

Poznajemy język ACTIONSCRIPT® 3.0

© 2010 Adobe Systems Incorporated. All rights reserved.

Poznajemy język ActionScript® 3.0

This guide is protected under copyright law, furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

This guide is licensed for use under the terms of the Creative Commons Attribution Non-Commercial 3.0 License. This License allows users to copy, distribute, and transmit the guide for noncommercial purposes only so long as (1) proper attribution to Adobe is given as the owner of the guide; and (2) any reuse or distribution of the guide contains a notice that use of the guide is governed by these terms. The best way to provide notice is to include the following link. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/>

Adobe, the Adobe logo, Adobe AIR, ActionScript, AIR, Flash, Flash Builder, Flash Lite, Flex, MXML, and Pixel Bender are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

ActiveX and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Macintosh is a trademark of Apple Inc., registered in the United States and other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

Updated Information/Additional Third Party Code Information available at www.adobe.com/go/thirdparty.

Portions include software under the following terms:

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

This product includes software developed by Fourthought, Inc. (<http://www.fourthought.com>).

MPEG Layer-3 audio compression technology licensed by Fraunhofer IIS and Thomson Multimedia (<http://www.iis.fhg.de/amm/>).

This software is based in part on the work of the Independent JPEG Group.

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com).

Video in Flash Player is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. <http://www.on2.com>.

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.



Sorenson Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Notice to U.S. Government End Users: The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Spis treści

Rozdział 1: Wprowadzenie do języka ActionScript 3.0

Informacje o języku ActionScript	1
Zalety języka ActionScript 3.0	1
Co nowego w języku ActionScript 3.0	2

Rozdział 2: Pierwsze kroki w języku ActionScript

Podstawy programowania	5
Praca z obiektami	7
Typowe elementy programu	15
Przykład: Animowana prezentacja prac (Flash Professional)	17
Budowanie aplikacji w języku ActionScript	20
Tworzenie własnych klas	24
Przykład: tworzenie prostej aplikacji	26

Rozdział 3: Język ActionScript i jego składnia

Przegląd języka	35
Obiekty i klasy	36
Pakiety i przestrzeń nazw	37
Zmienne	46
Typy danych	50
Składnia	62
Operatory	67
Instrukcje warunkowe	73
Pętle	75
Funkcje	77

Rozdział 4: Programowanie zorientowane obiektowo w języku ActionScript

Wprowadzenie do programowania zorientowanego obiektowo	89
Klasy	89
Interfejsy	104
Dziedziczenie	106
Tematy zaawansowane	115
Przykład: GeometricShapes	121

Rozdział 1: Wprowadzenie do języka ActionScript 3.0

Informacje o języku ActionScript

ActionScript to język programowania przeznaczony dla środowisk wykonawczych Adobe® Flash® Player i Adobe® AIR™. Umożliwia realizację funkcji interaktywnych, przetwarzania danych oraz różnorodnej innej funkcjonalności w ramach treści i aplikacji przeznaczonych dla środowisk Flash, Flex i AIR.

Kod w języku ActionScript jest wykonywany w maszynie wirtualnej ActionScript Virtual Machine (AVM), wchodzącej w skład programu Flash Player i środowiska AIR. Kod ActionScript jest zwykle przekształcany na format kodu bajtowego przez kompilator. (*Kod bajtowy* jest typem języka programowania, który jest zapisywany i rozumiany przez komputery. Przykładami kompilatorów są: kompilator wbudowany w programie Adobe® Flash® Professional, wbudowany w programie Adobe® Flash® Builder™ oraz dostępny w pakiecie Adobe® Flex™ SDK. Kod bajtowy jest osadzony w plikach SWF wykonywanych przez program Flash Player i środowisko AIR.

ActionScript 3.0 oferuje solidny model programowania znany projektantom z podstawową wiedzą na temat programowania obiektowego. Do kluczowych funkcji języka ActionScript 3.0, które stanowią udoskonalenie w porównaniu z poprzednimi wersjami ActionScript, należą:

- Nowa maszyna wirtualna ActionScript Virtual Machine, nazwana AVM2, w której stosowany jest nowy zestaw instrukcji kodu bajtowego i która charakteryzuje się znacznie wyższą wydajnością.
- Bardziej nowoczesna baza kodu kompilatora, która zapewnia dokładniejszą optymalizację niż poprzednie wersje kompilatora
- Rozbudowany i udoskonalony interfejs programowania aplikacji (API, ang. application programming interface) umożliwiający sterowanie obiektami na niskim poziomie i oferujący prawdziwie obiektowy model programowania.
- Interfejs API XML oparty na specyfikacji ECMAScript for XML (E4X) (ECMA-357 edition 2). E4X jest rozszerzeniem języka ECMAScript, w którym dodano typ XML jako jeden z rodzimych typów.
- Model zdarzeń oparty na specyfikacji Document Object Model (DOM) Level 3 Events Specification.

Zalety języka ActionScript 3.0

ActionScript w wersji 3.0 jest czymś więcej niż język skryptowy, którym w istocie we wcześniejszych wersjach. Został opracowany z myślą tworzeniu złożonych aplikacji operujących na obszernych zbiorach danych oraz opartych na obiektowo-zorientowanych fragmentach kodu przeznaczonych do wielokrotnego wykorzystania w różnych projektach. Język ActionScript 3.0 nie jest wymagany w przypadku zawartości działającej w programie Adobe Flash Player. Jednak umożliwia zwiększanie wydajności, ponieważ zapewnia dostęp do AVM2 — wirtualnej maszyny języka ActionScript 3.0. Kod ActionScript 3.0 może być wykonywany do dziesięciu razy szybciej niż starszy kod ActionScript.

Poprzednia wersja ActionScript Virtual Machine, AVM1, wykonuje kod ActionScript 1.0 i ActionScript 2.0. Programy Flash Player 9 i 10 obsługują wersję AVM1, dzięki czemu zapewniają kompatybilność wsteczną.

Co nowego w języku ActionScript 3.0

Język ActionScript 3.0 zawiera wiele klas i funkcji podobnych do stosowanych w językach ActionScript 1.0 i 2.0. Jednak ActionScript 3.0 różni się od poprzednich wersji ActionScript pod względem architektury i koncepcji. Udoskonalenia wprowadzone w wersji ActionScript 3.0 to między innymi nowe elementy języka podstawowego i udoskonalony interfejs API programu, który zapewnia większą kontrolę nad obiektami niskopoziomymi.

Cechy i elementy języka podstawowego

Język podstawowy to opis bazowych elementów języka programowania, takich jak instrukcje, wyrażenia, warunki, pętle i typy. Język ActionScript 3.0 zawiera wiele elementów, które przyspieszają proces programowania.

Wyjątki w czasie wykonywania

Środowisko języka ActionScript 3.0 sygnalizuje więcej rodzajów błędów niż poprzednie wersje języka ActionScript. Wyjątki w czasie wykonywania odzwierciedlają typowe błędy, ułatwiając proces debugowania i umożliwiając tworzenie aplikacji, które niezawodnie obsługują wszelkie błędy. Błędy w czasie wykonywania mogą być zgłaszane razem z zapisami śladu stosu, informacjami o pliku źródłowym i numerze wiersza, co znacznie przyspiesza lokalizowanie błędów.

Obsługa typów w czasie wykonywania

W języku ActionScript 3.0 informacje o typach są w czasie wykonywania zachowywane. Te informacje są wykorzystywane w celu sprawdzania typu w czasie wykonywania, co poprawia bezpieczeństwo typów systemowych. Informacje o typach są też wykorzystywane przy tworzeniu reprezentacji zmiennych w rodzimej formie maszynowej, co przyspiesza wykonywanie kodu i ogranicza zużycie pamięci. W języku ActionScript 2.0 adnotacje dotyczące typów są tworzone głównie przez programistę i wszystkie wartości są dynamicznie wpisywane w czasie wykonywania.

Klasy zapieczętowane

W języku ActionScript 3.0 objęto pojęcie klas zapieczętowanych. Klasa zapieczętowana ma ustalony zestaw właściwości i metod zdefiniowany w czasie kompilacji; nie można do niej dodawać nowych właściwości ani metod. Brak możliwości zmiany klasy w czasie wykonywania umożliwia dokładniejsze sprawdzanie czasu kompilacji, co zwiększa niezawodność programów. Optymalizuje również wykorzystanie pamięci, ponieważ nie wymaga tworzenia wewnętrznej tabeli mieszającej dla każdej instancji obiektu. Nadal możliwe jest tworzenie klas dynamicznych, które jednak muszą być deklarowane ze słowem kluczowym `dynamic`. W języku ActionScript 3.0 wszystkie klasy są domyślnie zapieczętowane, ale dodanie w deklaracji słowa kluczowego `dynamic` umożliwia utworzenie klasy dynamicznej.

Zamknięcia metod

W języku ActionScript 3.0 zamknięcia metod mogą automatycznie zapamiętywać, do jakiej instancji obiektu pierwotnie należały. Ta funkcja jest przydatna przy obsłudze zdarzeń. W języku ActionScript 2.0 zamknięcia metod nie pamiętały, z których instancji obiektów zostały wyodrębnione, co prowadziło do nieoczekiwanych zachowań po wywołaniu zamknięcia metody.

ECMAScript for XML (E4X)

W języku ActionScript 3.0 zaimplementowano specyfikację ECMAScript for XML (E4X), niedawno ujętą w standardzie ECMA-357. Specyfikacja E4X zawiera zestaw wygodnych, naturalnie skonstruowanych elementów języka służących do manipulowania kodem XML. Inaczej niż w tradycyjnych interfejsach API służących do analizy składniowej kody XML, kod XML wraz z elementami specyfikacji E4X może być traktowany jak rodzimy typ danych języka. Specyfikacja E4X usprawnia tworzenie aplikacji, które manipulują kodem XML, ponieważ radykalnie skraca kod wymagany do realizacji takich manipulacji.

Ze specyfikacją ECMA E4X można zapoznać się pod adresem www.ecma-international.org.

Wyrażenie regularne

Język ActionScript 3.0 oferuje rodzimą obsługę wyrażeń regularnych, ułatwiając szybkie przeszukiwanie ciągów znaków i manipulowanie nimi. Implementacja wyrażeń regularnych w języku ActionScript 3.0 jest zgodna ze specyfikacją języka ECMAScript (ECMA-262) edition 3.

Przestrzenie nazw

Przestrzenie nazw działają podobnie jak tradycyjne specyfikatory dostępu używane do sterowania widocznością deklaracji (`public`, `private`, `protected`). Pełnią rolę niestandardowych specyfikatorów dostępu i mogą mieć nazwy wybrane przez programistę. Przestrzenie nazw mają przypisane uniwersalne identyfikatory URI, które zapobiegają kolizjom, a ponadto służą do reprezentacji przestrzeni nazw XML w przypadku korzystania ze specyfikacji E4X.

Nowe typy pierwotne

Język ActionScript 3.0 zawiera trzy typy liczbowe: `Number`, `int` i `uint`. `Number` reprezentuje liczby zmiennoprzecinkowe o podwójnej precyzji. Typ `int` reprezentuje 32-bitowe liczby całkowite ze znakiem i pozwala na szybkie realizowanie obliczeń na takich liczbach na poziomie procesora. Typ `int` jest przydatny do realizacji liczników pętli i zmiennych, które przechowują tylko wartości całkowite. Typ `uint` reprezentuje 32-bitowe liczby całkowite bez znaku i jest przydatny w przypadku wartości kolorów RGB, liczników bajtów itp. W języku ActionScript 2.0 dostępny jest jeden typ liczbowy o nazwie `Number`.

Charakterystyka interfejsu API programu

Interfejs API programu dla języka ActionScript 3.0 zawiera wiele klas umożliwiających sterowanie obiektami na niskim poziomie. Architektura języka jest bardziej intuicyjna niż w poprzednich wersjach. Opisanie wszystkich klas zajmie zbyt dużo miejsca, ale o niektórych klasach warto wspomnieć.

Model zdarzeń DOM3

Model zdarzeń DOM3 (Document Object Model Level 3) udostępnia standardowy sposób generowania i obsługi komunikatów o zdarzeniach. Ten model zdarzeń umożliwia obiektom w aplikacjach interakcję oraz komunikowanie, utrzymanie stanu oraz reagowanie na zmiany. Wzorcem dla modelu zdarzeń w języku ActionScript 3.0 była specyfikacja World Wide Web Consortium DOM Level 3 Events Specification. Ten model udostępnia bardziej precyzyjne i wydajniejsze mechanizmy niż mechanizmy dostępne w systemach zdarzeń z poprzednich wersji języka ActionScript.

Klasy zdarzeń i błędów są umieszczone w pakiecie `flash.events`. Składniki programu Flash Professional oraz środowisko Flex wykorzystują ten sam model zdarzeń, dlatego system zdarzeń jest jednolity w całej platformie Flash.

Interfejs API listy wyświetlania

Interfejs API zapewniający dostęp do listy wyświetlania — czyli drzewa zawierającego wszystkie elementy wizualne aplikacji — zawiera klasy służące do pracy z wizualnymi obiektami pierwotnymi.

Klasa `Sprite` jest uproszczonym blokiem, który stanowi klasę bazową dla elementów graficznych, takich jak składniki interfejsu użytkownika. Klasa `Shape` reprezentuje surowe kształty wektorowe. Instancje tych klas można tworzyć w sposób naturalny za pomocą operatora `new` i w dowolnym momencie dynamicznie zmieniać ich lokalizację w hierarchii dziedziczenia.

Zarządzanie w ramach hierarchii odbywa się automatycznie. Dostępne są metody przeznaczone do określania i zarządzania kolejnością umieszczania obiektów na stosie.

Obsługa danych i treści dynamicznych

Język ActionScript 3.0 zawiera intuicyjne i spójne w całym interfejsie API mechanizmy służące do ładowania i obsługi zasobów oraz danych w aplikacji. Klasa `Loader` udostępnia jednolity mechanizm ładowania plików SWF i obrazów oraz zapewnia dostęp do szczegółowych informacji o załadowanej treści. Klasa `URLLoader` udostępnia odrębny mechanizm ładowania tekstu i danych binarnych w aplikacjach zorientowanych na manipulowanie danymi. Klasa `Socket` oferuje mechanizmy odczytywania i zapisywania danych binarnych w gniazdach serwera, w dowolnym formacie.

Niskopoziomowy dostęp do danych

Niskopoziomowy dostęp do danych jest zapewniony dzięki różnym interfejsom API. Klasa `URLStream` udostępnia pobierane dane w postaci surowego strumienia binarnego — w trakcie pobierania. Klasa `ByteArray` umożliwia optymalizowanie odczytywania i zapisywania danych binarnych oraz pracy z takimi danymi. Interfejs API do obsługi dźwięku zapewnia precyzyjną kontrolę nad dźwiękiem za pośrednictwem klas `SoundChannel` i `SoundMixer`. Nowe elementy interfejsu API związane z bezpieczeństwem udostępniają informacje o uprawnieniach dostępu pliku SWF lub załadowanej treści, umożliwiając obsługę błędów zabezpieczeń.

Praca z tekstem

Język ActionScript 3.0 zawiera pakiet `flash.text` grupujący wszystkie elementy interfejsu API związane z tekstem. Klasa `TextLineMetrics` udostępnia szczegółowe metryki wiersza tekstu w polu tekstowym; zastępuje metodę `TextFormat.getTextExtent()` znaną z języka ActionScript 2.0. Klasa `TextField` zawiera szereg interesujących metod niskopoziomowych, które udostępniają określone informacje o wierszach tekstu lub pojedynczych znakach w polu tekstowym. Na przykład: metoda `getCharBoundaries()` zwraca prostokąt reprezentujący obwódnię znaku. Metoda `getCharIndexAtPoint()` zwraca indeks znaku w określonym położeniu. Metoda `getFirstCharInParagraph()` zwraca indeks pierwszego znaku w akapicie. Do metod operujących na poziomie wiersza należy metoda `getLineLength()`, która zwraca liczbę znaków w określonym wierszu tekstu, oraz metoda `getLineText()`, która zwraca tekst z określonego wiersza. Klasa `Font` umożliwia zarządzanie osadzonymi czcionkami w plikach SWF.

Jeszcze niższy poziom kontroli nad tekstem zapewniają klasy z pakietu `flash.text.engine`, które tworzą mechanizm Flash Text Engine. Ten zbiór klas zapewnia niski poziom kontroli nad tekstem i jest przeznaczony do tworzenia struktur i składników tekstowych.

Rozdział 2: Pierwsze kroki w języku ActionScript

Podstawy programowania

Ponieważ ActionScript jest językiem programowania, poznawanie go najlepiej zacząć od kilku ogólnych pojęć z zakresu programowania komputerów.

Co robią programy komputerowe

Przed wszystkim warto zastanowić się, czym jest i co robi program komputerowy. Program komputerowy można postrzegać w dwóch aspektach:

- Program jest szeregiem instrukcji lub kroków, które komputer ma wykonać.
- Każdy krok sprowadza się do manipulacji na elemencie informacji lub danych.

W sensie ogólnym program komputerowy jest po prostu listą kroków, które komputer kolejno wykonuje. Każdy taki krok nazywany jest *instrukcją*. W języku ActionScript każda instrukcja jest zakończona znakiem średnika.

Co do zasady działanie każdej instrukcji w programie sprowadza się do manipulacji jakimś elementem danych przechowywanym w pamięci komputera. Prosty przykładem jest instrukcja, która powoduje, że komputer dodaje dwie liczby i zapisuje w wynik w pamięci. Bardziej skomplikowaną operacją byłoby na przykład przesunięcie w inne miejsce prostokąta narysowanego na ekranie. Komputer śledzi określone informacje na temat prostokąta: współrzędne x i y jego położenia, jego szerokość i wysokość, jego kolor itd. Każda z tych informacji jest przechowywana w pamięci komputera. Program, który spowoduje przeniesienie prostokąta do innego miejsca, powinien zawierać kroki, takie jak „zmień współrzędną x na 200; zmień współrzędną y na 150”. Innymi słowy: powinien określać nowe wartości dla współrzędnych x i y. W tle komputer wykonuje operacje, które powodują przekształcenie liczb w obraz, który pojawi się na ekranie komputera. Jednak na podstawowym poziomie szczegółowości najważniejszą informacją jest to, że zadanie „przesunięcia prostokąta na ekranie” obejmuje zmianę bitów danych w pamięci komputera.

Zmienne i stałe

Programowanie dotyczy głównie zmiany fragmentów informacji w pamięci komputera. Dlatego bardzo ważne jest istnienie sposobu reprezentowania pojedynczego fragmentu informacji w programie. *Zmienna* to nazwa reprezentująca wartość w pamięci komputera. W instrukcjach manipulujących wartościami używamy zamiast tych wartości nazw zmiennych; za każdym razem, gdy komputer napotka w programie nazwę zmiennej, szuka w pamięci wartości, którą ta zmienna reprezentuje. Na przykład, jeśli mamy dwie zmienne o nazwach `value1` i `value2`, a każda z nich zawiera liczbę, możemy napisać następującą instrukcję dodawania tych dwóch liczb:

```
value1 + value2
```

Realizując program, komputer odszuka wartości każdej ze zmiennych i doda te wartości.

W języku ActionScript 3.0 zmienna faktycznie składa się z trzech różnych elementów:

- nazwy zmiennej;
- typu danych, jakie mogą być przechowywane w zmiennej;
- faktycznej wartości przechowywanej w pamięci komputera.

Wyjaśniliśmy już, że nazwa zastępuje w programie wartość zmiennej. Ważną rolę odgrywa także typ danych. Po utworzeniu zmiennej w języku ActionScript należy określić konkretny typ danych, jakie będą przechowywane w zmiennej. Od tego momentu instrukcje programu będą mogły przechowywać w zmiennej tylko dane tego typu. Wartość można modyfikować za pomocą określonych cech skojarzonych z typem danych tej wartości. Aby utworzyć zmienną w języku ActionScript (proces ten nazywamy *deklarowaniem* zmiennej), używamy instrukcji `var`:

```
var value1:Number;
```

W tym przykładzie przedstawiona instrukcja powoduje, że komputer tworzy zmienną o nazwie `value1`, która może przechowywać tylko dane typu `Number`. („`Number`” to typ danych zdefiniowany w języku ActionScript). Możemy także od razu zapisać wartość w zmiennej:

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

Adobe Flash Professional

W programie Flash Professional istnieje też inny sposób deklarowania zmiennych. Gdy umieścimy symbol klipu filmowego, symbol przycisku lub pole tekstowe na stole montażowym, możemy nadać mu nazwę instancji w Inspektorze właściwości. W tle program Flash Professional tworzy zmienną o takiej samej nazwie jak nazwa instancji. Z tej nazwy można skorzystać w kodzie ActionScript w celu reprezentowania tego elementu stołu montażowego. Załóżmy, że na stole montażowym znajduje się symbol klipu filmowego, który otrzymał nazwę instancji `rocketShip`. W każdym przypadku użycia zmiennej `rocketShip` w kodzie ActionScript każda zmiana dotyczy tego klipu filmowego.

Stała jest podobna do zmiennej. Jest to nazwa, która reprezentuje wartość o określonym typie danych w pamięci komputera. Różnica polega na tym, że stałej można przypisać wartość tylko raz w całej aplikacji ActionScript. Raz przypisana wartość stałej pozostaje niezmienna przez cały czas wykonywania aplikacji. Składnia deklarowania stałej jest niemal taka sama, jak przeznaczona do deklarowania zmiennej. Jediną różnicą jest to, że używane jest słowo kluczowe `const` zamiast słowa kluczowego `var`:

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

Stałe są użyteczne do definiowania wartości używanych w wielu miejscach w projekcie, które w normalnych okolicznościach nie ulegają zmianie. Użycie stałych zamiast wartości literalnych zwiększa czytelność kodu. Przykład: rozważmy dwie wersje tego samego kodu. Jedna z nich mnoży cenę przez `SALES_TAX_RATE`. Drugi mnoży cenę przez `0.07`. Wersja, w której wykorzystano stałą `SALES_TAX_RATE`, jest bardziej zrozumiała. Ponadto: załóżmy, że wartość zdefiniowana przez stałą ulega zmianie. Jeśli wykorzystamy stałą w celu reprezentowania tej wartości w całym projekcie, wówczas wystarczy zmienić wartość w jednym miejscu (w deklaracji stałej). Gdybyśmy zastosowali wartości literalowe zapisane w kodzie, konieczna byłaby zmiana każdego wystąpienia tej wartości.

Typy danych

W języku ActionScript istnieje wiele typów danych, które można przypisywać tworzonej zmiennym. Niektóre z tych typów są typami „prostymi” lub „podstawowymi”:

- `String`: wartość tekstowa, taka jak nazwa lub tekst rozdziału książki.
- Typy liczbowe: w języku ActionScript 3.0 istnieją trzy typy danych reprezentujące dane liczbowe:
 - `Number`: dowolna wartość liczbową, całkowitą lub ułamkową;
 - `int`: liczba całkowita (bez części ułamkowej);
 - `uint`: liczba całkowita „bez znaku”, czyli liczba całkowita nieujemna.

- Boolean: wartość true (prawda) albo wartość (false), określająca np. stan przełącznika lub informująca, czy dwie wartości są sobie równe.

Proste typy danych reprezentują pojedyncze elementy informacji: na przykład jedną liczbę lub jeden ciąg tekstowy. Jednak większość typów danych zdefiniowanych w języku ActionScript to złożone typy danych. Reprezentują one zestaw wartości w pojedynczym kontenerze. Na przykład zmienna należąca do typu danych Date reprezentuje jedną wartość — moment w czasie. Jedną w istocie wartość typu Date składa się z kilku odrębnych liczb: dnia, miesiąca, roku, godzin, minut, sekund itd. Zwykle myślimy o dacie jak o pojedynczej wartości i w języku ActionScript możliwe jest traktowanie jej w ten sposób — należy tylko utworzyć zmienną Date. Jednak komputer traktuje datę jako grupę kilku wartości, które razem tworzą pojedynczą datę.

