deyimini kullanır. Her iki yapıcı da aynı parametre kümesine sahip olsaydı, `EquilateralTriangle()` yapıcısı tamamen çıkarılabilir ve bunun yerine `RegularPolygon()` yapıcısı çalıştırılabilirdi. Ancak, `RegularPolygon()` yapıcısı `numSides` adında fazladan bir parametre gerektirir. Bu nedenle `EquilateralTriangle()` yapıcısı, üçgenin kenarının olduğunu belirtmek için `len` girdi parametresi ve 3 değeriyle birlikte iletilen `super(len, 3)` öğesini çağırır.

`describe()` yöntemi ayrıca `super()` deyimini farklı bir şekilde kullanır. `describe()` yönteminin RegularPolygon üst sınıf sürümünü çağırmak için kullanır. `EquilateralTriangle.describe()` yöntemi ilk olarak `desc` dize değişkenini şeklin türüyle ilgili bir deyime ayarlar. Daha sonra `super.describe()` öğesini çağırarak `RegularPolygon.describe()` yönteminin sonuçlarını alır ve bu sonuçları `desc` dizesine ekler.

Square sınıfı burada ayrıntılı şekilde ele alınmamıştır ancak bu sınıf, EquilateralTriangle sınıfına çok benzeyip bir yapıcı ve `getArea()` ve `describe()` yöntemlerinin kendi uygulamalarını sağlar.

Çok biçimlilik ve fabrika yöntemi

Arabirim ve mirastan yararlanan bir sınıf kümesi birçok ilginç şekilde kullanılabilir. Örneğin, şu ana kadar açıklanan şekil sınıflarının tümü IGeometricShape arabirimini uygular veya bir üst sınıfı genişletir. Bu nedenle de, bir değişkeni IGeometricShape örneği olacak şekilde tanımlarsanız, o değişkene yönelik `describe()` yöntemini çağırmak için söz konusu değişkenin gerçekte Circle sınıfının mı yoksa Square sınıfının mı bir örneği olduğunu bilmenize gerek yoktur.

Aşağıdaki kod, bunun nasıl olduğunu gösterir:

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Değişken IGeometricShape arabiriminin bir örneği olarak tanımlansa da temel alınan sınıf Circle olduğundan, `myShape.describe()` öğesi çağrıldığında `Circle.describe()` yöntemini çalıştırır.

Bu örnek, çok biçimlilik ilkesinin uygulanmasını gösterir: tamamen aynı yöntem çağrısı, yöntemi çağrılan nesne sınıfına bağlı olarak, farklı bir kodun çalıştırılmasına neden olur.

GeometricShapes uygulaması, fabrika yöntemi olarak bilinen basitleştirilmiş bir tasarım modeli sürümü kullanarak bu türde bir arabirim tabanlı çok biçimliliği uygular. *Fabrika yöntemi* terimi, temel alınan veri türü veya içerikleri bağlama göre değişebilen bir nesneyi döndüren işlevi ifade eder.

Burada gösterilen GeometricShapeFactory sınıfı, `createShape()` adındaki bir fabrika yöntemini tanımlar:

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

`createShape()` fabrika yöntemi, yeni nesnelerin uygulama tarafından daha genel şekilde işlenebilmesi için yeni nesneleri IGeometricShape örnekleri olarak döndürürken, şekil alt sınıf yapıcılarının, oluşturdukları örneklerin ayrıntılarını tanımlamasına da olanak sağlar.

Önceki örnekte bulunan `describeShape()` yöntemi, bir uygulamanın daha belirli bir nesnenin genel başvurusunu almak için nasıl fabrika yöntemini kullanabildiğini gösterir. Uygulama, şu şekilde, yeni oluşturulmuş bir Circle nesnesinin açıklamasını alabilir:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

`describeShape()` yöntemi, aynı parametrelerle `createShape()` fabrika yöntemini çağırarak yeni Circle nesnesini IGeometricShape nesnesi olarak yazılmış `currentShape` adındaki bir statik değişkende saklar. Daha sonra, `currentShape` nesnesinde `describe()` yöntemi çağrılır ve `Circle.describe()` yöntemini çalıştırıp dairenin ayrıntılı bir açıklamasını döndürmek için bu çağrı otomatik olarak çözümlenir.

Örnek uygulamayı geliştirme

Arabirimlerin ve mirasın gerçek gücü, uygulamanızı geliştirdiğinizde veya değiştirdiğinizde belirgin olur.

Bu örnek uygulamaya yeni bir şekil olarak beşgen eklemek istediğinizi varsayın. Bu durumda, RegularPolygon sınıfını genişleten ve `getArea()` ve `describe()` yöntemlerinin kendi sürümlerini tanımlayan bir Pentagon sınıfı oluşturursunuz. Daha sonra, uygulamanın kullanıcı arabiriminde açılır kutuya yeni bir Pentagon seçeneği eklersiniz. Ve işte hepsi budur. Pentagon sınıfı miras yoluyla RegularPolygon sınıfından `getPerimeter()` yönteminin ve `getSumOfAngles()` yönteminin işlevselliğini otomatik olarak alır. Pentagon örneği, IGeometricShape arabirimini uygulayan bir sınıftan miras aldığından, IGeometricShape örneği olarak da değerlendirilebilir. Başka bir deyişle, yeni bir şekil türü eklemek için, GeometricShapeFactory sınıfındaki yöntemlerden herhangi birinin yöntem imzasını değiştirmeniz gerekmez. (Dolayısıyla, GeometricShapeFactory sınıfını kullanan kodlardan herhangi birini de değiştirmeniz gerekmez.)

Arabirimlerin ve mirasın bir uygulamaya yeni özellikler eklemeye yönelik iş yükünü nasıl azalttığını görmek için, uygulama amacıyla Geometric Shapes örneğine bir Pentagon sınıfı eklemek isteyebilirsiniz.