Większość wbudowanych typów danych, a także typów danych definiowanych przez programistów, to typy złożone. Oto niektóre złożone typy danych, z którymi czytelnicy mogli się już pośrednio zetknąć:

- MovieClip: symbol klipu filmowego;
- TextField: dynamiczne lub wejściowe pole tekstowe;
- SimpleButton: symbol przycisku;
- Date: informacje o pojedynczym momencie w czasie (dacie i godzinie).

Często używanymi synonimami typu danych są: klasa i obiekt. *Klasa* jest po prostu definicją typu danych. Klasę można traktować jako szablon dla wszystkich obiektów danego typu danych, który może zawierać na przykład następujące informacje: „wszystkie zmienne typu danych Przykład mają następujące cechy: A, B i C”. *Obiekt* jest po prostu instancją klasy. Na przykład: zmienna typu MovieClip może zostać opisana jako obiekt MovieClip. Oto różne sposoby wyrażenia tego samego faktu:

- Typem danych zmiennej `myVariable` jest `Number`.
- Zmienna `myVariable` jest instancją typu `Number`.
- Zmienna `myVariable` jest obiektem typu `Number`.
- Zmienna `myVariable` jest instancją klasy `Number`.

Praca z obiektami

ActionScript należy do języków programowania zorientowanych obiektowo. Programowanie zorientowane obiektowo to po prostu podejście do programowania. W ten sposób określana jest metoda porządkowania kodu w programie przy wykorzystaniu obiektów.

Wcześniej termin „program” został zdefiniowany jako seria kroków lub instrukcji, które wykonuje komputer. A zatem możemy wyobrażać sobie program jako jedną długą listę instrukcji. Jednak w programowaniu zorientowanym obiektowo instrukcje programu są podzielone między różne obiekty. Kod jest podzielony na fragmenty wg funkcji, dlatego powiązane typy funkcji lub powiązane fragmenty informacji są zgrupowane w jednym kontenerze.

Adobe Flash Professional

Każdy kto pracował z symbolami w programie Flash Professional, miał już okazję do pracy z obiektami. Wyobraźmy sobie, że zdefiniowaliśmy symbol klipu filmowego — np. rysunek prostokąta — i umieściliśmy egzemplarz tego symbolu na stole montażowym. Ten symbol klipu filmowego jest także (literalnie) obiektem w języku ActionScript; stanowi instancję klasy `MovieClip`.

Klip filmowy ma szereg cech, które możemy modyfikować. Po wybraniu klipu w Inspektorze właściwości można zmienić wartości, takie jak współrzędna x lub szerokość. Możliwe jest również dopasowywanie kolorów np. poprzez zmianę współczynnika alfa (przezroczystość) lub zastosowanie filtra cieniowania. Inne narzędzia programu Flash Professional pozwalają na wprowadzanie dalszych zmian, np. narzędzie Transformacja swobodna pozwala na obrócenie prostokąta. Wszystkie metody modyfikowania klipu filmowego w programie Flash Professional są również dostępne w języku ActionScript. Modyfikowanie klipu filmowego w języku ActionScript polega na zmianie elementów informacji, które są umieszczane w pojedynczym pakunku nazywanym obiektem MovieClip.

W obiektowo zorientowanym języku ActionScript każda klasa może mieć trzy rodzaje cech:

- właściwości;
- metody;
- zdarzenia.

Te elementy służą do zarządzania elementami danych używanymi w programie i do podejmowania decyzji o wykonywanych czynnościach oraz kolejności czynności.

Właściwości

Właściwość reprezentuje jeden z elementów danych, które powiązane ze sobą składają się na obiekt. Przykład obiektu utworu muzycznego może mieć właściwości o nazwie `artist` i `title`; klasa `MovieClip` ma takie właściwości, jak `rotation`, `x`, `width` i `alpha`. Właściwości należy traktować jako pojedyncze zmienne. W rzeczywistości właściwości można traktować jako zmienne „podrzędne” zawarte w obiekcie.

Oto przykłady kodu w języku ActionScript, w których używane są właściwości. Ten wiersz kodu przenosi obiekt `MovieClip` o nazwie `square` do położenia o współrzędnej `x=100` pikseli:

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

W tym przykładzie użyto właściwości `rotation` w celu obrócenia obiektu `MovieClip` o nazwie `square` o taki kąt, by był obrócony tak samo, jak obiekt `MovieClip` o nazwie `triangle`:

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

Ten przykład modyfikuje skalę poziomą obiektu `MovieClip` `square`, tak aby był o półtora raza szerszy niż dotychczas:

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

Należy zwrócić uwagę na wspólną strukturę: używamy zmiennej (`square`, `triangle`) jako nazwy obiektu, po niej stawiamy kropkę (`.`), a następnie podajemy nazwę właściwości (`x`, `rotation`, `scaleX`). Kropka, formalnie nazywana *operatorem kropki*, oznacza odwołanie do jednego z elementów podrzędnych zawartych w obiekcie. Cała struktura „nazwa zmiennej-kropka-nazwa właściwości” może być używana tak, jak pojedyncza zmienna, czyli nazwa pojedynczej wartości w pamięci komputera.

Metody

Metoda jest działaniem, które może wykonać obiekt. Na przykład: założmy, że, w programie Flash Professional utworzony został symbol klipu filmowego z kilkoma klatkami i klatkami kluczowymi na osi czasu. Ten klip filmowy może zostać odtworzony lub zatrzymany, a także możliwe jest przesunięcie głowicy odtwarzania do wybranej klatki.

Ten kod nakazuje obiektowi `MovieClip` o nazwie `shortFilm` rozpoczęcie odtwarzania:

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

Ten wiersz zatrzymuje odtwarzanie obiektu MovieClip o nazwie `shortFilm` (głowica odtwarzania zatrzymuje się w bieżącym położeniu, tak jak przy wstrzymaniu odtwarzania):

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

Ten kod powoduje, że głowica odtwarzania obiektu MovieClip o nazwie `shortFilm` jest przenoszona do klatki 1, a odtwarzanie jest zatrzymywane (tak jak przy przewinięciu materiału wideo):

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

Dostęp do metod, podobnie jak do właściwości, uzyskuje się poprzez podanie nazwy obiektu (zmiennej), kropki i nazwy metody, po której następują nawiasy. Nawiasy oznaczają, że *wywołujemy* metodę — czyli że nakazujemy obiektowi wykonanie danej operacji. Niekiedy w nawiasach umieszcza się wartości (lub zmienne), aby w ten sposób przekazać dodatkowe informacje potrzebne do wykonania operacji. Wartości te nazywane są *parametrami* metody. Na przykład metoda `gotoAndStop()` potrzebuje informacji o tym, do której klatki ma przejść, a zatem w nawiasach należy przekazać jej jeden parametr. Inne metody, jak `play()` i `stop()`, odzwierciedlają oczywiste operacje i nie wymagają żadnych dodatkowych informacji. Mimo to w ich zapisie także należy używać nawiasów.

W przeciwieństwie do właściwości (i zmiennych) metody zastępują wartości. Jednak niektóre metody mogą wykonywać obliczenia i zwracać wynik, którego następnie można używać tak, jak zmiennej. Na przykład metoda `toString()` klasy `Number` konwertuje wartość liczbową na jej reprezentację tekstową:

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

Metody `toString()` użylibyśmy na przykład po to, aby wyświetlić wartość zmiennej `Number` w polu tekstowym na ekranie. Właściwość `text` klasy `TextField` jest zdefiniowana jako `String`, a więc może zawierać tylko wartości tekstowe. (Właściwość `text` reprezentuje rzeczywistą treść tekstu wyświetlanego na ekranie). Ten wiersz kodu powoduje przekształcenie wartości liczbowej w zmiennej `numericData` na tekst. Następnie wartość jest wyświetlana na ekranie w obiekcie `TextField` i nazwie `calculatorDisplay`:

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

Zdarzenia

Program komputerowy jest szeregiem instrukcji wykonywanych krok po kroku przez komputer. Niektóre proste programy komputerowe faktycznie sprowadzają się do kilku kroków, po wykonaniu których program kończy działanie. Jednak programy w języku ActionScript mogą także oczekiwać na wprowadzenie danych przez użytkownika oraz na inne zdarzenia. Zdarzenia to mechanizm decydujący o tym, które instrukcje komputer wykonuje i kiedy je wykonuje.

Co do zasady *zdarzenia* są sytuacjami, które środowisko ActionScript rozpoznaje i na które potrafi zareagować. Wiele zdarzeń jest związanych z interakcją z użytkownikiem, na przykład: z kliknięciem przycisku lub naciśnięciem klawisza na klawiaturze. Istnieją również inne typy zdarzeń. Na przykład, jeśli w programie ActionScript ładujemy zewnętrzny obraz, zachodzi zdarzenie informujące o zakończeniu ładowania. Zgodnie z koncepcją: podczas działania program ActionScript oczekuje na zdarzenia. Gdy wystąpią zdarzenia, wówczas zostanie wykonany konkretny kod ActionScript określonych dla tych zdarzeń.

Podstawy obsługi zdarzeń

Technika określania operacji, które powinny być wykonywane w odpowiedzi na określone zdarzenia, nazywana jest *obsługą zdarzeń*. Pisząc w języku ActionScript kod obsługujący zdarzenia, musimy brać pod uwagę trzy ważne elementy:

- Źródło zdarzenia: w którym obiekcie zachodzi zdarzenie? Na przykład, który przycisk został kliknięty lub który obiekt Loader ładuje obraz? Źródło zdarzenia jest również określane jako *cel zdarzenia*. Taka nazwa jest spowodowana tym, że jest to obiekt, do którego komputer kieruje zdarzenie (miejsce, w którym w rzeczywistości dojdzie do zdarzenia).
- Zdarzenie: co takiego ma się stać i na co będziemy reagować? Identyfikacja zdarzenia jest ważna, ponieważ wiele obiektów może wywoływać kilka zdarzeń.
- Reakcja: jakie operacje chcemy wykonać, gdy zajdzie zdarzenie?

Każdy kod ActionScript przeznaczony do obsługi zdarzeń wymaga następujących elementów. Kod jest zgodny z następującą podstawową strukturą (elementy pogrubione są elementami zastępczymi, które zostaną wprowadzone w konkretnym przypadku):

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

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

Ten kod ma dwie funkcje. Po pierwsze definiuje funkcję, określając w ten sposób operacje, które mają być wykonane w reakcji na zdarzenie. Następnie wywołuje metodę `addEventListener()` obiektu źródłowego. Wywołanie metody `addEventListener()` powoduje „subskrypcję” funkcji dla określonego zdarzenia. Gdy wystąpi zdarzenie, zostaną wykonane operacje funkcji. Teraz przyjrzymy się bliżej każdej z tych części kodu.

Funkcja to sposób zgrupowania operacji pod jedną nazwą, pozwalającą na skrótowe wywołanie tych operacji. Funkcja jest identyczna z metodą, ale nie musi być skojarzona z konkretną klasą. (Termin „metoda” może zostać zdefiniowany jako funkcja, która jest powiązana z określoną klasą). Tworząc funkcję do obsługi zdarzeń, należy wybrać nazwę funkcji (w tym przypadku jest to nazwa `eventResponse`). Należy także określić jeden parametr (w tym przykładzie jest to parametr `eventObject`). Określenie parametru funkcji przypomina deklarowanie zmiennej, a zatem również wymaga podania typu danych parametru. (W omawianym przykładzie typem parametru jest `EventType`).

Z każdym typem zdarzeń, które mogą być wykrywane, jest w środowisku ActionScript skojarzona odpowiednia klasa. Typ danych określony dla parametru funkcji jest zawsze klasą skojarzoną z typem zdarzeń, na które chcemy reagować. Na przykład zdarzenie `click` (wywoływane, gdy użytkownik kliknie myszą na elemencie) jest skojarzone z klasą `MouseEvent`. Aby napisać funkcję wykrywającą zdarzenie `click` (tzw. detektor tego zdarzenia), musimy zdefiniować funkcję wykrywającą z parametrem typu `MouseEvent`. Na koniec, między otwierającym a zamykającym nawiasem sześciennym (`{ ... }`), wpisujemy instrukcje, które komputer ma wykonać po zajściu zdarzenia.

W ten sposób została napisana funkcja obsługi zdarzeń. Następnie wymagana jest instrukcja dla obiektu zdarzenia źródłowego (jest to obiekt, dla którego wystąpiło zdarzenie, na przykład: przycisk), która spowoduje, że obiekt wywoła funkcję po zajściu zdarzenia. Funkcję należy zarejestrować w obiekcie zdarzenia źródłowego poprzez wywołanie metody `addEventListener()` tego obiektu (wszystkie obiekty zawierające zdarzenia zawierają również metodę `addEventListener()`). Metoda `addEventListener()` ma dwa parametry:

- Pierwszym parametrem jest nazwa zdarzenia, na które chcemy reagować. Każdy obiekt jest łączony z określoną klasą. Każda klasa zdarzenia zawiera specjalną wartość, która pełni rolę unikalnej nazwy zdefiniowanej dla każdego obiektu. Ta wartość jest używana dla pierwszego parametru.

- Drugim parametrem jest nazwa funkcji reagującej na zdarzenie. Należy zwrócić uwagę, że nazwa funkcji przekazywana jako parametr jest zapisana bez nawiasów.

Proces obsługi zdarzeń

Poniżej krok po kroku opisano proces zachodzący podczas tworzenia detektora zdarzeń. W omawianym przykładzie stworzymy funkcję wykrywającą (detektor) wywoływaną w odpowiedzi na kliknięcie obiektu `myButton`.

Prawdziwy kod napisany przez programistę wygląda następująco:

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

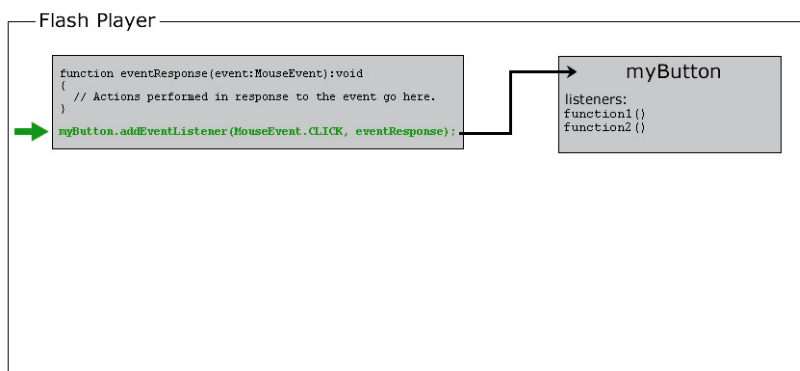
```
myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

A tak działałby ten kod:

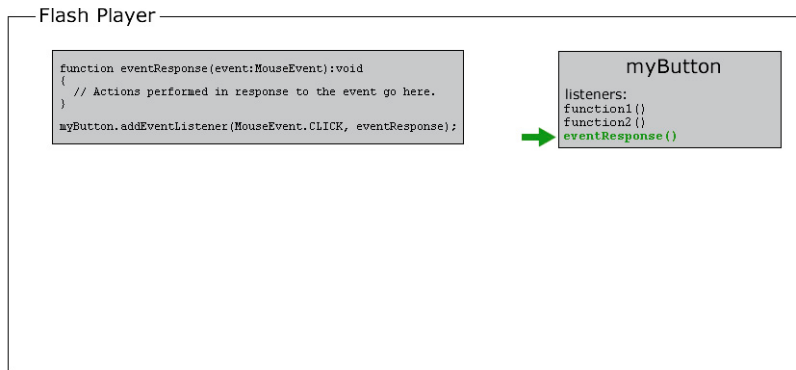
- 1 Po załadowaniu pliku SWF komputer odnotowuje fakt, że zawiera on funkcję o nazwie `eventResponse()`.



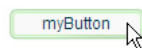
- 2 Następnie komputer uruchamia kod (przede wszystkim wiersze kodu, które nie należą do żadnej funkcji). W tym przypadku jest to tylko jeden wiersz kodu: wywołanie metody `addEventListener()` w obiekcie źródłowym (o nazwie `myButton`) i przekazanie funkcji `eventResponse` jako parametru.



Obiekt `myButton` zawiera listę funkcji-detektorów, które wykrywają poszczególne zdarzenia. Po wywołaniu metody `addEventListener()` obiekt `myButton` będzie zawierał funkcję `eventResponse()` na liście detektorów.

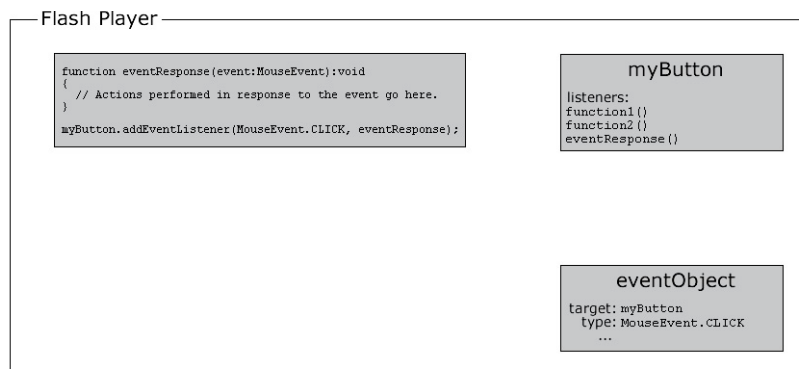


- 3 W pewnej chwili użytkownik klika przycisk `myButton`, wywołując jego zdarzenie `click` (w kodzie oznaczone `MouseEvent.CLICK`).

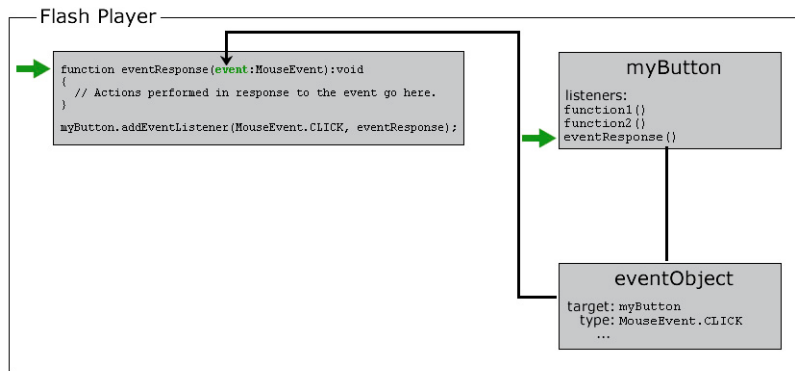


W tym momencie zachodzą następujące procesy:

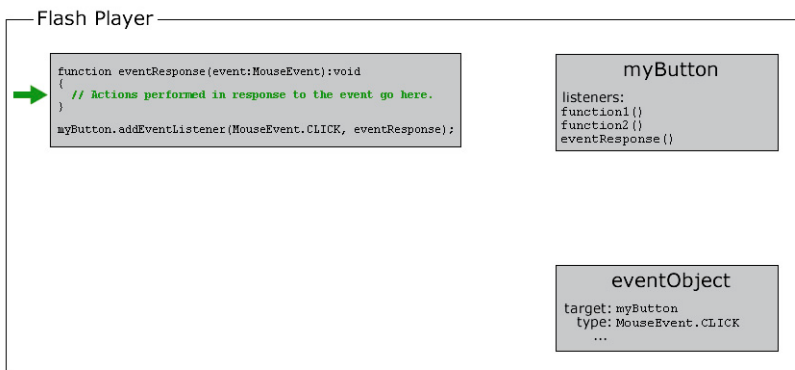
- a Następuje utworzenie obiektu będącego instancją klasy skojarzonej ze zdarzeniem (w tym przykładzie `MouseEvent`). W przypadku wielu zdarzeń ten obiekt jest instancją klasy `Event`: W przypadku zdarzeń myszy jest to instancja klasy `MouseEvent`. W przypadku innych zdarzeń jest to instancja klasy skojarzonej z konkretnym zdarzeniem. Utworzony obiekt nazywany jest *obiektem zdarzenia* i zawiera konkretne informacje o zdarzeniu, które miało miejsce: jakiego typu jest to zdarzenie, gdzie zaszło, a także inne informacje charakterystyczne dla zdarzenia.



- b Następnie komputer przeszukuje listę detektorów zdarzeń zapisaną w obiekcie `myButton`. Kolejno wywołuje funkcje wymienione na liście i przekazuje do każdej z nich obiekt zdarzenia jako parametr. Ponieważ funkcja `eventResponse()` jest jednym z detektorów zdarzeń obiektu `myButton`, w ramach tego procesu komputer wywołuje funkcję `eventResponse()`.



- c Po wywołaniu funkcji `eventResponse()` zawarty w niej kod realizuje określone operacje.



Przykłady obsługi zdarzeń

Poniżej przedstawiono kilka przykładów kodu przeznaczonego do obsługi zdarzeń. Te przykłady przedstawiają niektóre z najczęściej stosowanych elementów zdarzeń oraz możliwych zmian, jakie można wprowadzić podczas pisania kodu obsługi zdarzeń.

- Kliknięcie przycisku w celu rozpoczęcia odtwarzania klipu filmowego. W poniższym przykładzie `playButton` to nazwa instancji przycisku, a `this` to specjalna nazwa oznaczająca „bieżący obiekt”:

```
this.stop();

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

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

- Wykrywanie wpisywania w polu tekstowym. W tym przykładzie `entryText` to wejściowe pole tekstowe, a `outputText` to dynamiczne pole tekstowe:

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

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

- Przejście do adresu URL po kliknięciu przycisku. W tym przypadku `linkButton` to nazwa instancji przycisku:

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

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

Tworzenie instancji obiektów

Aby w języku ActionScript można było użyć obiektu, obiekt ten musi istnieć. Jednym z etapów tworzenia obiektu jest zadeklarowanie zmiennej; jednak deklaracja zmiennej rezerwuje jedynie puste miejsce w pamięci komputera. Zawsze należy przypisać zmiennej konkretną wartość (tzn. utworzyć obiekt i zapisać go w zmiennej) zanim spróbujemy użyć obiektu lub nim manipulować. Proces tworzenia obiektów jest określony jako *tworzenie instancji* klasy. Innymi słowy: należy utworzyć instancję konkretnej klasy.

Najprostszy sposób utworzenia instancji obiektu w ogóle nie wymaga programowania w języku ActionScript. W programie Flash Professional umieść symbol klipu filmowego, symbol przycisku lub pole tekstowe na stole montażowym i przypisz mu nazwę instancji. Program Flash Professional automatycznie zadeklaruje zmienną o nazwie instancji, utworzy instancję i umieści obiekt w tej zmiennej. Podobnie w programie Flex — składnik tworzony jest w kodzie MXML poprzez zdefiniowanie znacznika MXML lub umieszczenie składnika w edytorze w trybie projektowania programu Flash Builder. Po przypisaniu identyfikatora do składnika ten identyfikator staje się nazwą zmiennej ActionScript zawierającej instancję tego składnika.

Jednak nie zawsze konieczne jest tworzenie obiektu w trybie ekranowym, a w przypadku obiektów niewidocznych nie jest to możliwe. Istnieje również kilka dodatkowych sposobów tworzenia obiektów w kodzie ActionScript.

Niektóre typy danych języka ActionScript pozwalają na tworzenie instancji poprzez wpisanie *wyrażenia literalnego* — tj. wartości wpisanej wprost w kodzie ActionScript. Oto kilka przykładów:

- Literalna wartość liczbowa (liczby wprowadza się bezpośrednio):

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

- Literalny ciąg znaków (tekst należy ująć w cudzysłów):

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

- Literalna wartość logiczna (należy wpisać literał `true` albo `false`):

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

- Literalna wartość tablicy (listę wartości oddzielanych przecinkami należy ująć w nawiasy kwadratowe):

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

- Literalna wartość XML (kod XML należy wpisać bezpośrednio):

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

W języku ActionScript dozwolone są także wyrażenia literalne typu Array, RegExp, Object i Function.

Najczęściej stosowanym sposobem tworzenia instancji dla dowolnego typu danych jest użycie operatora `new` z nazwą klasy, co przedstawiono poniżej:

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

Tworzenie obiektu przy użyciu operatora `new` jest często nazywane „wywołaniem konstruktora klasy”. *Konstruktor* to specjalna metoda wywoływana w procesie tworzenia instancji klasy. W przypadku tworzenia instancji w ten sposób nawiasy należy umieścić za nazwą klasy. Czasami wartości parametrów również określa się w nawiasach. Istnieją dwie czynności, jakie należy wykonać w przypadku wywoływania metody.

Operatora `new` można używać nawet do tworzenia instancji obiektów należących do typów danych, które dopuszczają tworzenie instancji na podstawie wyrażen literalowych. Na przykład poniższe dwa wiersze kodu są równoważne:

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

Tworzenie obiektów przy użyciu konstrukcji `new NazwaKlasy()` jest bardzo ważnym aspektem programowania w języku ActionScript. Dla wielu typów danych języka ActionScript nie istnieje reprezentacja ekranowa. Dlatego nie można ich tworzyć poprzez umieszczenie elementu na stole montażowym programu Flash Professional ani w trybie projektowania edytora MXML programu Flash Builder. Instancje danych tego typu można tworzyć w języku ActionScript wyłącznie za pomocą operatora `new`.

Adobe Flash Professional

W programie Flash Professional operator `new` pozwala także na utworzenie instancji symbolu klipu filmowego, który jest zdefiniowany w bibliotece, ale nie został umieszczony na stole montażowym.

Więcej tematów Pomocy

[Praca z tablicami](#)

[Korzystanie z wyrażeń regularnych](#)

„Typ danych Object” na stronie 57

[Tworzenie obiektów MovieClip za pomocą kodu ActionScript](#)

Typowe elementy programu

Istnieje kilka dodatkowych elementów wykorzystywanych do tworzenia programów w kodzie ActionScript.

Operatory

Operator to specjalne symbole (a niekiedy słowa) służące do wykonywania obliczeń. Najczęściej używane są do wykonywania operacji matematycznych oraz do porównywania wartości. Z reguły operator działa na jednej lub kilku wartościach i generuje pojedynczy wynik. Na przykład:

- Operator dodawania (+) dodaje dwie liczby i daje w wyniku jedną liczbę:

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

- Operator mnożenia (*) mnoży dwie liczby i daje w wyniku jedną liczbę:

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

- Operator równości (==) sprawdza, czy dwie wartości są równe, i daje w wyniku jedną wartość prawda albo fałsz (Boolean):

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

Tak jak ilustrują to przykłady, operator równości oraz pozostałe operatory „porównań” najczęściej używane są z instrukcją `if` w celu ustalenia, czy określone instrukcje są wykonywane, czy nie.

Więcej tematów Pomocy

„[Operator](#)” na stronie 67

Komentarze

Podczas pisania kodu w języku ActionScript możliwe jest również pisanie komentarzy. Na przykład: czasami wymagane jest wyjaśnienie działania konkretnych wierszy kodu lub wyjaśnienie przyczyn podjęcia określonej decyzji. *Komentarze do kodu* to narzędzie umożliwiające dopisywanie w programie tekstów ignorowanych przez komputer. W języku ActionScript dostępne są dwa rodzaje komentarzy:

- Komentarze jednowierszowe: komentarz jednowierszowy rozpoczyna się od dwóch ukośników umieszczonych w dowolnym miejscu w wierszu. Komputer ignoruje wszystko za ukośnikami aż do końca wiersza:

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

- Komentarze wielowierszowe: komentarz wielowierszowy składa się ze znacznika początku (/*), treści komentarza i znacznika końca (*/). Komputer ignoruje wszystko między znacznikami początku i końca bez względu na to, ile wierszy obejmuje taki komentarz:

```
/*  
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.  
*/
```

Innym częstym zastosowaniem komentarzy jest tymczasowe „wyłączenie” jednego lub większej liczby wierszy kodu. Na przykład: można użyć znaków komentarza w celu przetestowania innego sposobu wykonania pewnych operacji. Znaki komentarza można również wykorzystać w celu określenia, dlaczego fragmenty kodu ActionScript nie działają zgodnie z oczekiwaniami.

Sterowanie przepływem wykonania

Często zdarza się, że pewne operacje w programie powinny być powtarzane, wykonywane lub nie w zależności od spełnienia określonych warunków itd. *Sterowanie przepływem wykonania* polega właśnie na podejmowaniu decyzji o wykonywaniu określonych operacji. Język ActionScript udostępnia kilka elementów sterujących przepływem wykonania.

- **Funkcje:** funkcje przypominają skróty. Pełnią rolę skrótów — umożliwiają zgrupowanie szeregu operacji pod jedną nazwą, mogą być także używane do wykonywania obliczeń. Funkcje są potrzebne przy obsłudze zdarzeń, ale stosuje się je także jako ogólne narzędzie do grupowania szeregu instrukcji.
- **Pętle:** struktura pętli umożliwia wyznaczenie zbioru instrukcji, które komputer wykonuje określoną liczbę razy lub tyle razy, aż zostanie spełniony określony warunek. Pętle są często używane do manipulowania kilkoma elementami pokrewnymi przy użyciu zmiennej, której wartość zmienia się przy każdym wykonaniu pętli.
- **Instrukcje warunkowe:** jest to sposób określania instrukcji, które są wykonywane tylko po spełnieniu określonych warunków. Służą również do udostępniania alternatywnych zestawów instrukcji dla różnych warunków. Najczęściej używaną instrukcją warunkową jest instrukcja `if`. Instrukcja `if` sprawdza wartość lub wyrażenie ujęte w nawiasy. Jeśli wartością jest `true`, wówczas wykonywane są wiersze kodu w nawiasach sześciennych. W przeciwnym wypadku są one ignorowane. Na przykład:

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

Instrukcja `else`, używana razem z instrukcją `if`, umożliwia wskazanie alternatywnych instrukcji, które komputer ma być wykonać, gdy warunek nie jest spełniony (nie ma wartości `true`):

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

Więcej tematów Pomocy

„[Funkcje](#)” na stronie 77

„[Pętle](#)” na stronie 75

„[Instrukcje warunkowe](#)” na stronie 73

Przykład: Animowana prezentacja prac (Flash Professional)

Przejdźmy teraz do przykładu, który będzie pierwszą okazją do zapoznania się z różnymi elementami języka ActionScript połączonymi w kompletną aplikację. Animowana prezentacja prac jest przykładem wykorzystania istniejącej animacji liniowej oraz dodawania mniejszych elementów interaktywnych. Na przykład: animację utworzoną dla klienta można wprowadzić do prezentacji online. Elementy interaktywne dodawane do prezentacji obejmują dwa przyciski dla użytkownika: przycisk służący do uruchamiania animacji i przycisk powodujący przejście pod inny adres URL (np. do menu prezentacji lub na stronę główną autora).

Proces tworzenia aplikacji można podzielić na następujące główne etapy:

- 1 Przygotowanie pliku FLA do uzupełnienia o kod ActionScript i elementy interaktywne.
- 2 Utworzenie i dodanie przycisków.
- 3 Napisanie kodu w języku ActionScript.
- 4 Testowanie aplikacji.

Przygotowanie do dodania elementów interaktywnych

Zanim dodamy do naszej animacji elementy interaktywne, wskazane będzie przygotowanie pliku FLA poprzez utworzenie miejsc przeznaczonych na dodanie nowej treści. To zadanie obejmuje utworzenie miejsc na stole montażowym, w których zostaną umieszczone przyciski. Konieczne jest również utworzenie „miejsca” w pliku FLA w celu przechowywania różnych elementów osobno.

Aby przygotować plik FLA na dodanie elementów interaktywnych:

- 1 Utwórz plik FLA z przykładową animacją, taką jak pojedyncza animacja ruchu lub animacja kształtu. Jeśli istnieje już plik FLA zawierający animację prezentowaną w projekcie, otwórz ten plik i zapisz go z nową nazwą.
- 2 Wybierz na ekranie miejsce dla dwóch przycisków. Jeden przycisk służy do uruchamiania animacji, a drugi jest łączem do prezentacji prac lub strony głównej. W razie potrzeby wyczyść lub dodaj przestrzeń na nowe elementy na stole montażowym. Jeśli animacja nie zawiera ekranu startowego, można go utworzyć w pierwszej klatce. W takim przypadku wymagane będzie przesunięcie animacji w taki sposób, aby zaczynała się od klatki 2 lub dalej.
- 3 Dodaj nową warstwę nad pozostałymi warstwami na osi czasu i nadaj jej nazwę **buttons**. Na tej warstwie zostaną dodane przyciski.
- 4 Nad warstwą buttons dodaj nową warstwę i nadaj jej nazwę **actions**. Na tej warstwie można dodawać kod ActionScript do swojej aplikacji.

Tworzenie i dodawanie przycisków

Następnie zostaną utworzone i ustawione przyciski, które będą stanowiły istotną część tej interaktywnej aplikacji.

Aby utworzyć i dodać przyciski w pliku FLA:

- 1 Korzystając z narzędzi rysunkowych, stwórz formę graficzną pierwszego przycisku (przycisku odtwarzania) na warstwie buttons. Np. narysuj poziomy owal z nałożonym tekstem.
- 2 Za pomocą narzędzia Zaznaczanie zaznacz wszystkie elementy graficzne jednego przycisku.
- 3 Z menu głównego wybierz polecenie Modyfikuj > Konwertuj na symbol.
- 4 W konie dialogowym jako typ symbolu wybierz Przycisk, nadaj symbolowi nazwę i kliknij przycisk OK.
- 5 Po zaznaczeniu przycisku nadaj mu nazwę instancji **playButton** w Inspektorze właściwości.
- 6 Powtórz kroki od 1 do 5, tworząc przycisk przenoszący użytkownika na stronę główną autora. Nadaj temu przyciskowi nazwę **homeButton**.

Pisanie kodu

Kod ActionScript tej aplikacji można podzielić na trzy grupy funkcji, mimo że będzie w całości wprowadzany w jednym miejscu. Trzy operacje, jakie wykonuje kod:

- Zatrzymanie głowicy odtwarzania natychmiast po załadowaniu pliku SWF (gdy głowica odtwarzania znajdzie się w klatce nr 1).

- Wykrywanie zdarzenia w celu rozpoczęcia odtwarzania pliku SWF, gdy użytkownik kliknie przycisk odtwarzania.
- Wykrywanie zdarzenia w celu przejścia pod odpowiedni adres URL, gdy użytkownik kliknie przycisk strony głównej autora.

Aby utworzyć kod zatrzymujący głowicę odtwarzania, gdy znajdzie się ona w klatce nr 1:

- 1 Zaznacz klatkę kluczową pokrywającą się z klatką nr 1 na warstwie actions.
- 2 Wybierz polecenie Okno > Operacje z menu głównego, aby otworzyć panel Operacje.
- 3 W panelu Skrypt wprowadź następujący kod:

```
stop();
```

Aby napisać kod rozpoczynający animację po kliknięciu przycisku odtwarzania:

- 1 Na końcu kodu wprowadzonego w poprzednich krokach dodaj dwa puste wiersze.
- 2 Wprowadź następujący kod na końcu skryptu:

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

Ten kod zawiera definicję funkcji o nazwie `startMovie()`. Wywołanie funkcji `startMovie()` powoduje rozpoczęcie odtwarzania głównej osi czasu.

- 3 W wierszu następującym po kodzie dodanym w poprzednim kroku wprowadź następujący wiersz kodu:

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

Ten wiersz kodu rejestruje funkcję `startMovie()` jako detektor zdarzenia `click` obiektu `playButton`. Innymi słowy powoduje, że za każdym razem, gdy użytkownik kliknie przycisk `playButton`, wywoływana będzie funkcja `startMovie()`.

Aby napisać kod powodujący przejście pod adres URL po kliknięciu przycisk strony głównej:

- 1 Na końcu kodu wprowadzonego w poprzednich krokach dodaj dwa puste wiersze.
- 2 Wprowadź ten kod na końcu skryptu:

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

Ten kod zawiera definicję funkcji o nazwie `gotoAuthorPage()`. Ta funkcja najpierw powoduje utworzenie instancji `URLRequest`, która reprezentuje adres URL `http://example.com/`. Następnie przekazuje ten adres URL do funkcji `navigateToURL()`, co powoduje otwarcie tego adresu URL w przeglądarce.

- 3 W wierszu następującym po kodzie dodanym w poprzednim kroku wprowadź następujący wiersz kodu:

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

Ten wiersz kodu rejestruje funkcję `gotoAuthorPage()` jako detektor zdarzenia `click` obiektu `homeButton`. Innymi słowy powoduje, że za każdym razem, gdy użytkownik kliknie przycisk `homeButton`, wywoływana będzie funkcja `gotoAuthorPage()`.

Testowanie aplikacji

Aplikacja jest już całkowicie gotowa do użytku. Przetestujmy ją, aby przekonać się, czy tak jest w istocie.

Aby przetestować aplikację:

- 1 Z menu głównego wybierz polecenie Sterowanie > Testuj film. Program Flash Professional utworzy plik SWF i otworzy go w oknie odtwarzacza Flash Player.
- 2 Wypróbuj oba przyciski, aby sprawdzić, czy działają zgodnie z oczekiwaniami.
- 3 Oto kilka sugestii na wypadek, gdyby przyciski nie działały:
 - Czy przyciski mają różne nazwy instancji?
 - Czy w wywołaniach metody `addEventListener()` użyto tych samych nazw, co nazwy instancji przycisków?
 - Czy w wywołaniach metody `addEventListener()` użyto prawidłowych nazw zdarzeń.
 - Czy dla każdej z funkcji określono prawidłowy parametr? (Obie metody potrzebują jednego parametru typu `MouseEvent`).

Wszystkie te pomyłki oraz większość innych pomyłek powoduje wyświetlenie komunikatu o błędzie. Komunikat o błędzie może się pojawić po wybraniu komendy Testuj film lub po kliknięciu przycisku podczas testowania projektu. Błędy kompilatora (takie, które występują przy pierwszym wykonaniu polecenia Testuj film) są przedstawione w panelu Błędy kompilatora. W panelu Wyjście przedstawione są błędy powstające w czasie wykonywania, które pojawiają się podczas odtwarzania treści — na przykład po kliknięciu przycisku.

Budowanie aplikacji w języku ActionScript

Do zbudowania aplikacji w języku ActionScript nie wystarczy znajomość składni i nazw klas, które będą używane. Większość dokumentacji platformy Flash opisuje te dwa zagadnienia (składnia i korzystanie z klas ActionScript). Jednak w celu zbudowania aplikacji w języku ActionScript wymagana jest również następująca wiedza:

- Jakie programy mogą być używane do pisanie w języku ActionScript?
- W jaki sposób można porządkować kod ActionScript?
- W jaki sposób dołączyć kod ActionScript do aplikacji?
- Jakie są etapy tworzenia aplikacji w języku ActionScript?

Warianty organizacji kodu

Język ActionScript 3.0 pozwala na realizację rozmaitych operacji — od prostej animacji grafiki po złożone przetwarzanie transakcji w modelu klient-serwer. W zależności od rodzaju budowanej aplikacji można użyć różnych sposobów uwzględniania kodu ActionScript w projekcie.

Zapisywanie kodu w klatkach na osi czasu w programie Flash Professional

W programie Flash Professional możliwe jest dodawanie kodu ActionScript do dowolnych klatek na osi czasu. Kod ten będzie wykonywany w trakcie odtwarzania filmu, gdy głowica odtwarzania znajdzie się w danej klatce.

Umieszczenie kodu ActionScript w klatkach pozwala na prostą rozbudowę funkcjonalności aplikacji zbudowanych w programie Flash Professional. Możliwe jest dodawanie kodu do dowolnych klatek na głównej osi czasu lub do dowolnych klatek na osi czasu każdego symbolu typu MovieClip. Jednak ta swoboda ma swoją cenę. W większych aplikacjach nietrudno stracić rozeznanie, które klatki zawierają poszczególne skrypty. W rezultacie ta skomplikowana struktura może utrudnić modyfikowanie i rozbudowywanie aplikacji.

Wielu programistów decyduje się na uproszczenie organizacji kodu ActionScript i umieszcza kod tylko w pierwszej klatce osi czasu lub na konkretnej warstwie dokumentu Flash. Oddzielenie kodu ułatwi wyszukiwanie i modyfikowanie kodu w plikach FLA programu Flash. Jednak użycie tego samego kodu w innym projekcie Flash Professional nie będzie możliwe, jeśli nie zostanie on skopiowany i wklejony do nowego pliku.

Aby ułatwić stosowanie kodu ActionScript w innych projektach Flash Professional, można zapisać kod w zewnętrznych plikach ActionScript (są to pliki tekstowe o rozszerzeniu .as).

Osadzanie kodu w plikach Flex MXML

W środowisku programistycznym Flex, takim jak Flash Builder, można umieścić kod ActionScript między znacznikami `<fx:Script>` w pliku MXML Flex. Jednak może to spowodować zwiększenie złożoności dużych projektów i utrudnić korzystanie z kodu w innych projektach Flex. Korzystanie z kodu ActionScript w innych projektach Flex w przyszłości będzie łatwiejsze, jeśli kod zostanie zapisany w zewnętrznych plikach ActionScript.

Uwaga: W tym celu można określić parametr dla znacznika `<fx:Script>`. Za pomocą parametru `source` można „zaimportować” kod ActionScript z pliku zewnętrznego w taki sposób, jakby został on wpisany bezpośrednio do znacznika `<fx:Script>`. Jednak używany plik źródłowy nie może definiować własnej klasy, co ogranicza możliwość stosowania tego samego pliku w różnych projektach.

Zapisywanie kodu w plikach ActionScript

Jeśli projekt zawiera obszerny kod w języku ActionScript, najlepszym sposobem na zorganizowanie tego kodu będzie umieszczenie go w odrębnych plikach źródłowych ActionScript (plikach tekstowych z rozszerzeniem AS). Plik ActionScript może mieć jedną z dwóch struktur, w zależności od planowanego sposobu wykorzystania go w aplikacji.

- Nieustrukturyzowany kod ActionScript: wiersze kodu ActionScript, w tym definicje instrukcji lub funkcji, są zapisane tak, jak gdyby zostały wprowadzone bezpośrednio do skryptu osi czasu lub pliku MXML.

Zapisany w ten sposób kod ActionScript można włączyć do projektu za pomocą instrukcji `include` języka ActionScript lub znacznika `<fx:Script>` w pliku Flex MXML. Instrukcja `include` ActionScript informuje kompilator o konieczności uwzględnienia treści zewnętrznego pliku ActionScript w wyznaczonym miejscu oraz w określonym zakresie w skrypcie. Rezultat jest taki sam, jakby kod został wprowadzony bezpośrednio. W języku MXML użycie znacznika `<fx:Script>` z atrybutem `source` jest sposobem na wskazanie zewnętrznego kodu ActionScript, który jest ładowany przez kompilator w odpowiednim momencie w aplikacji. Na przykład poniższy znacznik spowoduje załadowanie zewnętrznego pliku ActionScript o nazwie `Box.as`:

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

- Definicja klasy ActionScript: definicja klasy w języku ActionScript zawierająca definicje metod i właściwości.

Zdefiniowanie klasy umożliwia uzyskanie dostępu do kodu ActionScript w klasie poprzez utworzenie instancji klasy oraz z wykorzystaniem jej właściwości, metod i zdarzeń. Korzystanie z własnych klas przebiega tak samo, jak korzystanie z dowolnych wbudowanych klas ActionScript, i wymaga dwóch etapów:

- Użycia instrukcji `import` określającej pełną nazwę klasy, tak aby kompilator języka ActionScript był w stanie ją znaleźć. Na przykład: w celu użycia klasy `MovieClip` w kodzie ActionScript należy zaimportować klasę - wprowadzając jej pełną nazwę - wraz z pakietem i klasą:

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

Możemy też zaimportować pakiet zawierający klasę `MovieClip`, co jest równoważne użyciu odrębnych instrukcji `import` dla poszczególnych klas w pakiecie:

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

Klasy najwyższego poziomu stanowią jedyny wyjątek od reguły, która głosi, że klasa musi zostać zaimportowana, aby możliwe było jej użycie w kodzie. Te klasy nie są zdefiniowane w pakiecie.

- Napisz kod, w którym znajdzie się nazwa klasy. Na przykład: zadeklaruj zmienną, której typem danych będzie konkretna klasa, a następnie utwórz instancję klasy, która zostanie umieszczona w zmiennej. Użycie klasy w języku ActionScript nakazuje kompilatorowi załadowanie definicji tej klasy. Na przykład: w przypadku klasy zewnętrznej o nazwie Box ta instrukcja tworzy instancję klasy Box:

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

Gdy kompilator po raz pierwszy napotka odniesienie do klasy Box, wyszuka dostępny kod źródłowy w celu odnalezienia definicji klasy Box.

Wybór właściwego narzędzia

W celu pisania i edycji kodu ActionScript można wykorzystać jedno z kilku narzędzi (lub wiele narzędzi jednocześnie).

Flash Builder

Adobe Flash Builder to idealne narzędzie przeznaczone do tworzenia projektów w środowisku Flex oraz projektów, które zawierają głównie kod ActionScript. Flash Builder zawiera również w pełni wyposażony edytor ActionScript, a także funkcje do definiowania układu oraz edycji kodu MXML. Ten program może być używany do tworzenia projektów wyłącznie w środowisku Flex albo w kodzie ActionScript. Środowisko Flex ma wiele zalet, takich jak bogata gama gotowych elementów sterujących, elastyczne elementy do dynamicznego sterowania układem i wbudowane mechanizmy współpracy ze zdalnymi danymi oraz łączenia danych zewnętrznych z elementami interfejsu użytkownika. Jednak korzystanie z tych funkcji wymaga wprowadzenia dodatkowego kodu, dlatego pliki SWF projektów wykorzystujących środowisko Flex mogą być większe niż w przypadku ich odpowiedników nieutworzonych w środowisku Flex.

Z programu Flash Builder należy korzystać w przypadku tworzenia kompletnych, sterowanych danymi aplikacji RIA w środowisku Flex. Program może służyć do edycji kodu ActionScript, edycji kodu MXML oraz do definiowania układu aplikacji — wszystkie te czynności można wykonać w jednym narzędziu.

Wielu użytkowników programu Flash Professional, którzy tworzą rozbudowane projekty w kodzie ActionScript, korzysta z tego programu w celu tworzenia elementów wyświetlanych, a program Flash Builder jest wówczas stosowany jako edytor kodu ActionScript.

Flash Professional

Program Flash Professional — oprócz funkcji przeznaczonych do tworzenia grafiki i animacji — udostępnia narzędzia przeznaczone do pracy z kodem ActionScript. Kod może zostać dołączony do elementów w pliku FLA lub może znajdować się tylko w zewnętrznych plikach ActionScript. Flash Professional jest narzędziem idealnym w przypadku projektów zawierających rozbudowane animacje lub wideo. Jest bardzo wartościowy, gdy użytkownik samodzielnie tworzy większość elementów graficznych. Inną przyczyną korzystania z programu Flash Professional w celu programowania w języku ActionScript jest możliwość tworzenia elementów graficznych oraz pisania kodu w tej samej aplikacji. Program Flash Professional zawiera również wstępnie wbudowane składniki interfejsu użytkownika. Korzystając z tych składników można tworzyć mniejsze pliki SWF, a za pomocą narzędzi graficznych można dostosowywać te składniki do stylu konkretnego projektu.

Program Flash Professional zawiera dwa narzędzia do pisania kodu w języku ActionScript:

- Panel Operacje: jest dostępny podczas pracy w pliku FLA i umożliwia pisanie kodu ActionScript powiązanego z klatkami na osi czasu.
- Okno Skrypt: jest to odrębny edytor tekstu przeznaczony specjalnie do pracy z plikami ActionScript (AS).

Edytor kodu ActionScript dostarczony przez inną firmę

Ponieważ pliki ActionScript (z rozszerzeniem AS) mają postać prostych plików tekstowych, każdy program, który może edytować zwykłe pliki tekstowe umożliwia także pisanie plików ActionScript. Obok oferowanych przez firmę Adobe produktów do programowania w języku ActionScript dostępnych jest kilka edytorów tekstu opracowanych przez inne firmy i wyposażonych w funkcje specyficzne dla języka ActionScript. Treść pliku MXML lub klasy ActionScript można zapisać w dowolnym edytorze tekstu. Następnie za pomocą pakietu Flex SDK można utworzyć z tych plików aplikację. Projekt może wykorzystywać środowisko Flex lub tylko kod ActionScript. Niektórzy programiści korzystają z programu Flash Builder lub edytorów kodu ActionScript innych firm — w celu pisania klas ActionScript, a z programu Flash Professional w celu tworzenia treści graficznych.

Przyczyny korzystania z edytora ActionScript innej firmy:

- Użytkownik pisze kod ActionScript w osobnym programie, a elementy graficzne tworzy w programie Flash Professional.
- Używa aplikacji do programowania w technologiach innych niż ActionScript (np. do tworzenia stron HTML lub budowania aplikacji w innym języku programowania) i chce używać tej samej aplikacji do pisania programów w języku ActionScript.
- Tworzone są obiekty zawierające tylko kod ActionScript lub projekty w środowisku Flex przy wykorzystaniu pakietu Flex SDK bez programów Flash Professional i Flash Builder.

Oto niektóre warte wzmianki edytory kodu, które oferują funkcje specyficzne dla języka ActionScript:

- [Adobe Dreamweaver® CS4](#)
- [ASDT](#)
- [FDT](#)
- [FlashDevelop](#)
- [PrimalScript](#)
- [SE|PY](#)
- [TextMate](#) (oraz [Pakiety ActionScript i Flex](#))

Proces tworzenia aplikacji w języku ActionScript

Niezależnie od tego, czy projekt ActionScript jest duży, czy niewielki, zastosowanie procesu projektowania i programowania aplikacji sprawia, że praca przebiega bardziej wydajnie. Poniższe kroki składają się na podstawowy proces budowania aplikacji, w której wykorzystywany jest język ActionScript 3.0:

1 Projektowanie aplikacji.

Przed przystąpieniem do budowania aplikacji należy ją opisać.

2 Tworzenie kodu w języku ActionScript 3.0.

Do tworzenia kodu w języku ActionScript można użyć programów Flash Professional, Flash Builder, Dreamweaver lub edytora tekstu.

3 Utworzenie projektu Flash lub Flex umożliwiającego wykonanie kodu.

W programie Flash Professional utwórz plik FLA, skonfiguruj ustawienia publikowania, dodaj do aplikacji składniki interfejsu użytkownika, a następnie utwórz odniesienie do kodu ActionScript. W środowisku Flex zdefiniuj aplikację, dodaj składniki interfejsu użytkownika za pomocą kodu MXML, a następnie utwórz odniesienie do kodu ActionScript

4 Opublikowanie i przetestowanie aplikacji.

Testowanie aplikacji obejmuje uruchomienie aplikacji ze środowiska programistycznego oraz sprawdzenie, czy działa zgodnie z oczekiwaniami.

Kroki te nie muszą być wykonywane w kolejności, w jakiej zostały tu wymienione. Do kolejnych kroków można też przechodzić przed ukończeniem poprzednich. Możemy na przykład zaprojektować jeden ekran aplikacji (krok 1), utworzyć grafikę, przyciski itd. (krok 3), dopiero potem napisać kod ActionScript (krok 2) i przetestować aplikację (krok 4). Możliwe jest także projektowanie kolejnych części aplikacji, dodawanie po jednym przycisku lub elemencie interfejsu i pisanie kodu ActionScript dla każdego nowego elementu, a także testowanie aplikacji w miarę, jak powstaje. Zapamiętanie tych czterech kroków procesu programowania może być pomocne. Jednak w rzeczywistym programowaniu bardziej efektywne jest wykonywanie kolejnych kroków, a w razie potrzeby wracanie do poprzednich.

Tworzenie własnych klas

Proces tworzenia klas przeznaczonych do użytku we własnych projektach może wydawać się żmudny. Jednak w istocie najtrudniejszym etapem tworzenia klasy jest jej zaprojektowanie — czyli wybór jej metod, właściwości i zdarzeń.

Strategie projektowania klas

Projektowanie zorientowane obiektowo jest złożonym zagadnieniem; dziedzina ta stanowi przedmiot pracy zawodowej i badań naukowych. Możliwe jest jednak wskazanie kilku rekomendacji, które znacznie ułatwią zmierzenie się z tym zagadnieniem.

- 1 Zastanów się, jaką rolę w aplikacji odgrywają instancje tej klasy. Z reguły obiekty pełnią jedną z trzech ról:
 - Obiekty wartości: te obiekty pełnią głównie rolę kontenerów danych. Mogą zawierać kilka właściwości i niewielką liczbę metod (lub zero metod). Zwykle reprezentują one w kodzie wyrażenie zdefiniowane elementy. Na przykład: aplikacja odtwarzacza muzyki może zawierać klasę `Song`, która będzie reprezentować pojedynczy rzeczywisty utwór; a także klasę `Playlist`, która będzie reprezentować grupę utworów.
 - Obiekty wyświetlane: są to obiekty widoczne na ekranie. Należą do nich na przykład elementy interfejsu użytkownika, takie jak listy rozwijane, wskaźniki stanu, elementy graficzne, np. postacie w grze wideo, itd.
 - Obiekty struktury aplikacji: pełnią rozmaite role pomocnicze w strukturze logiki i przetwarzania aplikacji. Na przykład: można utworzyć obiekt, który będzie wykonywał określone obliczenia w ramach symulacji biologicznej. Możliwe jest utworzenie obiektu, który będzie odpowiedzialny za synchronizowanie wartości między pokrętkiem, a polem odczytu głośności w aplikacji odtwarzacza muzyki. Inny obiekt może zarządzać regułami w grze wideo. Możliwe jest również utworzenie klasy, która będzie ładowała zapisany obraz do aplikacji przeznaczonej do rysowania.
- 2 Należy określić konkretne funkcje, jakie będzie pełniła klasa. Różne funkcje realizowane przez klasę często kodowane są w postaci metod tej klasy.
- 3 Jeśli klasa ma pełnić rolę obiektu wartości, zdecyduj, jakie dane mają zawierać instancje tej klasy. Te dane prawdopodobnie znajdą odzwierciedlenie w postaci właściwości.
- 4 Ponieważ klasa projektowana jest pod kątem konkretnego projektu, powinna realizować funkcje potrzebne w danej aplikacji. Należy udzielić odpowiedzi na następujące pytania:
 - Jakie elementy informacji są przechowywane, rejestrowane i modyfikowane w aplikacji? Udzielenie odpowiedzi na to pytanie ułatwi określenie wymaganych obiektów wartości i właściwości.
 - Jakie operacje wykonuje aplikacja? Na przykład: co dzieje się po załadowaniu aplikacji, po kliknięciu konkretnego przycisku, po uruchomieniu filmu itp. Tak zidentyfikowane operacje mogą stać się metodami. Mogą być również właściwościami, jeśli „operacje” obejmują zmiany wartości.

- Jakie są informacje wymagane do wykonania poszczególnych operacji? Te informacje staną się później parametrami metod.
 - O jakich zmianach zachodzących w klasie aplikacja musi „wiedzieć” na różnych etapach wykonywania? Tak zidentyfikowane zmiany staną się później zdarzeniami.
- 5 Czy istnieje obiekt podobny do obiektu wymaganego, ale różni się od niego tym, że nie zawiera dodatkowych funkcji, które użytkownik chce dodać? Należy rozważyć utworzenie podklasy. (*Podklasa* to klasa, która wykorzystuje funkcje istniejącej klasy, a nie definiuje własnych funkcji). Na przykład: w celu utworzenia klasy, która jest obiektem ekranowym, można wykorzystać zachowanie istniejącego obiektu ekranowego jako podstawę nowej klasy. W takim przypadku obiekt ekranowy (taki jak MovieClip lub Sprite) będzie *klasą bazową*, a nowa klasa będzie rozszerzała istniejącą klasę.

Więcej tematów Pomocy

„[Dziedziczenie](#)” na stronie 106

Pisanie kodu klasy

Po zaprojektowaniu klasy — lub przynajmniej po zaplanowaniu, jakie informacje klasa będzie przechowywać oraz jakie operacje będzie wykonywać — napisanie składni klasy jest dosyć proste.

Oto minimalny zestaw czynności, jakie należy wykonać, aby utworzyć własną klasę w języku ActionScript:

- 1 Otwórz dokument tekstowy w edytorze kodu ActionScript.
- 2 Wpisz instrukcję `class`, aby zdefiniować nazwę klasy. Aby dodać instrukcję `class`, wprowadź słowa `public class`, a następnie nazwę klasy. Dodaj otwierające i zamykające nawiasy sześciennic, które będą zawierały treść klasy (definicje metody i właściwości). Na przykład:

```
public class MyClass
{
}
```

Słowo `public` oznacza, że klasa będzie dostępna z dowolnego miejsca kodu. Alternatywne warianty dostępności klasy opisano w sekcji „[Atrybuty przestrzeni nazw sterujące dostępem](#)” na stronie 92.

- 3 Wpisz instrukcję `package` w celu wskazania nazwy pakietu, który zawiera klasę. Składnia jest następująca: słowo `package`, a za nim pełna nazwa pakietu, po której występują otwierające i zamykające nawiasy klamrowe otaczające blok instrukcji `class`). Przykład: zmień kod z poprzedniego kroku na poniższy:

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

- 4 W treści klasy zdefiniuj wszystkie jej właściwości, używając instrukcji `var`; obowiązuje taka sama składnia, jak przy deklarowaniu zmiennych (z dodatkowym modyfikatorem `public`). Na przykład: dodanie tych wierszy między otwierające i zamykające nawiasy klamrowe definicji klasy powoduje utworzenie właściwości o nazwach `textProperty`, `numericProperty` i `dateProperty`:

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

5 Zdefiniuj wszystkie metody klas. Obowiązuje ta sama składnia, co przy definiowaniu funkcji. Na przykład:

- Aby utworzyć metodę `myMethod()`, wprowadź:

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

- Aby utworzyć konstruktor (specjalną metodę wywoływaną w procesie tworzenia instancji klasy), utwórz metodę o nazwie identycznej z nazwą klasy:

```
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);
}
```

Jeśli do klasy nie zostanie dołączona metoda konstruktora, kompilator automatycznie utworzy pusty konstruktor dla klasy. (Innymi słowy: konstruktor bez parametrów i bez instrukcji).

Istnieje kilka dodatkowych elementów klasy, które można zdefiniować. Te elementy są bardziej złożone.

- *Akcesory* to elementy stojące w pół drogi między metodą a właściwością. W definicji klasy akcesor zapisany jest tak, jak metoda. Umożliwia zatem wykonywanie wielu operacji, a nie tylko odczytywanie lub przypisywanie wartości, tak jak w przypadku właściwości. Jednak po utworzeniu instancji klasy akcesor można traktować tak, jak właściwość — tj. odczytywać i przypisywać wartości, odwołując się tylko do jego nazwy.
- Język ActionScript nie oferuje specjalnej składni służącej do definiowania zdarzeń. Zamiast tego należy zdefiniować zdarzenia w klasie za pomocą funkcji klasy `EventDispatcher`.

Więcej tematów Pomocy

„[Zmienne](#)” na stronie 94

„[Metody](#)” na stronie 95

„[Akcesory pobierające i ustawiające](#)” na stronie 98

[Obsługa zdarzeń](#)

Przykład:tworzenie prostej aplikacji

Język ActionScript 3.0 może być wykorzystywany w wielu środowiskach programistycznych, łącznie z programami Flash Professional i Flash Builder oraz w edytorach tekstu.

W niniejszym przykładzie krok po kroku przedstawiono proces tworzenia i udoskonalania prostej aplikacji w języku ActionScript 3.0 przy użyciu programu Flash Professional lub Flash Builder. Przykładowa aplikacja ilustruje prosty schemat użycia zewnętrznych plików klas ActionScript 3.0 we Flash Professional i Flex.

Projektowanie aplikacji w języku ActionScript

Przykładową aplikacją ActionScript jest standardowa aplikacja „Hello World”, dlatego jej projekt jest bardzo prosty:

- Aplikacja ma nazwę `HelloWorld`.

- Wyświetla jedno pole tekstowe zawierające tekst „Hello World!”.
- Wykorzystuje jedną klasę zorientowaną obiektowo o nazwie Greeter. Taki projekt umożliwia użycie klasy z projektu Flash Professional lub Flex.
- W przykładzie najpierw tworzona jest podstawowa wersja aplikacji. Następnie należy dodać funkcję, która spowoduje wprowadzenie nazwy użytkownika przez użytkownika. Wówczas aplikacja sprawdzi, czy nazwa znajduje się na liście znanych użytkowników.

Po określeniu tej zwięzłej definicji można rozpocząć tworzenie aplikacji.

Tworzenie projektu HelloWorld i klasy Greeter

W opisie projektu aplikacji Hello World zawarty jest wymóg, aby kod aplikacji dało się łatwo wykorzystać w innych projektach. W celu osiągnięcia tego celu aplikacja wykorzystuje jedną klasę zorientowaną obiektowo, o nazwie Greeter. Ta klasa jest wykorzystywana z aplikacji utworzonej w programie Flash Builder lub Flash Professional.

Aby utworzyć projekt HelloWorld i klasę Greeter w programie Flex:

- 1 W programie Flash Builder wybierz kolejno opcje Plik > Nowy > Projekt Flex.
- 2 Wpisz HelloWorld jako nazwę projektu. Upewnij się, że typ aplikacji jest ustawiony na „Web (działa w Adobe Flash Player)”, a następnie kliknij przycisk Zakończ.

Program Flash Builder utworzy projekt i wyświetli go w eksploratorze projektu. Domyślnie projekt powinien zawierać plik o nazwie HelloWorld.mxml i plik ten powinien być otwarty w edytorze..

- 3 Teraz, aby utworzyć plik niestandardowej klasy ActionScript w programie Flash Builder, wybierz polecenie Plik > Nowy > Klasa ActionScript.
- 4 W oknie dialogowym Nowa klasa ActionScript, w polu Nazwa, wpisz słowo **Greeter** jako nazwę klasy, a następnie kliknij przycisk Zakończ.

Zostanie wyświetlone nowe okno edytora ActionScript.

Przejdź do punktu „[Dodawanie kodu do klasy Greeter](#)” na stronie 27.

W celu utworzenia klasy Greeter w programie Flash Professional:

- 1 W programie Flash Professional wybierz opcje Plik > Nowy.
- 2 W oknie dialogowym Nowy dokument wybierz typ Plik ActionScript, a następnie kliknij przycisk OK.

Zostanie wyświetlone nowe okno edytora ActionScript.

- 3 Wybierz polecenie Plik > Zapisz. Wybierz folder, w którym ma być zapisana aplikacja, nadaj plikowi ActionScript nazwę **Greeter.as** i kliknij przycisk OK.

Przejdź do punktu „[Dodawanie kodu do klasy Greeter](#)” na stronie 27.

Dodawanie kodu do klasy Greeter

Klasa Greeter definiuje obiekt Greeter, który jest używany w aplikacji HelloWorld.

Aby dodać kod do klasy Greeter:

- 1 Wpisz następujący kod w nowym pliku (możliwe, że część kodu zostanie wpisana automatycznie przez edytor):

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

Klasa Greeter zawiera jedną metodę `sayHello()`, która zwraca ciąg znaków „Hello World!”.

- 2 Wybierz polecenie Plik > Zapisz, aby zapisać ten plik ActionScript.

Klasa Greeter jest teraz gotowa do użycia w aplikacji.

Tworzenie aplikacji korzystającej z kodu w języku ActionScript

Zbudowana właśnie klasa Greeter zawiera autonomiczny zestaw funkcji programowych, ale nie stanowi kompletnej aplikacji. Aby możliwe było korzystanie z tej klasy, utwórz dokument Flash Professional lub projekt Flex.

W kodzie wymagana jest instancja klasy Greeter. Oto sposób dołączenia klasy Greeter do naszej aplikacji.

Aby utworzyć aplikację ActionScript za pomocą narzędzia Flash Professional:

- 1 Wybierz polecenie Plik > Nowy.
- 2 W oknie dialogowym Nowy dokument wybierz typ Plik Flash (ActionScript 3.0), a następnie kliknij przycisk OK.
Zostanie otwarte nowe okno dokumentu.
- 3 Wybierz polecenie Plik > Zapisz. Wybierz folder zawierający plik klasy Greeter.as, nadaj dokumentowi Flash nazwę **HelloWorld.fla** i kliknij przycisk OK.
- 4 W palecie narzędzi Flash Professional wybierz narzędzie Tekst. Przesuń kursor po stole montażowym, aby zdefiniować nowe pole tekstowe o wymiarach: około 300 pikseli szerokości i 100 pikseli wysokości.
- 5 W panelu Właściwości, gdy pole tekstowe wciąż jest zaznaczone na stole montażowym, wybierz typ tekstu „Tekst dynamiczny” oraz wpisz **mainText** jako nazwę instancji pola tekstowego.
- 6 Kliknij pierwszą klatkę na głównej osi czasu. Otwórz panel Operacje, wybierając polecenie Okno > Operacje.
- 7 W panelu Operacje wpisz następujący skrypt:

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

- 8 Zapisz plik.

Przejdź do punktu „[Publikowanie i testowanie aplikacji w języku ActionScript](#)” na stronie 29.

Aby utworzyć aplikację ActionScript za pomocą narzędzia Flash Builder:

- 1 Otwórz plik HelloWorld.mxml i uzupełnij zawarty w nim kod, tak aby uzyskać treść przedstawioną na poniższym listingu:

```
<?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>
```

Ten projekt Flex zawiera cztery znaczniki MXML:

- Znacznik `<s:Application>`, który definiuje kontener Application.
- Znacznik `<s:layout>`, który definiuje styl układu (układ pionowy) znacznika Application.
- Znacznik `<fx:Script>`, który zawiera kod ActionScript.
- Znacznik `<s:TextArea>`, który definiuje pole służące do wyświetlania komunikatów tekstowych dla użytkownika.

Kod w znaczniku `<fx:Script>` definiuje metodę `initApp()` wywoływaną po załadowaniu aplikacji. Metoda `initApp()` ustawia wartość tekstu w polu tekstowym `TextArea mainTxt` na ciąg znaków „Hello World!”, zwracany przez metodę `sayHello()` niestandardowej klasy `Greeter` (którą właśnie napisaliśmy).

2 Wybierz polecenie `File > Save`, aby zapisać aplikację.

Przejdź do punktu „[Publikowanie i testowanie aplikacji w języku ActionScript](#)” na stronie 29.

Publikowanie i testowanie aplikacji w języku ActionScript

Tworzenie oprogramowania jest procesem iteracyjnym. Piszemy kod, próbujemy go skompilować i edytujemy, dopóki nie zostanie skompilowany bez błędów. Należy uruchomić skompilowaną aplikację i sprawdzić, czy jest zgodna z założeniami projektowymi. Jeśli nie, należy ponownie poddać kod edycji, aż aplikacja będzie działać zgodnie z oczekiwaniami. Środowiska programistyczne Flash Professional i Flash Builder oferują szereg metod publikowania, testowania i debugowania aplikacji.

Poniżej przedstawiono podstawowe etapy testowania aplikacji HelloWorld w każdym z tych środowisk.

Aby opublikować i przetestować aplikację ActionScript za pomocą programu Flash Professional.

- 1 Opublikuj aplikację, zwracając uwagę na ewentualne błędy kompilacji. W programie Flash Professional, wybierz polecenie Sterowanie > Testuj film, aby skompilować kod ActionScript i uruchomić aplikację HelloWorld.
 - 2 Jeśli podczas testowania aplikacji w oknie Wyjście zostaną wyświetlone błędy lub ostrzeżenia, należy skorygować te błędy w plikach HelloWorld.fla lub HelloWorld.as. Następnie należy ponownie przetestować aplikację.
 - 3 Jeśli podczas kompilacji nie wystąpią błędy, pojawi się okno odtwarzacza Flash Player z aplikacją Hello World.
- Stworzyliśmy prostą, ale kompletną aplikację w języku ActionScript 3.0. Przejdź teraz do punktu „[Udoskonalanie aplikacji HelloWorld](#)” na stronie 30.

Aby opublikować i przetestować aplikację ActionScript za pomocą narzędzia Flash Builder:

- 1 Wybierz polecenie Uruchom > Uruchom HelloWorld.
- 2 Aplikacja HelloWorld zostanie uruchomiona.
 - Jeśli podczas testowania aplikacji w oknie Wyjście zostaną wyświetlone błędy lub ostrzeżenia, należy skorygować te błędy w plikach HelloWorld.mxml lub Greeter.as. Następnie należy ponownie przetestować aplikację.
 - Jeśli podczas kompilacji nie wystąpią błędy, zostanie otwarte okno przeglądarki z aplikacją Hello World. Powinien zostać wyświetlony tekst „Hello World!”.

Stworzyliśmy prostą, ale kompletną aplikację w języku ActionScript 3.0. Przejdź teraz do punktu „[Udoskonalanie aplikacji HelloWorld](#)” na stronie 30.

Udoskonalanie aplikacji HelloWorld

Aby aplikacja była nieco bardziej interesująca, uzupełnimy ją o funkcję pytania o imię użytkownika i sprawdzania, czy imię to występuje na zdefiniowanej z góry liście imion.

Zacniemy od uzupełnienia klasy Greeter o nową funkcjonalność. Następnie zmodyfikujemy aplikację tak, aby korzystała z tej nowej funkcjonalności.

Aby zmodyfikować plik Greeter.as:

- 1 Otwórz plik Greeter.as.
- 2 Zmień jego treść na następującą (nowe i zmienione wiersze wyróżniono pogrubioną czcionką):

```
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;
            }
        }
    }
}
```

Klasa Greeter ma teraz kilka nowych możliwości i elementów:

- Tablica `validNames` zawiera listę poprawnych imion użytkowników. Tablica jest inicjowana trzema imionami podczas ładowania klasy Greeter.

- Metoda `sayHello()` ma teraz parametr w postaci imienia użytkownika i wybiera treść pozdrowienia w zależności od pewnych warunków. Jeśli parametr `userName` jest pustym ciągiem znaków (`""`), do właściwości `greeting` wpisywana jest prośba o podanie imienia. Jeśli imię użytkownika jest poprawne, pozdrowienie przyjmuje postać `"Hello, userName."` Jeśli żaden z tych dwóch warunków nie jest spełniony, zmiennej `greeting` przypisywany jest tekst `"Sorry userName, you are not on the list."` (Niestety, nie ma Cię na liście).
- Metoda `validName()` zwraca wartość `true`, jeśli imię `inputName` zostanie znalezione w tablicy `validNames`, oraz wartość `false`, jeśli nie zostanie znalezione. Instrukcja `validNames.indexOf(inputName)` porównuje każdy z ciągów znaków w tablicy `validNames` z ciągiem znaków `inputName`. Metoda `Array.indexOf()` zwraca położenie pierwszej instancji obiektu w tablicy. Zwraca wartość `-1`, jeśli obiekt nie został znaleziony w tablicy.

Następnie należy poddać edycji plik, który odnosi się do klasy `ActionScript`.

Aby zmodyfikować aplikację za pomocą programu Flash Professional:

- 1 Otwórz plik `HelloWorld.fla`.
- 2 Zmodyfikuj skrypt w klatce nr 1, aby do metody `sayHello()` klasy `Greeter` był przekazywany pusty ciąg (`""`):

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```
- 3 Wybierz narzędzie Tekst w paletce narzędzi. Utwórz dwa nowe pola tekstowe na stole montażowym. Umieść je obok siebie bezpośrednio pod istniejącym polem tekstowym `mainText`.
- 4 Do pierwszego nowego pola tekstowego, które będzie etykietą, wpisz tekst **User Name:**.
- 5 Zaznacz drugie z nowych pól tekstowych i w Inspektorze właściwości jako typ pola tekstowego wybierz Tekst wejściowy. Jako typ wiersza wybierz Pojedynczy. Jako nazwę instancji wpisz **textIn**.
- 6 Kliknij pierwszą klatkę na głównej osi czasu.
- 7 W panelu Operacje dodaj następujące wiersze do istniejącego skryptu:

```
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);
    }
}
```

Nowy kod wprowadza następującą funkcjonalność:

- Pierwsze dwa wiersze po prostu definiują granice dwóch pól tekstowych.
- Wejściowe pole tekstowe, takie jak `textIn` może wywoływać szereg różnych zdarzeń. Metoda `addEventListener()` pozwala na wskazanie funkcji wywoływanej po zajściu zdarzenia określonego typu. W tym przypadku zdarzeniem jest naciśnięcie klawisza na klawiaturze.
- Funkcja niestandardowa `keyPressed()` sprawdza, czy naciśniętym klawiszem jest Enter. Jeśli tak, wywołuje metodę `sayHello()` obiektu `myGreeter`, przekazując jako parametr tekst z pola `textIn`. Metoda ta zwraca tekst pozdrowienia właściwy dla przekazanej wartości. Następnie zwrócony ciąg znaków jest przypisywany do właściwości `text` pola tekstowego `mainText`.

Oto kompletny skrypt dla klatki nr 1:

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

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

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

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

8 Zapisz plik.

9 Wybierz polecenia Sterowanie > Testuj film, aby uruchomić aplikację.

Po uruchomieniu aplikacja wyświetli monit informujący o konieczności wprowadzenia nazwy użytkownika. Jeśli imię będzie poprawne (Sammy, Frank albo Dean), aplikacja wyświetli komunikat potwierdzający „Hello...”.

Aby zmodyfikować aplikację za pomocą narzędzia Flash Builder:

1 Otwórz plik HelloWorld.mxml.

2 Następnie zmodyfikuj znacznik `<mx:TextArea>` w taki sposób, aby wskazywał, że jest to pole tekstowe tylko do odczytu. Zmień kolor tła na jasnoszary i ustaw dla atrybutu `editable` wartość `false`:

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

3 Teraz dodaj następujące wiersze bezpośrednio po zamykającym znaczniku `<s:TextArea>`. Te wiersze tworzą składnik `TextInput`, w którym użytkownik może wprowadzić imię:

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

Atrybut `enter` definiuje operacje wykonywane, gdy użytkownik naciśnie klawisz Enter w polu `userNameTxt`. W tym przykładzie kod przekazuje tekst z pola do metody `Greeter.sayHello()`. Powoduje to zmianę powitania w polu `mainTxt`.

Plik `HelloWorld.mxml` wygląda w następujący sposób:

```
<?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 Zapisz zmodyfikowany plik HelloWorld.mxml. Wybierz polecenie Uruchom > Uruchom HelloWorld, aby uruchomić aplikację.

Po uruchomieniu aplikacja wyświetli monit informujący o konieczności wprowadzenia nazwy użytkownika. Je li imi b dzie poprawne (Sammy, Frank albo Dean), aplikacja wy wietli komunikat potwierdzaj cy „Hello, *userName*.”

Rozdział 3: Język ActionScript i jego składnia

Pod nazwą ActionScript 3.0 kryje się zarówno podstawowy język ActionScript, jak i interfejs programowania aplikacji (API, ang. Application Programming Interface) programu Adobe Flash Platform. Język podstawowy jest częścią języka ActionScript, która definiuje składnię języka, a także typy danych najwyższego poziomu. Język ActionScript 3.0 zapewnia programistyczny dostęp do środowisk wykonawczych platformy Adobe Flash: Adobe Flash Player i Adobe AIR.

Przegląd języka

Istotą języka ActionScript 3.0 oraz jego fundamentalnymi elementami składowymi są obiekty. Obiektem jest każda zadeklarowana zmienna, napisana funkcja i utworzona instancja klasy. Program ActionScript 3.0 można postrzegać jako grupę obiektów realizujących zadania, reagujących na zdarzenia i komunikujących się ze sobą nawzajem.

Programiści zaznajomieni z programowaniem zorientowanym obiektowo w języku Java lub C++ mogą traktować obiekty jak moduły zawierające dwa rodzaje członków: dane przechowywane w zmiennych obiektu (właściwościach) oraz zachowania dostępne za pośrednictwem metod. Język ActionScript 3.0 definiuje obiekt w sposób podobny, ale nieznacznie inny. W języku ActionScript 3.0 obiekty są po prostu kolekcjami właściwości. Właściwości te są kontenerami, które mogą zawierać nie tylko dane, lecz również funkcje lub inne obiekty. Funkcja dołączona w ten sposób do obiektu nazywana jest metodą.

Wprawdzie definicja języka ActionScript 3.0 może wydawać się nieco nietypowa z perspektywy programistów przyzwyczajonych do języków Java lub C++, w praktyce definiowanie typów obiektów za pomocą klas ActionScript 3.0 odbywa się bardzo podobnie, jak w języku Java lub C++. Rozróżnienie między dwiema definicjami obiektów jest ważne przy omawianiu modelu obiektów języka ActionScript oraz innych zaawansowanych zagadnień, ale w większości innych sytuacji termin *właściwości* oznacza zmienne należące do klasy, w odróżnieniu od metod. W skorowidzu Adobe ActionScript 3.0 dla platformy Adobe Flash termin *właściwości* oznacza zmienne lub właściwości pobierania/ustawiania. Termin *metody* oznacza funkcje stanowiące część klasy.

Jedną z subtelnych różnic między klasami w języku ActionScript a klasami w języku Java lub C++ jest fakt, że w języku ActionScript klasy nie są tylko jednostkami abstrakcyjnymi. Klasy ActionScript są reprezentowane przez *obiekty klas* zawierające właściwości i metody klasy. Umożliwia to stosowanie technik, które mogą wydawać się obce programistom posługującym się językami Java i C++ — na przykład umieszczanie instrukcji lub kodu wykonywalnego na najwyższym poziomie klasy lub pakietu.

Inna różnica między klasami języka ActionScript a klasami Java lub C++ polega na tym, że każda klasa ActionScript ma tzw. *obiektprototypowy*. W poprzednich wersjach języka ActionScript obiekty prototypowe, powiązane w *łańcuchy*, stanowiły łącznie fundament całej hierarchii dziedziczenia klas. Jednak w wersji ActionScript 3.0 obiekty prototypowe odgrywają tylko uboczną rolę w systemie dziedziczenia. Obiekt prototypowy może być nadal użyteczny jako alternatywa dla właściwości i metod statycznych, jeśli chcemy udostępnić właściwość i jej wartość wszystkim instancjom danej klasy.

W przeszłości zaawansowani programiści posługujący się językiem ActionScript mogli bezpośrednio manipulować łańcuchem prototypów za pomocą specjalnych wbudowanych elementów języka. Ponieważ obecnie język udostępnia bardziej dojrzałą implementację interfejsu programowania w oparciu o klasy, wiele z tych specjalnych elementów języka, m.in. `__proto__` i `__resolve`, usunięto z jego specyfikacji. Ponadto optymalizacja wewnętrznego mechanizmu dziedziczenia, która zapewnia znaczną poprawę wydajności, wyklucza bezpośredni dostęp do mechanizmu dziedziczenia.

Obiekty i klasy

W języku ActionScript 3.0 każdy obiekt jest zdefiniowany przez klasę. Klasę można wyobrazić sobie jako szablon lub wzorzec typu obiektu. Definicja klasy może zawierać zmienne i stałe przechowujące wartości danych oraz metody będące funkcjami, które realizują zachowania powiązane z klasą. Wartości przechowywane we właściwościach mogą być *wartościami pierwotnymi* lub innymi obiektami. Wartości pierwotne to liczby, ciągi znaków lub wartości typu Boolean.

Język ActionScript obejmuje szereg wbudowanych klas stanowiących część języka podstawowego. Niektóre z tych wbudowanych klas, takie jak Number, Boolean i String, odzwierciedlają wartości pierwotne dostępne w języku ActionScript. Inne klasy, takie jak Array, Math i XML, definiują bardziej złożone obiekty.

Wszystkie klasy, zarówno wbudowane, jak i zdefiniowane przez użytkownika, dziedziczą z klasy Object. Programiści znający poprzednie wersje języka ActionScript powinni zwrócić uwagę, że typ danych Object nie jest już domyślnym typem danych, mimo że dziedziczą z niego wszystkie pozostałe klasy. W języku ActionScript 2.0 poniższe dwa wiersze kodu były równoważne, ponieważ brak wskazania typu oznaczał, że zmienna jest typu Object:

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

Jednak w języku ActionScript 3.0 wprowadzono koncepcję zmiennych bez typu, które można definiować na dwa sposoby:

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

Zmienna bez typu nie jest tożsama ze zmienną typu Object. Najważniejsza różnica polega na tym, że zmienne bez typu mogą przyjmować specjalną wartość `undefined`, natomiast zmienne typu Object nie mogą przyjmować tej wartości.

Programista może definiować własne klasy za pomocą słowa kluczowego `class`. Właściwości klasy można deklarować na trzy sposoby: stałe definiuje się za pomocą słowa kluczowego `const`, zmienne definiuje się za pomocą słowa kluczowego `var`, a właściwości pobierające/ustawiające definiuje się za pomocą odpowiednich atrybutów `get` i `set` w deklaracji metody. Do deklarowania metod służy słowo kluczowe `function`.

Instancję klasy tworzy się, korzystając z operatora `new`. Poniższy przykład ilustruje tworzenie instancji klasy Date o nazwie `myBirthday`.

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

Pakiety i przestrzenie nazw

Pakiety i przestrzenie nazw to pojęcia pokrewne. Pakiety umożliwiają powiązanie definicji klas w sposób usprawniający współużytkowanie kodu i minimalizujący konflikty nazw. Przestrzenie nazw umożliwiają sterowanie widocznością identyfikatorów, np. nazw właściwości i metod, i mogą być stosowane w kodzie niezależnie od tego, czy znajduje się on w pakiecie, czy poza pakietem. Pakiety pozwalają na porządkowanie plików klas, a przestrzenie nazw służą do zarządzania widocznością poszczególnych właściwości i metod.

Pakiety

Pakiety w języku ActionScript 3.0 są implementowane przy użyciu przestrzeni nazw, ale nie są z nimi tożsame. Deklaracja pakietu powoduje niejawnie utworzenie specjalnego typu przestrzeni nazw, który na pewno będzie znany w czasie kompilacji. Przestrzenie nazw utworzone jawnie nie zawsze są znane w czasie kompilacji.

W poniższym przykładzie zastosowano dyrektywę `package` do utworzenia prostego pakietu zawierającego jedną klasę:

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

Nazwa klasy w tym przykładzie to `SampleCode`. Ponieważ klasa znajduje się w pakiecie `samples`, kompilator automatycznie kwalifikuje nazwę klasy w czasie kompilacji, przekształcając ją do pełnej postaci: `samples.SampleCode`. Ponadto kompilator kwalifikuje nazwy wszelkich właściwości lub metod, a zatem `sampleGreeting` i `sampleFunction()` staną się odpowiednio nazwami kwalifikowanymi `samples.SampleCode.sampleGreeting` i `samples.SampleCode.sampleFunction()`.

Wielu programistów, zwłaszcza z doświadczeniem w języku Java, preferuje umieszczanie na najwyższym poziomie pakietu wyłącznie klas. Jednak w języku ActionScript 3.0 na najwyższym poziomie pakietu mogą znajdować się nie tylko klasy, lecz także zmienne, funkcje, a nawet instrukcje. Jednym z zaawansowanych zastosowań tej funkcji jest zdefiniowanie przestrzeni nazw na najwyższym poziomie pakietu, tak aby była ona dostępna dla wszystkich klas w tym pakiecie. Należy jednak zauważyć, że tylko dwa specyfikatory dostępu — `public` i `internal` — są dozwolone na najwyższym poziomie pakietu. W przeciwieństwie do języka Java, który dopuszcza deklarowanie klas zagnieżdżonych jako klas prywatnych, język ActionScript 3.0 nie obsługuje ani klas zagnieżdżonych, ani prywatnych.

Jednak pod wieloma innymi względami pakiety języka ActionScript 3.0 są podobne do pakietów w języku programowania Java. Jak widzimy w poprzednim przykładzie, w pełni kwalifikowane odniesienia do pakietu zapisuje się przy użyciu operatora kropki (`.`), podobnie jak w języku Java. Pakietów można użyć do zorganizowania kodu w intuicyjną strukturę hierarchiczną, z której będą mogli korzystać inni programiści. Usprawnia to współużytkowanie kodu, ponieważ każdy programista może utworzyć własne pakiety udostępnione następnie innym, a także używać we własnym kodzie pakietów utworzonych przez innych programistów.

Zastosowanie pakietów pomaga także w zapewnieniu unikalności używanych nazw i wyeliminowaniu ich konfliktów z innymi nazwami (identyfikatorami). W istocie wiele osób właśnie to zastosowanie postrzega jako najważniejszą zaletę pakietów. Załóżmy, że dwóch programistów, którzy chcą używać nawzajem swoich fragmentów kodu, utworzyło klasy o nazwie `SampleCode`. Bez pakietów powstałby konflikt nazw, a jedynym rozwiązaniem byłaby zmiana nazwy jednej z klas. Jednak dzięki pakietom można w prosty sposób uniknąć konfliktu nazw, umieszczając każdą z klas w osobnym pakiecie (lub co najmniej jedną z klas w osobnym pakiecie). Wystarczy, że pakiety będą miały różne nazwy.

Dozwolone jest również umieszczanie kropek w nazwach pakietów w celu tworzenia pakietów zagnieżdżonych. Pozwala to na stworzenie hierarchicznej struktury pakietów. Dobrym przykładem jest pakiet `flash.display` dostępny w języku ActionScript 3.0. Pakiet `flash.display` jest zagnieżdżony w pakiecie `flash`.

Większa część języka ActionScript 3.0 jest zorganizowana w pakiecie `flash`. Na przykład pakiet `flash.display` zawiera interfejs API listy wyświetlania, a pakiet `flash.events` zawiera nowy model zdarzeń.

Tworzenie pakietów

Język ActionScript 3.0 zapewnia dużą swobodę w organizowaniu pakietów, klas i plików źródłowych. W poprzednich wersjach języka ActionScript na jeden plik źródłowy mogła przypadać tylko jedna klasa, a nazwa pliku źródłowego musiała być identyczna z nazwą klasy. Język ActionScript 3.0 dopuszcza umieszczenie wielu klas w jednym pliku źródłowym, ale tylko jedna klasa w każdym pliku może być dostępna dla kodu zewnętrznego względem tego pliku. Innymi słowy, tylko jedna klasa w każdym pliku może być zadeklarowana wewnątrz deklaracji pakietu. Ewentualne dodatkowe klasy należy zadeklarować poza definicją pakietu, przez co staną się one niewidoczne dla kodu poza plikiem źródłowym. Nazwa klasy zadeklarowanej w definicji pakietu musi być identyczna z nazwą pliku źródłowego.

Język ActionScript 3.0 zapewnia także większą elastyczność w sposobie deklarowania pakietów. W poprzednich wersjach języka ActionScript pakiety odzwierciedlały po prostu katalogi zawierające pliki źródłowe i nie były deklarowane za pomocą instrukcji `package`; zamiast tego nazwa pakietu była uwzględniana w pełni kwalifikowanej nazwie klasy w ramach deklaracji klasy. Mimo że w języku ActionScript 3.0 pakiety nadal odzwierciedlają katalogi, mogą teraz zawierać nie tylko klasy. W języku ActionScript 3.0 do deklarowania pakietów używana jest instrukcja `package`, co oznacza, że na najwyższym poziomie pakietu można także deklarować zmienne, funkcje i przestrzenie nazw. Dozwolone jest nawet umieszczanie instrukcji wykonywalnych na najwyższym poziomie pakietu. W przypadku zadeklarowania zmiennych, funkcji lub przestrzeni nazw na najwyższym poziomie pakietu jedynymi atrybutami dostępnymi na tym poziomie są `public` i `internal`, a ponadto tylko jedna deklaracja na poziomie pakietu w danym pliku może zawierać atrybut `public`, niezależnie od tego, czy jest to deklaracja klasy, zmiennej, funkcji, czy przestrzeni nazw.

Pakiety są użyteczne do organizowania kodu i zapobiegania konfliktom nazw. Nie należy mylić koncepcji pakietów z zupełnie odrębną koncepcją dziedziczenia klas. Dwie klasy rezydujące w tym samym pakiecie mają wspólną przestrzeń nazw, ale nie muszą być wcale powiązane ze sobą w żaden inny sposób. Podobnie, pakiet zagnieżdżony nie musi mieć relacji semantycznej ze swoim pakietem nadrzędnym.

Importowanie pakietów

Jeśli chcemy użyć klasy znajdującej się wewnątrz pakietu, musimy zaimportować albo cały pakiet, albo tę konkretną klasę. Jest to rozwiązanie różne od przyjętego w języku ActionScript 2.0, w którym importowanie klas było opcjonalne.

Przykład: weźmy pod uwagę klasę `SampleCode` przedstawioną poprzednio. Jeśli klasa rezyduje w pakiecie o nazwie `samples`, należy użyć jednej z poniższych instrukcji importu przed użyciem klasy `SampleCode`:

```
import samples.*;
```

lub

```
import samples.SampleCode;
```

Co do zasady instrukcje `import` powinny być maksymalnie precyzyjne. Jeśli planowane jest użycie tylko klasy `SampleCode` z pakietu `samples`, należy zaimportować tylko klasę `SampleCode`, a nie cały pakiet, do którego ona należy. Zaimportowanie całego pakietu może doprowadzić do nieoczekiwanych konfliktów nazw.

Ponadto konieczne jest umieszczenie kodu źródłowego definiującego pakiet lub klasę w *ścieżce klas*. Ścieżka klas jest to zdefiniowana przez użytkownika lista ścieżek do katalogów lokalnych, w których kompilator poszukuje importowanych pakietów i klas. Ścieżka klas jest czasami nazywana *ścieżką budowania* lub *ścieżką źródłową*.

Po prawidłowym zaimportowaniu klasy lub pakietu można używać w pełni kwalifikowanej nazwy klasy (`samples.SampleCode`) lub tylko nazwy klasy (`SampleCode`).

W pełni kwalifikowane nazwy są użyteczne, gdy identyczne nazwy klas, metod lub właściwości prowadziłyby do niejednoznaczności w kodzie, jednak z drugiej strony używanie ich we wszystkich identyfikatorach może być uciążliwe. Na przykład zastosowanie w pełni kwalifikowanej nazwy prowadzi do niepotrzebnej rozwlekłości kodu tworzącego instancję klasy `SampleCode`:

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

Im więcej poziomów zagnieżdżenia pakietów, tym mniej czytelny będzie kod. W sytuacjach, w których mamy pewność, że niejednoznaczne identyfikatory nie stanowią problemu, można poprawić czytelność kodu, stosując proste identyfikatory. Na przykład utworzenie nowej instancji klasy `SampleCode` można zapisać znacznie krócej, jeśli użyjemy tylko identyfikatora klasy:

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

Próba użycia nazw bez wcześniejszego zaimportowania odpowiedniego pakietu lub klasy spowoduje, że kompilator nie będzie mógł znaleźć definicji klas. Z drugiej strony po zaimportowaniu pakietu lub klasy każda próba zdefiniowania nazwy kolidującej z nazwą zaimportowaną spowoduje błąd.

Po utworzeniu pakietu domyślnym specyfikatorem dostępu wszystkich jego elementów jest specyfikator `internal`, co oznacza, że domyślnie elementy pakietu są widoczne tylko dla innych elementów tego samego pakietu. Aby klasa była dostępna dla kodu poza pakietem, należy ją zadeklarować jako `public`. Poniższy przykładowy pakiet zawiera dwie klasy: `SampleCode` i `CodeFormatter`:

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

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

Klasa `SampleCode` jest widoczna na zewnątrz pakietu, ponieważ jest zadeklarowana jako publiczna (`public`). Jednak klasa `CodeFormatter` jest widoczna tylko wewnątrz pakietu `samples`. Próba dostępu do klasy `CodeFormatter` poza pakietem `samples` spowoduje błąd, co ilustruje poniższy przykład:

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

Aby obie klasy były dostępne na zewnątrz pakietu, należy zadeklarować obie jako publiczne (`public`). Nie jest dozwolone zastosowanie atrybutu `public` do deklaracji pakietu.

W pełni kwalifikowane nazwy są przydatne do eliminowania konfliktów nazw, jakie mogą wystąpić podczas korzystania z pakietów. Taka sytuacja może wystąpić np. w wypadku zaimportowania dwóch pakietów, które definiują klasy o tym samym identyfikatorze. Na przykład, rozważmy następujący pakiet, który także zawiera klasę o nazwie `SampleCode`:

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

Jeżeli zaimportujemy obie klasy, tak jak poniżej, to przy próbie użycia klasy `SampleCode` wystąpi konflikt nazw:

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

Kompilator nie wie, której klasy `SampleCode` ma użyć. Aby rozstrzygnąć ten konflikt, należy użyć w pełni kwalifikowanej nazwy każdej z klas:

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

Uwaga: Programiści z doświadczeniem w programowaniu w języku C++, często mylą instrukcję `import` z instrukcją `#include`. Dyrektywa `#include` jest potrzebna w języku C++, ponieważ kompilatory C++ przetwarzają po jednym pliku naraz i nie odczytują definicji klas z innych plików, o ile ich nagłówki nie zostaną jawnie włączone do bieżącego pliku. W języku ActionScript 3.0 występuje dyrektywa `include`, jednak nie służy ona do importowania klas ani pakietów. Do importowania klas lub pakietów w języku ActionScript 3.0 należy używać instrukcji `import`, a w ścieżce klas należy umieścić plik źródłowy zawierający pakiet.

Przestrzenie nazw

Przestrzenie nazw zapewniają kontrolę nad widocznością właściwości i metod tworzonych przez programistę. Specyfikatory dostępu `public`, `private`, `protected`, i `internal` można potraktować w istocie jako wbudowane przestrzenie nazw. Jeśli te predefiniowane specyfikatory dostępu nie odpowiadają potrzebom w danym programie, można utworzyć własne przestrzenie nazw.

Dla czytelników, którym nieobca jest koncepcja przestrzeni nazw w języku XML, większość z poniższego omówienia będzie zawierać informacje już znane, z tym że składnia i szczegóły implementacji w języku ActionScript nieznacznie różnią się od specyfikacji języka XML. Jeśli czytelnik nie pracował jeszcze z przestrzeniami nazw, zrozumienie samej koncepcji nie powinno nastręczać problemów, konieczne jest natomiast zapoznanie się z charakterystyczną terminologią używaną w tej implementacji języka.

Aby zrozumieć koncepcję przestrzeni nazw, warto uświadomić sobie, że nazwa właściwości lub metody zawsze składa się z dwóch części: identyfikatora i przestrzeni nazw. Identyfikator jest tą częścią, którą intuicyjnie postrzegamy jako nazwę. Na przykład identyfikatorami w poniższej definicji klasy są `sampleGreeting` oraz `sampleFunction()`:

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

Gdy definicja nie jest poprzedzona atrybutem `namespace`, definiowana nazwa jest kwalifikowana domyślną przestrzenią nazw `internal`, co oznacza, że jest widoczna tylko dla kodu wywołującego w tym samym pakiecie. W trybie ścisłym kompilator generuje ostrzeżenie mówiące, że przestrzeń nazw `internal` obowiązuje dla wszystkich identyfikatorów bez atrybutu `namespace`. Aby sprawić, że identyfikator będzie dostępny wszędzie, należy jawnie poprzedzić jego nazwę atrybutem `public`. W poprzednim przykładowym kodzie zarówno identyfikator `sampleGreeting`, jak i identyfikator `sampleFunction()` należą do przestrzeni nazw `internal`.

Korzystając z przestrzeni nazw, programista wykonuje trzy podstawowe kroki. Najpierw należy zdefiniować przestrzeń nazw, używając słowa kluczowego `namespace`. Na przykład w poniższym kodzie zdefiniowano przestrzeń nazw `version1`:

```
namespace version1;
```

Następnie należy zastosować przestrzeń nazw, podając jej nazwę zamiast specyfikatora dostępu w deklaracji właściwości lub metody. W poniższym przykładzie funkcja o nazwie `myFunction()` została umieszczona w przestrzeni nazw `version1`:

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

Po trzecie, już po zastosowaniu przestrzeni nazw, można odwoływać się do niej za pomocą dyrektywy `use` lub poprzez kwalifikację identyfikatora nazwą przestrzeni nazw. W poniższym przykładzie występuje odwołanie do funkcji `myFunction()` przy użyciu dyrektywy `use`:

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

Do funkcji `myFunction()` można się też odwoływać za pomocą nazwy kwalifikowanej, co ilustruje poniższy przykład:

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

Definiowanie przestrzeni nazw

Każda przestrzeń nazw zawiera jedną wartość, tzw. identyfikator URI (ang. Uniform Resource Identifier), który jest czasami nazywany *nazwą przestrzeni nazw*. Identyfikator URI zapewnia unikalność definicji przestrzeni nazw.

Przestrzeń nazw można utworzyć, deklarując jej definicję na jeden z dwóch sposobów. Przestrzeń nazw można zdefiniować, podając jawnie identyfikator URI (tak jak przy definiowaniu przestrzeni nazw XML), lub nie podając identyfikatora URI. Poniższy przykład ilustruje definiowanie przestrzeni nazw przy użyciu identyfikatora URI:

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

URI jest ciągiem znaków jednoznacznie identyfikującym daną przestrzeń nazw. W wypadku pominięcia identyfikatora URI, tak jak w poniższym przykładzie, kompilator tworzy w jego miejsce unikalny wewnętrzny ciąg identyfikacyjny. Programista nie ma dostępu do tego wewnętrznego ciągu znaków.

```
namespace flash_proxy;
```

Raz zdefiniowanej przestrzeni nazw — z użyciem lub bez użycia identyfikatora URI — nie można ponownie zdefiniować w tym samym zasięgu. Próba zdefiniowania przestrzeni nazw, która została już wcześniej zdefiniowana w tym samym zasięgu, spowoduje błąd kompilatora.

Przestrzeń nazw zdefiniowana w pakiecie lub klasie może być niewidoczna dla kodu poza tym pakietem lub klasą, o ile nie zostanie użyty odpowiedni specyfikator dostępu. Poniższy przykładowy kod przedstawia definicję przestrzeni nazw `flash_proxy` wewnątrz pakietu `flash.utils`. W przykładzie tym brak specyfikatora dostępu oznacza, że klasa `flash_proxy` będzie widoczna tylko dla kodu w pakiecie `flash.utils` i że będzie niewidoczna dla kodu poza pakietem:

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

W poniższym kodzie użyto atrybutu `public`, aby przestrzeń nazw `flash_proxy` była widoczna dla kodu poza pakietem:

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

Stosowanie przestrzeni nazw

Zastosowanie przestrzeni nazw polega na umieszczeniu definicji w tej przestrzeni. W przestrzeniach nazw można umieszczać funkcje, zmienne i stałe (nie można umieścić klasy w niestandardowej przestrzeni nazw).

Rozważmy na przykład funkcję zadeklarowaną przy użyciu specyfikatora dostępu (przestrzeni nazw) `public`. Użycie atrybutu `public` w definicji funkcji powoduje umieszczenie tej funkcji w publicznej przestrzeni nazw, przez co staje się ona dostępna dla całego kodu. Po zdefiniowaniu niestandardowej przestrzeni nazw można używać jej w taki sam sposób, jak atrybutu `public`, co spowoduje, że definicje będą dostępne dla kodu, który może odwoływać się do niestandardowej przestrzeni nazw. Na przykład, jeśli zdefiniujemy przestrzeń nazw `example1`, możemy dodać metodę `myFunction()` z atrybutem `example1`, co ilustruje poniższy przykład:

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

Zadeklarowanie metody `myFunction()` przy użyciu przestrzeni nazw `example1` jako atrybutu oznacza, że metoda ta będzie należała do przestrzeni nazw `example1`.

Stosując przestrzenie nazw, należy pamiętać o następujących uwarunkowaniach:

- Do każdej deklaracji można zastosować tylko jedną przestrzeń nazw.
- Nie ma możliwości zastosowania atrybutu przestrzeni nazw do więcej niż jednej definicji naraz. Innymi słowy, jeśli chcemy zastosować własną przestrzeń nazw do dziesięciu różnych funkcji, należy dodać tę przestrzeń nazw jako atrybut do definicji każdej z tych funkcji.
- Zastosowanie przestrzeni nazw wyklucza jednocześnie zastosowanie specyfikatora dostępu. Innymi słowy, nie można zadeklarować funkcji lub właściwości z atrybutem `public`, `private`, `protected`, lub `internal` i dodatkowo zastosować własnej przestrzeni nazw.

Odwołania do przestrzeni nazw

Nie jest konieczne jawne odwoływanie się do przestrzeni nazw, gdy używamy metod lub właściwości zadeklarowanych za pomocą dowolnego ze specyfikatorów dostępu, np. `public`, `private`, `protected` lub `internal`. Wynika to z faktu, że dostęp do przestrzeni nazw identyfikowanych przez te specyfikatory odbywa się na podstawie kontekstu. Na przykład definicje umieszczone w przestrzeni nazw `private` będą automatycznie dostępne dla kodu w tej samej klasie. Jednak w przypadku przestrzeni nazw zdefiniowanych przez użytkownika dostępność nie jest zależna od kontekstu. Aby użyć metody lub właściwości umieszczonej w niestandardowej przestrzeni nazw, należy odwołać się do tej przestrzeni.

Do przestrzeni nazw można się odwołać za pomocą dyrektywy `use namespace` lub kwalifikując nazwę przestrzeni nazw za pomocą operatora `::`. Odwołanie za pomocą dyrektywy `use namespace` „otwiera” przestrzeń nazw, tak że zaczyna ona obowiązywać dla wszystkich identyfikatorów niekwalifikowanych. Na przykład, jeśli została zdefiniowana przestrzeń nazw `example1`, można uzyskiwać dostęp do nazw w tej przestrzeni po wydaniu dyrektywy `use namespace example1`:

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

Możliwe jest otwarcie więcej niż jednej przestrzeni nazw w danej chwili. Przestrzeń nazw otwarta za pomocą dyrektywy `use namespace` pozostaje otwarta do końca bloku kodu zawierającego tę dyrektywę. Nie ma możliwości jawnego zamknięcia przestrzeni nazw.

Jednak otwarcie więcej niż jednej przestrzeni nazw naraz zwiększa prawdopodobieństwo konfliktów nazw. Aby uniknąć otwierania przestrzeni nazw za pomocą dyrektywy `use namespace`, można kwalifikować nazwy metod lub właściwości za pomocą nazwy przestrzeni nazw i operatora `::`. Poniższy przykładowy kod ilustruje kwalifikowanie nazwy `myFunction()` przestrzenią nazw `example1`:

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

Korzystanie z przestrzeni nazw

W klasie `flash.utils.Proxy`, która jest częścią języka ActionScript 3.0, można znaleźć rzeczywiste przestrzenie nazw użyte w celu zapobieżenia konfliktom nazw. Klasa `Proxy`, która zastępuje właściwość `Object.__resolve` występującą w języku ActionScript 2.0, umożliwia przechwytywanie odwołań do niezdefiniowanych właściwości lub metod zanim takie odwołanie spowoduje błąd. Wszystkie metody klasy `Proxy` rezydują w przestrzeni nazw `flash_proxy`, co zapobiega konfliktom nazw.

Aby lepiej zrozumieć sposób wykorzystania przestrzeni nazw `flash_proxy`, należy zapoznać się z zasadami korzystania z klasy `Proxy`. Funkcjonalność klasy `Proxy` jest dostępna tylko dla klas, które z niej dziedziczą. Innymi słowy, aby można było używać metod klasy `Proxy` w obiekcie, klasa tego obiektu musi rozszerzać klasę `Proxy`. Na przykład, jeśli chcemy przechwytywać próby wywołania niezdefiniowanej metody, należałoby rozszerzyć klasę `Proxy`, a następnie przesłonić metodę `callProperty()` tej klasy.

Jak pamiętamy, implementacja przestrzeni nazw przebiega zazwyczaj w trzech krokach i polega na zdefiniowaniu, zastosowaniu przestrzeni nazw i odwołaniu się do niej. Ponieważ jednak metody klasy `Proxy` nigdy nie są wywoływane jawnie, przestrzeń nazw `flash_proxy` jest zdefiniowana i zastosowana, ale kod nigdy się do niej nie odwołuje. Język ActionScript 3.0 definiuje przestrzeń nazw `flash_proxy` i stosuje ją w klasie `Proxy`. Kod użytkownika musi stosować przestrzeń nazw `flash_proxy` tylko do klas rozszerzających klasę `Proxy`.

Przestrzeń nazw `flash_proxy` jest zdefiniowana w pakiecie `flash.utils` w sposób podobny do przedstawionego poniżej:

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

Przestrzeń nazw jest stosowana do metod klasy `Proxy` tak, jak ilustruje to poniższy fragment tej klasy:

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

Tak jak przedstawiono to w poniższym przykładzie, należy najpierw zaimportować zarówno klasę `Proxy`, jak i przestrzeń nazw `flash_proxy`. Następnie należy zadeklarować własną klasę tak, aby rozszerzała ona klasę `Proxy` (należy także dodać atrybut `dynamic`, jeśli kompilacja odbywa się w trybie ścisłym). Przesłaniając metodę `callProperty()`, należy użyć przestrzeni nazw `flash_proxy`.

```
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);
        }
    }
}
```

W wypadku utworzenia instancji klasy `MyProxy` i wywołania metody niezdefiniowanej, jak metoda `testing()` wywoływana w poniższym przykładzie, obiekt `Proxy` przechwyci wywołanie metody i wykona instrukcje wewnątrz przesłoniętej metody `callProperty()` (w tym przypadku jest to prosta instrukcja `trace()`).

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

Umieszczenie metod klasy `Proxy` wewnątrz przestrzeni nazw `flash_proxy` przynosi dwie korzyści. Po pierwsze, użycie odrębnej przestrzeni nazw upraszcza interfejs publiczny każdej klasy, która rozszerza klasę `Proxy`. (Klasa `Proxy` zawiera kilkanaście metod, które można przesłaniać, a które nie są przeznaczone do bezpośredniego wywoływania. Umieszczenie ich wszystkich w przestrzeni nazw publicznej byłoby mylące). Po drugie użycie przestrzeni nazwa `flash_proxy` pozwala uniknąć konfliktów nazw w sytuacji, gdy podklasa `Proxy` zawiera metody instancji o nazwach identycznych z nazwami metod klasy `Proxy`. Załóżmy, że jednej z naszych metod chcemy nadać nazwę `callProperty()`. Poniższy kod jest formalnie poprawny, ponieważ nasza wersja metody `callProperty()` znajduje się w innej przestrzeni nazw:

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

Przestrzenie nazw są także pomocne, gdy chcemy zapewnić dostęp do metod lub właściwości w sposób niemożliwy do zrealizowania za pomocą żadnego z czterech specyfikatorów dostępu (`public`, `private`, `internal` i `protected`). Załóżmy na przykład, że mamy kilka metod pomocniczych rozproszonych po kilku pakietach. Chcemy, aby te metody były dostępne dla wszystkich naszych pakietów, ale nie chcemy deklarować ich jako publicznych. Aby uzyskać pożądany skutek, utworzymy przestrzeń nazw i użyjemy jej jako własnego specjalnego specyfikatora dostępu.

W poniższym przykładzie zastosowano przestrzeń nazw zdefiniowaną przez użytkownika w celu zgrupowania dwóch funkcji rezydujących w różnych pakietach. Zgrupowanie ich w jednej przestrzeni nazw powoduje, że mogą być uwidocznione dla klasy lub pakietu za pomocą jednej instrukcji `use namespace`.

W tym przykładzie omawianą technikę zademonstrowano przy wykorzystaniu czterech plików. Wszystkie te pliki muszą znajdować się w ścieżce klas. Pierwszy plik, `myInternal.as`, definiuje przestrzeń nazw `myInternal`. Ponieważ plik ten znajduje się w pakiecie o nazwie `example`, należy umieścić go w folderze o nazwie `example`. Przestrzeń nazw jest oznaczona jako `public`, a więc można ją importować do innych pakietów.

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

Drugi i trzeci plik, Utility.as oraz Helper.as, definiują klasy zawierające metody, które powinny być dostępne w innych pakietach. Klasa Utility znajduje się w pakiecie example.alpha, co oznacza, że plik należy umieścić w folderze o nazwie alpha będącym podfolderem folderu example. Klasa Helper znajduje się w pakiecie example.beta, co oznacza, że plik należy umieścić w folderze o nazwie beta będącym podfolderem folderu example. Oba pakiety, example.alpha i example.beta, muszą zaimportować przestrzeń nazw, zanim zostanie ona użyta.

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

Czwarty plik, NamespaceUseCase.as, to główna klasa aplikacji, i należy go umieścić na tym samym poziomie, co folder example. W programie Flash Professional byłaby to klasa dokumentu dla pliku FLA. Klasa NamespaceUseCase również importuje przestrzeń nazw myInternal i używa jej do wywołania dwóch metod statycznych rezydujących w innych pakietach. W przykładzie zastosowano metody statyczne wyłącznie w celu uproszczenia kodu. W przestrzeni nazw myInternal można umieszczać zarówno metody statyczne, jak i metody instancji.

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

Zmienne

Zmienne umożliwiają przechowywanie wartości używanych w programie. W celu zadeklarowania zmiennej należy użyć instrukcji var z nazwą zmiennej. W języku ActionScript 3.0 instrukcja var zawsze jest wymagana. Na przykład w poniższym wierszu kodu ActionScript deklarowana jest zmienna o nazwie i:

```
var i;
```

W wypadku pominięcia instrukcji var w deklaracji zmiennej zgłoszony zostanie błąd (błąd kompilatora w trybie ścisłym i błąd czasu wykonania w trybie standardowym). Poniższy wiersz kodu spowoduje błąd, jeśli zmienna i nie została wcześniej zadeklarowana:

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

Aby skojarzyć zmienną z typem danych, należy wskazać ten typ w deklaracji zmiennej. Deklarowanie zmiennej bez wskazania typu jest formalnie dozwolone, ale powoduje wygenerowanie przez kompilator ostrzeżenia w trybie ścisłym. Typ zmiennej wskazuje się, dodając do nazwy zmiennej dwukropek (:) i nazwę typu. Na przykład poniższy kod deklaruje zmienną i typu int:

```
var i:int;
```

Zmiennej można przypisać wartość, używając operatora przypisania (=). Na przykład poniższy kod deklaruje zmienną i, a następnie przypisuje jej wartość 20:

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

Niekiedy wygodniejszym rozwiązaniem jest przypisanie zmiennej wartości w momencie jej deklarowania, co ilustruje poniższy przykład:

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

Technika przypisywania zmiennym wartości już w momencie deklarowania jest powszechnie stosowana nie tylko do przypisywania wartości pierwotnych, jak liczby całkowite i ciągi znaków, lecz także przy tworzeniu tablic oraz instancji klas. Poniższy przykład ilustruje deklarację tablicy i przypisanie jej wartości w jednym wierszu kodu.

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

Instancję klasy można utworzyć, korzystając z operatora `new`. W poniższym przykładzie tworzona jest instancja klasy `CustomClass`, po czym odwołanie do nowo utworzonej instancji przypisywane jest zmiennej o nazwie `customItem`:

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

Jeśli mamy do zadeklarowania więcej niż jedną zmienną, możemy zadeklarować je wszystkie w jednym wierszu kodu, oddzielając je operatorem przecinka `,`. Na przykład w poniższym kodzie w jednym wierszu deklarowane są trzy zmienne:

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

W tym samym wierszu kodu można przypisać wartości wszystkim zmiennym. Na przykład w poniższym kodzie deklarowane są trzy zmienne (`a`, `b`, oraz `c`) i każdej z nich przypisywana jest wartość:

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

Mimo że dozwolone jest użycie przecinka do grupowania deklaracji zmiennych w jednej instrukcji, może to pogorszyć czytelność kodu.

Zasięg zmiennych — omówienie

Zasięg zmiennej jest to obszar kodu, w którym do zmiennej można odwoływać się leksykalnie. Zmienna *globalna* to zmienna zdefiniowana we wszystkich obszarach kodu, natomiast zmienna *lokalna* to zmienna zdefiniowana tylko w jednej części kodu. W języku ActionScript 3.0 zmienne mają zawsze przypisany zasięg funkcji lub klasy, w której zostały zadeklarowane. Zmienna globalna to zmienna zdefiniowana poza definicją jakiegokolwiek funkcji lub klasy. Na przykład poniższy kod tworzy zmienną globalną `strGlobal`, ponieważ jest ona zadeklarowana poza jakąkolwiek funkcją. Przykład ilustruje fakt, że zmienna globalna jest dostępna zarówno wewnątrz definicji funkcji, jak i poza tą definicją.

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

Zmienną lokalną deklaruje się wewnątrz definicji funkcji. Najmniejszym obszarem kodu, w jakim można zadeklarować zmienną lokalną, jest definicja funkcji. Zmienna lokalna zadeklarowana wewnątrz funkcji istnieje tylko w tej funkcji. Na przykład, jeśli zadeklarujemy zmienną o nazwie `str2` w funkcji o nazwie `localScope()`, zmienna ta nie będzie dostępna na zewnątrz funkcji.

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

Jeśli nazwa zmiennej lokalnej jest identyczna z nazwą zadeklarowanej zmiennej globalnej, to w zasięgu zmiennej lokalnej deklaracja lokalna ukryje (przesłoni) definicję globalną. Zmienna globalna istnieje nadal na zewnątrz funkcji. Na przykład poniższy kod tworzy zmienną globalną typu string o nazwie `str1`, a następnie tworzy zmienną lokalną o tej samej nazwie wewnątrz funkcji `scopeTest()`. Instrukcja `trace` wewnątrz funkcji wyświetla lokalną wartość zmiennej, ale instrukcja `trace` poza funkcją wyświetla wartość zmiennej globalnej.

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

W języku ActionScript, inaczej niż w języku C++ i Java, zmienne nie mają zasięgu na poziomie bloku. Blok kodu jest to dowolna grupa instrukcji między otwierającym nawiasem sześciennym (`{`) a zamykającym nawiasem sześciennym (`}`). W niektórych językach programowania, takich jak C++ i Java, zmienne zadeklarowane wewnątrz bloku kodu nie są dostępne poza tym blokiem. To ograniczenie zasięgu nie istnieje w języku ActionScript. Zmienna zadeklarowana wewnątrz bloku kodu jest dostępna nie tylko w tym bloku, lecz także we wszystkich innych częściach funkcji, do której należy ten blok kodu. Na przykład poniższa funkcja zawiera zmienne zdefiniowane w różnych blokach. Wszystkie te zmienne są dostępne w całej funkcji.

```
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"]);
```

Interesującą implikacją braku zasięgu na poziomie bloków jest możliwość odczytywania i przypisywania wartości zmiennej zanim zostanie ona zadeklarowana, pod warunkiem że deklaracja zmiennej znajduje się gdziekolwiek przed końcem funkcji. Wynika to z zastosowania w kompilatorze techniki niejawnego przenoszenia deklaracji na początek funkcji (ang. *hoisting*). Na przykład poniższy kod jest formalnie poprawny mimo że pierwsze wywołanie funkcji `trace()` ze zmienną `num` znajduje się przed deklaracją zmiennej `num`:

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

Przenoszenie deklaracji na początek funkcji nie jest jednak równoznaczne z analogicznym przenoszeniem instrukcji przypisania. Wyjaśnia to, dlaczego pierwsze wywołanie `trace()` dla zmiennej `num` zwraca wartość `NaN` (nie liczba), czyli domyślną wartość zmiennych typu `Number`. A zatem możliwe jest przypisywanie zmiennym wartości zanim zmienne te zostaną zadeklarowane, co ilustruje poniższy przykład:

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

Wartości domyślne

Wartość domyślna jest to wartość przechowywana w zmiennej zanim kod użytkownika przypisze jej inną wartość. Pierwsze przypisanie wartości zmiennej nazywamy *inicjowaniem* zmiennej. Jeśli zmienna została zadeklarowana, ale nie przypisano jej wartości, to zmienna jest *niezainicjowana*. Wartość zmiennej niezainicjowanej zależy od jej typu danych. W poniższej tabeli opisano domyślne wartości zmiennych dla poszczególnych typów danych:

Typ danych	Wartość domyślna
Boolean	false
int	0
Number	NaN
Obiekt	null
String	null
uint	0
Niezadeklarowany (równoważny wskazaniu typu *)	undefined
Wszystkie pozostałe klasy, w tym klasy zdefiniowane przez użytkownika.	null

W przypadku zmiennych typu Number wartością domyślną jest NaN (ang. not a number — nie liczba), czyli specjalna wartość określona w standardzie IEEE-754 jako niereprezentująca liczby.

W wypadku zadeklarowania zmiennej, ale niewskazania jej typu danych, zmienna otrzyma domyślny typ danych *, co zazwyczaj oznacza, że będzie zmienną bez typu. Jeśli zmienna bez typu nie zostanie zainicjowana wartością, będzie miała domyślną wartość undefined.

W przypadku typów danych innych niż Boolean, Number, int oraz uint wszystkie niezainicjowane zmienne mają wartość null. Dotyczy to wszystkich klas zdefiniowanych w języku ActionScript 3.0 oraz wszelkich niestandardowych klas utworzonych przez użytkownika.

Wartość null nie jest poprawną wartością zmiennych typu Boolean, Number, int, lub uint. Próba przypisania wartości null takiej zmiennej spowoduje przekonwertowanie wartości na wartość domyślną odpowiedniego typu danych. Zmiennym typu Object można przypisywać wartość null. Próba przypisania wartości undefined zmiennej typu Object spowoduje przekonwertowanie wartości na null.

W przypadku zmiennych typu Number dostępna jest specjalna funkcja globalna isNaN(), która zwraca wartość logiczną true, jeśli zmienna nie jest liczbą, albo false w przeciwnym razie.

Typy danych

Typ danych definiuje zbiór wartości. Na przykład typ danych Boolean to zbiór zawierający dokładnie dwie wartości: `true` i `false`. Obok typu danych Boolean w języku ActionScript 3.0 istnieje kilka częściej używanych typów danych, np. `String`, `Number` i `Array`. Programista może definiować własne typy danych, definiując własny zbiór wartości przy użyciu klas lub interfejsów. W języku ActionScript 3.0 wszystkie wartości, zarówno pierwotne, jak i złożone, są obiektami.

Wartość pierwotna to wartość należąca do jednego z następujących typów danych: `Boolean`, `int`, `Number`, `String` lub `uint`. Przetwarzanie wartości pierwotnych odbywa się zwykle szybciej niż przetwarzanie wartości złożonych, ponieważ w środowisku ActionScript wartości pierwotne są przechowywane w specjalny sposób umożliwiający optymalne wykorzystanie pamięci i przyspieszenie operacji.

Uwaga: *Z myślą o czytelnikach zainteresowanych szczegółami technicznymi wyjaśniamy, że w środowisku ActionScript wartości pierwotne są wewnętrznie przechowywane jako obiekty niezmiennicze. Fakt, że są przechowywane jako obiekty niezmiennicze, oznacza, że przekazywanie ich przez referencję (odwołanie) jest faktycznie tożsame z przekazywaniem przez wartość. Ogranicza to zużycie pamięci i przyspiesza wykonywanie kodu, ponieważ odwołania zajmują zwykle znacznie mniej miejsca niż same wartości.*

Wartość złożona to każda wartość niebędąca wartością pierwotną. Do typów danych, które definiują zbiór wartości złożonych, należą `Array`, `Date`, `Error`, `Function`, `RegExp`, `XML` i `XMLList`.

W wielu językach programowania istnieje rozróżnienie między wartościami pierwotnymi a ich obiektami opakowanymi (ang. wrapper). Na przykład w języku Java istnieje typ wartości pierwotnych `int` oraz opakowująca je klasa `java.lang.Integer`. Wartości pierwotne w języku Java nie są obiektami, natomiast ich opakowania — już tak. Dlatego do pewnych operacji lepiej nadają się wartości pierwotne, a do innych lepiej jest zastosować odpowiednie obiekty opakowaniowe. W języku ActionScript 3.0 wartości pierwotne i ich obiekty opakowaniowe są, ze względów praktycznych, nierozróżnialne. Wszystkie wartości, nawet pierwotne, są obiektami. W czasie wykonywania te typy pierwotne są traktowane w specjalny sposób, dzięki czemu zachowują się jak obiekty, ale z ich tworzeniem nie jest związany narzut charakterystyczny dla klasycznych obiektów. Oznacza to, że następujące dwa wiersze kodu są równoważne:

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

Wszystkie wymienione powyżej pierwotne i złożone typy danych są zdefiniowane przez klasy podstawowe języka ActionScript 3.0. Klasy podstawowe umożliwiają tworzenie obiektów przy użyciu wartości literalnych, bez stosowania operatora `new`. Na przykład możliwe jest utworzenie tablicy przy użyciu wartości literalnej lub konstruktora klasy `Array`, tak jak przedstawiono to poniżej:

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

Kontrola typów

Kontrola typów może być realizowana w czasie kompilacji lub w czasie wykonywania. W językach z typami statycznymi, takich jak C++ i Java, kontrola typów odbywa się zawsze w czasie kompilacji. W językach z typami dynamicznymi, takich jak Smalltalk i Python, kontrola typów odbywa się w czasie wykonywania. Ponieważ w języku ActionScript 3.0 typy mają charakter dynamiczny, kontrola odbywa się w czasie wykonywania, jednak istnieje również możliwość kontroli typów w trakcie kompilacji w specjalnym trybie, nazywanym *trybem ścisłym*. W trybie ścisłym kontrola typów odbywa się zarówno w czasie kompilacji, jak i w czasie wykonywania, natomiast w trybie standardowym — jedynie w czasie wykonywania.

Języki z typami dynamicznymi oferują wyjątkową elastyczność w budowie struktury kodu, jednak za cenę potencjalnego ujawnienia błędów niezgodności typów w trakcie wykonywania programu. W językach z typami statycznymi błędy niezgodności typów są zgłaszane w trakcie kompilacji, jednak ich niedogodnością jest konieczność znajomości typów już na etapie kompilacji.

Kontrola typów w czasie kompilacji.

Kontrola typów w czasie kompilacji jest często preferowanym rozwiązaniem w dużych projektach, w których elastyczność w użyciu typów danych jest zazwyczaj mniej istotna niż możliwe wczesne wykrywanie wszelkich błędów. Dlatego domyślnie kompilator języka ActionScript w programach Flash Professional i Flash Builder działa w trybie ścisłym.

Adobe Flash Builder

W programie Flash Builder można wyłączyć tryb ścisły za pośrednictwem ustawień kompilatora ActionScript w oknie dialogowym właściwości projektu.

Aby możliwa była kontrola typów w trakcie wykonywania, kompilator musi dysponować informacjami o typach danych zmiennych lub wyrażeń w kodzie. Aby jawnie zadeklarować typ danych zmiennej, należy za jej nazwą umieścić operator dwukropka (:), a następnie nazwę typu. Aby skojarzyć typ danych z parametrem, należy użyć dwukropka wraz z następującą po nim nazwą typu. Na przykład w poniższym kodzie określany jest typ parametru `xParam` oraz deklarowana jest zmienna `myParam` o jawnie określonym typie danych:

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

W trybie ścisłym kompilator języka ActionScript zgłasza niezgodności typów jako błędy kompilatora. Na przykład w poniższym kodzie zadeklarowany jest parametr funkcji `xParam` typu `Object`, a następnie podejmowana jest próba przypisania temu parametrowi wartości typu `String` i `Number`. W trybie ścisłym spowoduje to zgłoszenie błędu przez kompilator.

```
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);
    }
}
```

Jednak nawet w trybie ścisłym możliwe jest selektywne wyłączenie kontrolowania zgodności typów w czasie wykonywania. Należy w tym celu pominąć wskazanie typu po prawej stronie instrukcji przypisania. Zmienną lub wyrażenie można oznaczyć jako niemającą/niemające typu, pomijając wskazanie typu lub używając gwiazdki (*) jako nazwy typu. Przykładowo, jeśli parametr `xParam` z poprzedniego przykładu zostanie zmodyfikowany tak, aby nie był wskazywany jego typ, poniższy kod zostanie skompilowany bez błędów nawet w trybie ścisłym:

```
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");
```

Kontrola typów w czasie wykonywania

W środowisku ActionScript 3.0 zgodność typów jest kontrolowana w czasie wykonywania niezależnie od tego, czy kompilacja odbywała się w trybie ścisłym, czy standardowym. Rozważmy sytuację, w której wartość 3 zostanie przekazana jako argument do funkcji, która oczekuje tablicy. W trybie ścisłym kompilator wygeneruje błąd, ponieważ wartość 3 jest niezgodna z typem danych Array. W wypadku wyłączenia trybu ścisłego i kompilacji w trybie standardowym kompilator nie zgłosi niezgodności typów, ale kontrola typów w trakcie wykonywania kodu spowoduje zgłoszenie błędu.

Poniższy przykład ilustruje funkcję o nazwie `typeTest()`, która oczekuje argumentu typu Array, a do której przekazano wartość 3. W trybie standardowym spowoduje to błąd czasu wykonywania, ponieważ wartość 3 nie należy do zadeklarowanego typu danych parametru (Array).

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

Mogą także wystąpić sytuacje, w których błąd w czasie wykonywania zostanie zgłoszony mimo przeprowadzenia kompilacji w trybie ścisłym. Jest to możliwe, jeśli kompilacja odbywa się w trybie ścisłym, ale zrezygnowano kontrolę typów w czasie kompilacji, ponieważ użyto zmiennej bez typu. Użycie zmiennej bez typu nie eliminuje kontroli typów, a jedynie odracza ją do czasu wykonywania. Na przykład, jeśli zmienna `myNum` z poprzedniego przykładu nie ma zadeklarowanego typu danych, kompilator nie może wykryć niezgodności typów, ale kod zgłosi błąd w czasie wykonywania, ponieważ porówna bieżącą wartość `myNum`, która w wyniku przypisania wynosi 3, z typem parametru `xParam`, czyli Array.

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

Kontrola typów dopiero w trakcie wykonywania pozwala także na bardziej elastyczne korzystanie z mechanizmów dziedziczenia. Dzięki odroczeniu kontroli typów do momentu wykonania kodu tryb standardowy umożliwia odwoływanie się do właściwości podklasy nawet jeśli dokonano rzutowania *rozszerzającego* (ang. upcast). Rzutowanie rozszerzające ma miejsce, gdy instancja zostaje zadeklarowana przy użyciu klasy bazowej, ale jest tworzona przy użyciu podklasy. Możemy na przykład utworzyć klasę `ClassBase`, którą można rozszerzyć (klas z atrybutem `final` nie można rozszerzać):

```
class ClassBase
{
}
```

Następnie możemy utworzyć podklasę klasy `ClassBase` o nazwie `ClassExtender`, która będzie miała jedną właściwość o nazwie `someString`:

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

Używając obu klas, możemy utworzyć instancję zadeklarowaną przy użyciu typu danych `ClassBase`, ale utworzoną faktycznie przy użyciu konstruktora klasy `ClassExtender`. Rzutowanie rozszerzające jest uznawane za operację bezpieczną, ponieważ klasa bazowa nie zawiera żadnych właściwości ani metod nieistniejących w podklasie.

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

Podklasa natomiast zawiera właściwości lub metody, których brak w klasie bazowej. Na przykład klasa `ClassExtender` zawiera właściwość `someString`, która nie istnieje w klasie `ClassBase`. W trybie standardowym kompilatora ActionScript 3.0 dozwolone jest odwoływanie się do tej właściwości z instancji `myClass` — nie spowoduje to zgłoszenia błędu w czasie kompilacji:

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

Operator `is`

Operator `is` umożliwia przetestowanie, czy zmienna lub wyrażenie należy do określonego typu danych. W poprzednich wersjach języka ActionScript tę samą funkcjonalność realizował operator `instanceof`, ale w języku ActionScript 3.0 operatora `instanceof` nie należy stosować do testowania przynależności do typów danych. Do ręcznego sprawdzania typów należy używać operatora `is`, a nie operatora `instanceof`, ponieważ wyrażenie `x instanceof y` sprawdza tylko, czy w łańcuchu prototypów `x` istnieje `y` (a w języku ActionScript 3.0 łańcuch prototypów nie odzwierciedla w pełni hierarchii dziedziczenia).

Operator `is` analizuje właściwą hierarchię dziedziczenia i może być stosowany nie tylko do sprawdzania, czy obiekt jest instancją konkretnej klasy, lecz także czy obiekt jest instancją klasy implementującej konkretny interfejs. W poniższym przykładzie tworzona jest instancja klasy `Sprite` o nazwie `mySprite`, po czym operator `is` używany jest do sprawdzania, czy `mySprite` jest instancją klas `Sprite` i `DisplayObject` oraz czy implementuje interfejs `IEventDispatcher`:

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

Operator `is` sprawdza hierarchię dziedziczenia i prawidłowo informuje, że instancja `mySprite` jest zgodna z klasami `Sprite` oraz `DisplayObject` (klasa `Sprite` jest podklasą klasy `DisplayObject`). Operator `is` sprawdza także, czy `mySprite` dziedziczy z jakichkolwiek klas, które implementują interfejs `IEventDispatcher`. Ponieważ klasa `Sprite` dziedziczy z klasy `EventDispatcher`, która implementuje interfejs `IEventDispatcher`, operator `is` prawidłowo informuje, że `mySprite` implementuje ten interfejs.

Poniższy przykład ilustruje te same testy, co poprzednio, ale wykonywane za pomocą operatora `instanceof`, a nie `is`. Operator `instanceof` prawidłowo rozpoznaje `mySprite` jako instancję klasy `Sprite` lub `DisplayObject`, ale zwraca `false`, gdy próbujemy sprawdzić, czy `mySprite` implementuje interfejs `IEventDispatcher`.

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

Operator `as`

Operator `as` również umożliwia sprawdzenie, czy wyrażenie należy do określonego typu danych. Jednak w przeciwieństwie do operatora `is`, operator `as` nie zwraca wartości typu `Boolean`. Operator `as` zwraca wartość wyrażenia zamiast `true` albo wartość `null` zamiast `false`. Poniższy przykład ilustruje wynik użycia operatora `as` zamiast operatora `is` w prostym przypadku sprawdzenia, czy instancja klasy `Sprite` należy do typów danych `DisplayObject`, `IEventDispatcher` i `Number`.

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

Operand po prawej stronie operatora `as` musi być typem danych. Próba użycia jako prawego operandu wyrażenia innego niż typ danych spowoduje zgłoszenie błędu.

Klasy dynamiczne

Klasa *dynamiczna* definiuje obiekt, który może być modyfikowany w czasie wykonywania poprzez dodawanie lub zmienianie właściwości i metod. Klasa, która nie jest dynamiczna, np. klasa `String`, jest klasą *zapięczętowaną*. Do klasy *zapięczętowanej* nie można dodawać właściwości ani metod w czasie wykonywania.

Aby utworzyć klasę dynamiczną, należy w jej deklaracji użyć atrybutu `dynamic`. Na przykład poniższy kod tworzy klasę dynamiczną o nazwie `Protean`:

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

Jeśli później utworzymy instancję klasy `Protean`, będzie możliwe dodawanie do niej właściwości lub metod poza jej definicją. Na przykład poniższy kod tworzy instancję klasy `Protean` i dodaje do tej instancji właściwości o nazwach `aString` oraz `aNumber`:

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

Właściwości dodane do instancji klasy dynamicznej istnieją w czasie wykonywania, dlatego wszelka kontrola typów również odbywa się w czasie wykonywania. Nie jest możliwe wskazanie typu właściwości dodanej w ten sposób.

Możliwe jest także dodanie metody do instancji `myProtean` — należałoby w tym celu zdefiniować funkcję i powiązać ją z właściwością instancji `myProtean`. W poniższym kodzie instrukcja `trace` została przeniesiona do metody o nazwie `traceProtean()`:

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

Jednak utworzone w ten sposób metody nie mają dostępu do właściwości ani metod prywatnych klasy `Protean`. Co więcej, nawet odwołania do publicznych właściwości lub metod klasy `Protean` muszą być kwalifikowane słowem kluczowym `this` albo nazwą klasy. Poniższy przykład ilustruje metodę `traceProtean()`, która próbuje uzyskać dostęp do zmiennych prywatnych i publicznych klasy `Protean`.

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

Opisy typów danych

Pierwotnymi typami danych są: `Boolean`, `int`, `Null`, `Number`, `String`, `uint` oraz `void`. Klasy podstawowe języka ActionScript definiują również następujące złożone typy danych: `Object`, `Array`, `Date`, `Error`, `Function`, `RegExp`, `XML` i `XMLList`.

Typ danych Boolean

Typ danych `Boolean` zawiera dwie wartości: `true` oraz `false`. Zmienne typu `Boolean` nie mogą przyjmować żadnych innych wartości. Domyślna wartość zadeklarowanej, ale niezainicjowanej zmiennej typu `Boolean` to `false`.

int, typ danych

Wartości typu danych `int` są przechowywane wewnętrznie jako 32-bitowe liczby całkowite; typ ten obejmuje zbiór liczb całkowitych od

-2 147 483 648 (-2^{31}) do 2 147 483 647 ($2^{31} - 1$) włącznie. We wcześniejszych wersjach języka ActionScript istniał tylko typ danych `Number` używany zarówno do liczb całkowitych, jak i zmiennopozycyjnych. W języku ActionScript 3.0 programista ma dostęp do niskopoziomowych typów maszynowych reprezentujących 32-bitowe liczby całkowite ze znakiem i bez znaku. Jeśli zmienna nie potrzebuje wartości ułamkowych lub bardzo dużych/małych, użycie typu `int` zamiast typu `Number` powinno zapewnić większą szybkość wykonywania programu i bardziej oszczędne wykorzystanie pamięci.

Do reprezentowania wartości całkowitych spoza zakresu typu `int` należy stosować typ danych `Number`, który uwzględnia wartości z zakresu od plus do minus 9 007 199 254 740 992 (53-bitowe liczby całkowite). Domyślna wartość zmiennych typu `int` to 0.

Typ danych Null

Typ danych `Null` zawiera tylko jedną wartość, `null`. Jest to domyślna wartość danych typu `String` oraz wszystkich klas definiujących złożone typy danych, w tym klasy `Object`. Żadne inne typy pierwotne, w tym `Boolean`, `Number`, `int` oraz `uint`, nie zawierają wartości `null`. W czasie wykonywania wartość `null` jest przekształcana na odpowiednią wartość domyślną, jeśli podjęto próbę przypisania `null` do wartości typu `Boolean`, `Number`, `int` lub `uint`. Tego typu danych nie można używać we wskazaniach typu.

Typ danych Number

W języku ActionScript 3.0 typ danych `Number` może reprezentować liczby całkowite ze znakiem, liczby całkowite bez znaku i liczby zmiennopozycyjne. Jednak ze względów wydajnościowych należy używać typu danych `Number` wyłącznie do reprezentacji liczb całkowitych niemieszczących się w zakresie 32-bitowych typów `int` i `uint` oraz liczb zmiennopozycyjnych. Aby liczba była przechowywana jako zmiennopozycyjna, należy zapisać ją z kropką dziesiętną. Pominięcie kropki dziesiętnej spowoduje, że liczba będzie przechowywana jako wartość całkowita

Dane typu `Number` są zapisywane w formacie 64-bitowym o podwójnej precyzji zgodnie ze specyfikacją IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754). Standard ten reguluje zasady zapisywania liczb zmiennopozycyjnych w słowach 64-bitowych. Jeden bit określa, czy liczba jest dodatnia, czy ujemna. Jedenaście bitów zajmuje wykładnik dla podstawy 2. Pozostałe 52 bity zawierają *mantysę* (ang. *significand* lub *mantissa*), czyli liczbę podnoszoną do potęgi określonej przez wykładnik.

Dzięki wykorzystaniu części bitów do przechowywania wykładnika, typ danych `Number` umożliwia przechowywanie liczb zmiennopozycyjnych znacznie większych niż wówczas, gdyby wszystkie bity były zajęte przez mantysę. Na przykład, gdyby w danych typu `Number` wszystkie 64 bity były zajęte przez mantysę, możliwe byłoby zapisywanie liczb nie większych niż $2^{55} - 1$. Wykorzystanie 11 bitów na wykładnik sprawia, że w danych typu `Number` mantysa może być podniesiona do potęgi 2^{1023} .

Maksymalna i minimalna wartość danych typu `Number` przechowywane są we właściwościach statycznych klasy `Number` o nazwach `Number.MAX_VALUE` i `Number.MIN_VALUE`.

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

Ceną, jaką trzeba zapłacić za bardzo szeroki zakres wartości typu `Number`, jest precyzja reprezentacji liczb. W danych typu `Number` mantysa jest zapisana na 52 bitach, przez co wartości, których nie da się dokładnie zapisać w tej liczbie bitów (np. ułamek $1/3$) są tylko przybliżeniami. Jeśli w aplikacji wymagana jest bezwzględna precyzja zapisu liczb dziesiętnych, konieczne będzie użycie oprogramowania, które implementuje dziesiętną arytmetykę zmiennopozycyjną (a nie omawianą tutaj binarną arytmetykę zmiennopozycyjną).

Wartości całkowite typu `Number` zapisane są tylko na 52 bitach mantysy. Przy użyciu tych 52 bitów i specjalnego bitu ukrytego można zapisać liczby z zakresu od -9 007 199 254 740 992 (-2^{33}) do 9 007 199 254 740 992 (2^{33}).

Wartość `NaN` jest używana nie tylko jako wartość domyślna zmiennych typu `Number`, lecz również jako wynik wszelkich operacji, które powinny zwracać liczbę, ale jej nie zwracają. Na przykład próba obliczenia pierwiastka kwadratowego z liczby ujemnej da w wyniku `NaN`. Inne specjalne wartości typu `Number` to *dodatnia nieskończoność* oraz *ujemna nieskończoność*.

Uwaga: Wynikiem dzielenia przez 0 jest `NaN` tylko wówczas, gdy dzielnik również jest równy 0. Dzielenie przez 0 daje w wyniku *infinity* (nieskończoność), gdy dzielnik jest dodatni, oraz *-infinity* (-nieskończoność), gdy dzielnik jest ujemny.

Typ danych String

Typ danych String reprezentuje ciąg znaków 16-bitowych. Ciągi znaków są przechowywane wewnętrznie w formacie Unicode UTF-16. Ciągi znaków są wartościami niezmiennymi, podobnie jak w języku oprogramowania Java. Każda operacja na wartości typu String zwraca nową instancję ciągu. Wartość domyślna zmiennej z zadeklarowanym typem String jest `null`. Wartość `null` nie jest tym samym, co pusty łańcuch (`" "`). Wartość `null` oznacza, że w zmiennej nie jest zapisana wartość, a pusty łańcuch oznacza, że ma wartość typu String, która nie zawiera znaków.

Typ danych uint

Wartości typu danych uint są przechowywane wewnętrznie jako 32-bitowe liczby całkowite bez znaku; typ ten obejmuje zbiór liczb całkowitych od 0 do 4 294 967 295 ($2^{32} - 1$) włącznie. Typ uint jest przeznaczony do stosowania w okolicznościach, w których wymagane jest użycie nieujemnych liczb całkowitych. Na przykład dane typu uint muszą być używane do zapisu wartości kolorów pikseli, ponieważ dane typu int zawierają bit znaku, który zakłóca interpretację wartości kolorów. Do przechowywania wartości całkowitych większych niż maksymalna wartość typu uint należy stosować typ danych Number, który pozwala na zapis 53-bitowych liczb całkowitych. Domyślna wartość zmiennych typu uint to 0.

Typ danych void

Typ danych void zawiera tylko jedną wartość, `undefined`. We wcześniejszych wersjach języka ActionScript wartość `undefined` była domyślną wartością instancji klasy Object. W języku ActionScript 3.0 domyślna wartość instancji klasy Object to `null`. Próba przypisania wartości `undefined` instancji klasy Object spowoduje przekonwertowanie wartości na `null`. Wartość `undefined` można przypisywać tylko zmiennym bez typu. Zmienne bez typu są to zmienne, w których deklaracji pominięto wskazanie typu lub użyto wskazania typu w postaci gwiazdki (*). Słowa `void` można użyć tylko jako wskazania typu wartości zwracanej.

Typ danych Object

Typ danych Object jest zdefiniowany przez klasę Object. Klasa Object pełni rolę klasy bazowej dla wszystkich innych definicji klas w języku ActionScript. Wersja typu danych Object w języku ActionScript 3.0 różni się od poprzednich wersji pod trzema względami. Po pierwsze, typ danych Object nie jest już domyślnym typem danych przypisywanym zmiennym bez wskazanego typu. Po drugie typ danych Object nie zawiera już wartości `undefined`, która była domyślną wartością instancji klasy Object. Po trzecie w języku ActionScript 3.0 wartość domyślna instancji klasy Object to `null`.

W poprzednich wersjach języka ActionScript zmiennej bez wskazanego typu automatycznie przypisywany był typ danych Object. Ta zasada nie obowiązuje w języku ActionScript 3.0, w którym wprowadzono koncepcję zmiennej rzeczywiście pozbawionej typu. Zmienne bez wskazanego typu są teraz uznawane za zmienne bez typu. Aby w sposób jednoznaczny zaznaczyć w kodzie, że zmienna jest celowo pozostawiana bez typu, można użyć wskazania typu w postaci symbolu gwiazdki (*), który jest równoważny pominięciu typu. Poniżej przedstawiono dwie równoważne instrukcje. Obie one deklarują zmienną `x` bez typu.

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

Tylko zmienne bez typu mogą przyjmować wartość `undefined`. Próba przypisania wartości `undefined` zmiennej o określonym typie danych spowoduje, że środowisko wykonawcze przekonwertuje wartość `undefined` na wartość domyślną tego typu. W przypadku danych typu Object wartością domyślną jest `null`, co oznacza, że w przy próbie przypisania `undefined` do instancji Object wartość zostanie przekształcona na `null`.

Konwersje typów

O konwersji typu mówimy, gdy wartość jest przekształcana w wartość należącą do innego typu danych. Konwersja typu może być *niejawna* albo *jawna*. Konwersja niejawna, która jest nazywana również *koercją*, jest w niektórych przypadkach przeprowadzana w czasie wykonywania. Na przykład: przypisanie wartości 2 zmiennej typu Boolean powoduje, że wartość 2 zostanie przekonwertowana na wartość typu Boolean `true`, a dopiero potem zostanie przypisana do zmiennej. Konwersja jawna, nazywana także *rzutowaniem*, ma miejsce, gdy kod użytkownika nakazuje kompilatorowi potraktowanie zmiennej jednego typu danych jak gdyby należała ona do innego typu. W przypadku wartości pierwotnych rzutowanie faktycznie konwertuje wartości z jednego typu danych na drugi. Aby rzutować obiekt na inny typ, należy ująć nazwę obiektu w nawiasy i poprzedzić nazwą nowego typu. Na przykład poniższy kod rzutuje wartość Boolean na liczbę całkowitą:

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

Konwersje niejawne

Konwersje niejawne zachodzą w czasie wykonywania w różnych kontekstach:

- W instrukcjach przypisania
- Przy przekazywaniu wartości jako argumentów funkcji
- Gdy wartości są zwracane z funkcji
- W wyrażeniach zawierających niektóre operatory, np. operator dodawania +)

W przypadku typów zdefiniowanych przez użytkownika konwersje niejawne są realizowane pomyślnie, gdy wartość przeznaczona do konwersji jest instancją klasy docelowej lub klasy pochodnej względem klasy docelowej. W wypadku niepowodzenia konwersji niejawnej zgłaszany jest błąd. Na przykład poniższy kod zawiera dwie konwersje niejawne, z których jedna jest realizowana pomyślnie, a druga nie.

```
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.
```

Konwersje niejawne typów pierwotnych są realizowane poprzez wywołanie tych samych wewnętrznych algorytmów konwersji, które używane są przy konwersji jawnej.

Konwersje jawne

Użycie konwersji jawnych, czyli rzutowania, bywa przydatne przy kompilacji w trybie ścisłym, ponieważ niekiedy chcemy uniknąć zgłoszenia błędu kompilacji spowodowanego niezgodnością typów. Dotyczy to sytuacji, w których wiemy z góry, że wartości zostaną prawidłowo, w sposób wymuszony, przekonwertowane w trakcie wykonywania. Na przykład podczas pracy z danymi odebranymi z formularza, chcemy skorzystać z wymuszonej konwersji ciągów znaków na wartości liczbowe. Poniższy kod wywoła błąd podczas kompilacji mimo że działałby prawidłowo w trybie standardowym:

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

Jeśli chcemy pozostać przy trybie ścisłym, ale jednocześnie skorzystać z wymuszonej konwersji ciągu znaków na liczbę, możemy użyć konwersji jawnej:

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

Rzutowanie na typy int, uint i Number

Dowolny typ danych można rzutować na jeden z trzech typów liczbowych: int, uint i Number. Jeśli liczba nie może zostać przekonwertowana z tego samego powodu, wówczas wartość domyślna 0 zostanie przypisana do danych typu int i uint, a wartość domyślna NaN zostanie przypisana do danych typu Number. Podczas konwersji wartości typu Boolean na liczbę wartość true zamieniana jest na 1, a false na 0.

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

Wartości typu String zawierające tylko cyfry mogą być pomyślnie przekonwertowane na jeden z typów liczbowych. Konwersja na typy liczbowe jest też możliwa w przypadku ciągów znaków, które mają postać liczb ujemnych lub zapisanych szesnastkowo (na przykład 0x1A). W procesie konwersji ignorowane są początkowe i końcowe białe spacje w ciągu znaków. Można również rzutować ciągi znaków zawierające zapisy liczb zmiennopozycyjnych, korzystając ze składni Number(). W razie obecności kropki dziesiętnej rzutowanie uint() oraz int() zwraca liczbę całkowitą uzyskaną z obcięciem kropki i znaków następujących po niej. Na przykład następujące ciągi znaków można rzutować na liczby:

```
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
```

Wartości ciągów zawierające znaki nieliczbowe są konwertowane na 0 w przypadku rzutowania int() lub uint() albo NaN w przypadku rzutowania Number(). W procesie konwersji ignorowane są początkowe i końcowe białe spacje, ale jeśli ciąg zawiera liczby rozdzielone białymi spacjami, zwracana jest wartość 0 lub NaN.

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

W języku ActionScript 3.0 funkcja Number() nie obsługuje już liczb ósemkowych (o podstawie 8). Przekazanie ciągu znaków zaczynającego się od zera do funkcji Number() w języku ActionScript 2.0 spowodowało interpretowanie liczby jako ósemkowej i przekonwertowanie na odpowiednią wartość dziesiętną. Natomiast funkcja Number() w języku ActionScript 3.0 ignoruje początkowe zero. Na przykład poniższy kod generuje różne wyniki w zależności od tego, w której wersji kompilatora ActionScript zostanie skompilowany:

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

Rzutowanie nie jest konieczne w przypadku przypisywania wartości jednego typu liczbowego zmiennej innego typu liczbowego. Nawet w trybie ścisłym typy liczbowe są niejawnie konwertowane na inne typy liczbowe. Oznacza to, że w niektórych przypadkach — w razie przekroczenia zakresu typu docelowego — uzyskane wartości mogą być różne od oczekiwanych. Poniższe przykłady dają się prawidłowo skompilować w trybie ścisłym, mimo że wyniki działania niektórych z nich są nieoczekiwane:

```
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
```

W poniższej tabeli zamieszczono podsumowanie wyników rzutowania różnych typów danych na typy Number, int lub uint.

Typ danych lub wartość	Wynik konwersji na Number, int lub uint
Boolean	Wartość <code>true</code> jest zamieniana na 1; w przeciwnym razie 0.
Date	Wewnętrzna reprezentacja obiektu Date, czyli liczba milisekund od północy 1 stycznia 1970 roku UTC.
null	0
Obiekt	Jeśli instancja jest równa <code>null</code> i zostanie przekonwertowana na typ Number, to wartością jest <code>NaN</code> ; w przeciwnym razie 0.
String	Liczba, jeśli łańcuch może zostać przekonwertowany na liczbę; w przeciwnym wypadku <code>NaN</code> — w przypadku konwersji na Number, albo 0 — w przypadku konwersji na int lub uint.
undefined	Przy konwersji na typ Number — wartość <code>NaN</code> ; przy konwersji na typ int lub uint — wartość 0.

Rzutowanie na typ Boolean

Rzutowanie dowolnego liczbowego typu danych (uint, int lub Number) na typ Boolean daje w wyniku `false`, jeśli liczba miała wartość 0, albo `true` w przeciwnym razie. W przypadku typu danych Number konwersja wartości `NaN` również daje w wyniku `false`. Poniższy przykład demonstruje wyniki rzutowania liczb -1, 0 i 1:

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

Wyniki wykonania przykładu wykazują, że spośród trzech liczb tylko wartość 0 jest konwertowana na `false`:

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

Rzutowanie wartości String na typ Boolean daje w wyniku `false`, jeśli ciąg jest równy `null` lub jest pusty(""). W przeciwnym razie wynikiem jest `true`.

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

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

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

Rzutowanie instancji klasy `Object` na typ `Boolean` daje w wyniku `false`, jeśli instancja jest równa `null`; w przeciwnym razie wynikiem jest `true`:

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

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

W trybie ścisłym zmienne typu `Boolean` są traktowane w szczególny sposób, ponieważ dane dowolnego typu danych można przypisać do zmiennej typu `Boolean` bez rzutowania. Niejawna wymuszona konwersja wszystkich typów danych na typ `Boolean` ma miejsce nawet w trybie ścisłym. Innymi słowy, inaczej niż w przypadku niemal wszystkich innych typów danych, rzutowanie na typ `Boolean` nie jest konieczne dla uniknięcia zgłaszania błędów w trybie ścisłym. Wszystkie poniższe przykłady dają się prawidłowo skompilować w trybie ścisłym i zachowują się zgodnie z oczekiwaniami w czasie wykonywania:

```
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
```

W poniższej tabeli zamieszczono podsumowanie wyników rzutowania różnych typów danych na typ `Boolean`.

Typ danych lub wartość	Wynik konwersji na typ Boolean
String	false dla wartości null lub pustego ciągu (" "); true w przeciwnym razie.
null	false
Number, int lub uint	false dla wartości NaN lub 0; true w przeciwnym razie.
Obiekt	false, jeśli instancja jest równa null; true w przeciwnym razie.

Rzutowanie na typ String

Rzutowanie dowolnego typu liczbowego na typ `String` daje w wyniku ciąg znaków reprezentujący liczbę. Rzutowanie wartości `Boolean` na typ danych `String` daje w wyniku ciąg znaków `"true"` dla wartości `true` albo ciąg znaków `"false"` dla wartości `false`.

Rzutowanie instancji klasy `Object` na typ danych `String` daje w wyniku ciąg znaków `"null"`, jeśli instancja jest równa `null`. W przeciwnym wypadku rzutowanie klasy `Object` na typ `String` daje w wyniku ciąg znaków `"[object Object]"`.

Rzutowanie instancji klasy Array na typ String daje w wyniku ciąg znaków zawierający listę elementów tablicy oddzielanych przecinkami. Na przykład poniższe rzutowanie na typ danych String daje w wyniku jeden ciąg zawierający wszystkie trzy elementy tablicy:

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

Rzutowanie instancji klasy Date na typ String daje w wyniku ciąg znaków z zapisem daty zawartej w tej instancji. Poniższy przykład zwraca ciąg znaków zawierający zapis daty pochodzącej z instancji klasy Date (wynik wyświetlany jest dla strefy czasowej PDT):

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

W poniższej tabeli zamieszczono podsumowanie wyników rzutowania różnych typów danych na typ String.

Typ danych lub wartość	Wynik konwersji na typ String
Tablica	łańcuch złożony z samych elementów tablicowych.
Boolean	"true" albo "false"
Date	Ciąg znaków zawierający zapis zawartości obiektu Date.
null	"null"
Number, int lub uint	Ciąg znaków reprezentujący liczbę.
Obiekt	Jeśli instancja jest równa null — ciąg znaków "null"; w przeciwnym razie ciąg znaków "[object Object]".

Składnia

Składnia języka definiuje zestaw reguł, których należy przestrzegać, pisząc kod programu.

Rozróżnianie wielkości liter

W języku ActionScript 3.0 wielkość liter jest rozróżniana. Identyfikatory, które różnią się tylko wielkością liter, są uznawane za różne. Na przykład poniższy kod tworzy dwie różne zmienne:

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

Składnia z kropką

Operator kropki (.) umożliwia dostęp do właściwości i metod obiektu. Korzystając ze składni z kropką, można odwołać się do właściwości lub metody klasy, podając nazwę instancji, kropkę i nazwę tej właściwości lub metody. Weźmy pod uwagę następującą definicję klasy:

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

Składnia z kropką umożliwia dostęp do właściwości `prop1` i do metody `method1()` poprzez odwołanie do nazwy instancji utworzonej w następującym kodzie:

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

Składni z kropką można używać do definiowania pakietów. Kropka umożliwia odwoływanie się do pakietów zagnieżdżonych. Przykładowo, klasa EventDispatcher znajduje się w pakiecie o nazwie events, który jest zagnieżdżony w pakiecie o nazwie flash. Następujące wyrażenie stanowi poprawne odwołanie do pakietu events:

```
flash.events
```

Do klasy EventDispatcher można odwołać się również za pomocą następującego wyrażenia:

```
flash.events.EventDispatcher
```

Składnia z ukośnikiem

Składnia z ukośnikiem nie jest obsługiwana w języku ActionScript 3.0. Składnia taka była używana we wcześniejszych wersjach języka ActionScript do określania ścieżki klipu filmowego lub zmiennej.

Literały

Literał (wartość literalna) to wartość podana bezpośrednio w kodzie. Wszystkie poniższe przykłady są literałami:

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

Literały mogą być także grupowane w literały złożone. Literały tablicowe są ujęte w nawiasy kwadratowe ([]), a poszczególne elementy tablicy są w nich oddzielone przecinkami.

Literał tablicowy może posłużyć do zainicjowania tablicy. Poniższe przykłady ilustrują dwie tablice inicjowane za pomocą literałów tablicowych. Korzystając z instrukcji new, można przekazać literał złożony jako parametr do konstruktora klasy Array. Możliwe jest jednak również przypisywanie wartości literalnych bezpośrednio podczas tworzenia instancji następujących klas podstawowych języka ActionScript: Object, Array, String, Number, int, uint, XML, XMLList i 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];
```

Literały mogą być również stosowane do inicjowania obiektów ogólnych. Obiekt ogólny jest to instancja klasy Object. Literały obiektowe są ujęte w nawiasy sześciennicne ({ }), a właściwości obiektu są w nich oddzielone przecinkami. Każda właściwość jest zadeklarowana za pomocą dwukropka (:), który oddziela nazwę właściwości od jej wartości.

Można utworzyć nowy obiekt za pomocą instrukcji new i przekazać literał obiektowy jako parametr do konstruktora klasy Object. Można także przypisać literał obiektu bezpośrednio do deklarowanej instancji. Poniższy przykład ilustruje dwie alternatywne metody tworzenia nowego obiektu ogólnego i zainicjowania go trzema właściwościami (propA, propB i propC), które otrzymują odpowiednio wartości 1, 2 i 3:

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

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

Więcej tematów Pomocy

[Praca z ciągami znaków](#)

[Korzystanie z wyrażeń regularnych](#)

[Inicjowanie zmiennych XML](#)

Średniki

Instrukcje można kończyć znakiem średnika (;). W wypadku pominięcia średnika kompilator zakłada, że każdy wiersz kodu jest pojedynczą instrukcją. Ponieważ wielu programistów jest przyzwyczajonych do oznaczania końca instrukcji średnikiem, kod będzie bardziej czytelny, jeśli wszystkie instrukcje będą w ten sposób zakończone.

Kończenie instrukcji średnikami umożliwia umieszczenie więcej niż jednej instrukcji w wierszu, co jednak potencjalnie utrudnia czytanie i zrozumienie kodu.

Nawiasy

W języku ActionScript 3.0 nawiasy (()) mają trzy zastosowania. Po pierwsze umożliwiają zmianę kolejności wykonywania operacji w wyrażeniu. Operacje ujęte w nawiasy są zawsze wykonywane jako pierwsze. W poniższym przykładzie użyto nawiasów do zmiany kolejności operacji:

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

Po drugie, można użyć nawiasów z operatorem przecinka (,) w celu wyznaczenia wartości szeregu wyrażeń i zwrócenia wyniku ostatniego wyrażenia, co ilustruje poniższy przykład:

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

Po trzecie, nawiasy umożliwiają przekazanie jednego lub większej liczby parametrów do funkcji lub metody, co ilustruje poniższy przykład, w którym wartość typu String jest przekazywana do funkcji trace():

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

Komentarze

W języku ActionScript 3.0 można stosować dwa typy komentarzy: jednowierszowe i wielowierszowe. Sposób zapisu komentarzy jest podobny, jak w językach C++ i Java. Kompilator ignoruje tekst oznaczony jako komentarz.

Komentarze jednowierszowe rozpoczynają się od dwóch ukośników (//) i kończą wraz z końcem danego wiersza. Na przykład poniższy kod zawiera komentarz jednowierszowy:

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

Komentarze wielowierszowe rozpoczynają się od ukośnika i gwiazdki (/*), a kończą gwiazdką i ukośnikiem (*).

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

Słowa kluczowe i słowa zastrzeżone

Słowa *zastrzeżone* są to słowa, których nie można używać jako identyfikatorów w kodzie, ponieważ są zastrzeżone do użytku przez środowisko ActionScript. Do słów zastrzeżonych należą *leksykalne słowa kluczowe*, które kompilator usuwa z przestrzeni nazw programu. Kompilator zgłosi błąd w wypadku próby użycia leksykalnego słowa kluczowego jako identyfikatora. W poniższej tabeli wymieniono leksykalne słowa kluczowe języka ActionScript 3.0.

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

Istnieje mały zbiór słów kluczowych nazywanych *słowami kluczowymi składni*, które mogą być używane jako identyfikatory, ale mają specjalne znaczenie w określonych kontekstach. W poniższej tabeli wymieniono słowa kluczowe składni języka ActionScript 3.0.

each	get	set	namespace
include	dynamic	final	native
override	static		

Istnieje także kilka identyfikatorów, które czasami są nazywane *słowami zastrzeżonymi na przyszłość*. Identyfikatory te nie są zastrzeżone w języku ActionScript 3.0, ale niektóre z nich mogą być traktowane jako słowa kluczowe przez oprogramowanie, w którym język ActionScript 3.0 jest zaimplementowany. Mimo że użycie wielu z tych słów kluczowych w kodzie jest zwykle możliwe, firma Adobe nie zaleca używania ich, ponieważ mogą uzyskać status słów kluczowych w przyszłych wersjach języka.

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

Stałe

W języku ActionScript 3.0 istnieje instrukcja `const`, której można używać do tworzenia stałych. Stałe są właściwościami o stałej wartości, której nie można zmienić. Stałej można przypisać wartość tylko raz, a przypisanie musi wystąpić w bezpośredniej bliskości deklaracji stałej. Na przykład, jeśli stała jest zadeklarowana jako element klasy, można jej przypisać wartość wyłącznie wewnątrz deklaracji lub wewnątrz konstruktora klasy.

W poniższym kodzie deklarowane są dwie stałe. Wartość pierwszej stałej, `MINIMUM`, jest przypisywana w instrukcji deklaracji. Wartość drugiej stałej, `MAXIMUM`, jest przypisywana w konstruktorze. Należy zwrócić uwagę, że ten przykład daje się skompilować bez błędów wyłącznie w trybie standardowym, ponieważ w trybie ścisłym przypisywanie wartości stałym jest dozwolone tylko podczas inicjowania.

```
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
```

Próba przypisania wartości początkowej stałej w jakikolwiek inny sposób spowoduje zgłoszenie błędu. Na przykład próba ustawienia wartości początkowej stałej `MAXIMUM` poza klasą spowoduje zgłoszenie błędu w trakcie wykonywania.

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

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

Język ActionScript 3.0 definiuje szeroką gamę stałych. W języku ActionScript przyjęto konwencję, zgodnie z którą nazwy stałych zapisane są wielkimi literami, a słowa w nazwach są oddzielone znakami podkreślenia (`_`). Na przykład w definicji klasy `MouseEvent` zastosowano tę konwencję nazewnictwa do stałych, które reprezentują zdarzenia związane z interakcją z myszą:

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

Operatory

Operatory to specjalne funkcje, które przyjmują jeden lub więcej operandów i zwracają wartość. *Operand* jest to wartość — zwykle literał, zmienna lub wyrażenie — na której operuje operator. Na przykład w poniższym kodzie zastosowano operatory dodawania (+) i mnożenia (*) wraz z trzema operandami literalnymi (2, 3 i 4) w celu zwrócenia wartości. Wartość ta, 14, jest następnie używana z operatorem przypisania (=) w celu przypisania jej do zmiennej `sumNumber`.

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

Wyróżnia się operatory jedno-, dwu- i trójargumentowe. Operator *jednoargumentowy* ma jeden operand. Przykładem operatora jednoargumentowego jest operator inkrementacji (++), który przyjmuje tylko jeden operand. Operator *dwuargumentowy* ma dwa operandy. Przykładem operatora dwuargumentowego jest operator dzielenia (/), który przyjmuje dwa operandy. Operator *trójargumentowy* ma trzy operandy. Przykładem operatora trójargumentowego jest operator warunkowy (? :), który przyjmuje trzy operandy.

Niektóre operatory są *przeciążone*, co oznacza, że ich zachowanie zależy od typu lub liczby przekazanych do nich operandów. Przykładem operatora przeciążonego jest operator dodawania (+), który działa różnie w zależności od typu danych operandów. Jeśli oba operandy są liczbami, operator dodawania zwraca sumę wartości. Jeśli oba operandy są ciągami znaków, operator dodawania zwraca ciąg znaków będący konkatencją obu operandów. Poniższy przykładowy kod ilustruje różne zachowanie operatora w zależności od typu operandów:

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

Działanie operatorów może być także uzależnione od liczby przekazanych operandów. Operator odejmowania (-) jest zarówno jedno-, jak i dwuargumentowy. Gdy jest używany z jednym operandem, zmienia znak tego operandu. Gdy jest używany z dwoma operandami, zwraca różnicę między operandami. Poniższy przykład ilustruje użycie operatora odejmowania najpierw jako operatora jednoargumentowego, a następnie jako operatora dwuargumentowego.

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

Priorytet i łączność operatorów

Priorytet i łączność operatorów decydują o kolejności ich przetwarzania. Mimo że dla każdego, kto uczył się arytmetyki, oczywistością jest, że kompilator powinien wykonywać operację mnożenia (*) przed operacją dodawania (+), to kompilator i tak musi otrzymać jednoznaczne instrukcje co do kolejności przetwarzania. Instrukcje takie nazywamy łącznie *priorytetem operatorów*. W języku ActionScript zdefiniowany jest domyślny priorytet operatorów, który programista może zmienić, korzystając z nawiasów (). Na przykład poniższy kod modyfikuje domyślny priorytet (obowiązujący w poprzednim przykładzie), aby wymusić na kompilatorze przetworzenie operatora dodawania przed operatorem mnożenia:

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

Zdarza się, że w jednym wyrażeniu występują dwa lub większa liczba operatorów o tym samym priorytecie. W takich przypadkach kompilator stosuje zasady *łączności* w celu określenia kolejności przetwarzania operatorów. Wszystkie operatory dwuargumentowe, z wyjątkiem operatorów przypisania, są *lewostronne*, co oznacza, że operatory po lewej stronie są przetwarzane przed operatorami po prawej stronie. Operatory przypisania i operator warunkowy (? :) są *prawostronne*, co oznacza, że operatory po prawej stronie są przetwarzane przed operatorami po lewej stronie.

Rozważmy na przykład operator mniejszości(<) i większości(>), które mają ten sam priorytet. Jeśli oba operatory zostaną użyte w tym samym wyrażeniu, operator po lewej stronie zostanie przetworzony jako pierwszy, ponieważ oba operatory są lewostronne. Oznacza to, że następujące dwie instrukcje dadzą ten sam wynik:

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

W pierwszej kolejności przetwarzany jest operator większości, co daje w wyniku wartość `true`, ponieważ operand 3 jest większy od operandu 2. Wartość `true` jest następnie przekazywana do operatora mniejszości wraz z operandem 1. Poniższy kod odzwierciedla ten stan pośredni:

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

Operator mniejszości konwertuje wartość `true` na wartość liczbową 1 i porównuje tę wartość liczbową z drugim operandem 1, co w wyniku daje wartość `false` (ponieważ 1 nie jest mniejsze od 1).

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

Nawiasy umożliwiają domyślną łączność lewostronnej. Można nakazać kompilatorowi, by przetwarzał operator mniejszości w pierwszej kolejności. W tym celu należy ująć ten operator i jego operandy w nawiasy. W poniższym przykładzie w wyniku zastosowania nawiasów generowany jest inny wynik na podstawie tych samych wartości liczbowych:

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

W pierwszej kolejności przetwarzany jest operator mniejszości, co daje w wyniku wartość `false`, ponieważ operand 2 jest nie mniejszy od operandu 1. Wartość `false` jest następnie przekazywana do operatora większości wraz z operandem 3. Poniższy kod odzwierciedla ten stan pośredni:

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

Operator większości konwertuje wartość `false` na wartość liczbową 0 i porównuje tę wartość liczbową z drugim operandem, 3, zwracając w wyniku `true` (wartość 3 jest większa od 0).

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

W poniższej tabeli wymieniono operatory języka ActionScript 3.0 w kolejności malejących priorytetów. Każdy wiersz tabeli zawiera operatory o tym samym priorytecie. Operatory w danym wierszu mają wyższy priorytet niż operatory w niższym wierszu tabeli.

Grupa	Operatory
Podstawowe	[] {x:y} () f(x) new x.y x[y] <></> @ :: ..
Przyrostkowe	x++ x--
Jednoargumentowe	++x --x + - ~ ! delete typeof void
Multiplikatywne	* / %
Addytywne	+ -
Przesunięcia bitowego	<< >> >>>
Relacyjne	< > <= >= as in instanceof is
Równość	== != === !==
Bitowe AND	&
Bitowe XOR	^
Bitowe OR	
Logiczne AND	&&
Logiczne OR	

Grupa	Operatory
Warunkowe	? :
Przypisanie	= *= /= %= += -= <<= >>= >>>= &= ^= =
Przecinek	,

Operatory podstawowe

Do operatorów podstawowych należą operatory używane do tworzenia literałów typu Array i Object, grupowania wyrażeń, wywoływania funkcji, tworzenia instancji klas i odwoływania się do właściwości.

Wszystkie operatory podstawowe wymienione w poniższej tabeli mają równy priorytet: Operatory należące do specyfikacji E4X oznaczono zapisem (E4X).

Operator	Wykonywana operacja
[]	Inicjuje tablicę
{x:y}	Inicjuje obiekt
()	Grupuje wyrażenia
f(x)	Wywołuje funkcję
new	Wywołuje konstruktor
x.y x[y]	Odwołuje się do właściwości
<></>	Inicjuje obiekt XMLList (E4X)
@	Odwołuje się do atrybutu (E4X)
::	Kwalifikuje nazwę (E4X)
..	Odwołuje się do potomnego elementu XML (E4X)

Operatory przyrostkowe

Operatory przyrostkowe mają jeden operand i inkrementują lub dekrementują jego wartość. Mimo że są to operatory jednoargumentowe, zostały sklasyfikowane inaczej niż pozostałe takie operatory, ponieważ mają wyższy priorytet i szczególne działanie. Gdy w większym wyrażeniu występuje operator przyrostkowy, wartość wyrażenia jest zwracana przed przetworzeniem operatora przyrostkowego. Na przykład poniższy kod demonstruje, że wartość wyrażenia xNum++ jest zwracana zanim dojdzie do inkrementacji:

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

Wszystkie operatory przyrostkowe wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
++	Inkrementuje (przyrostkowo)
--	Dekrementuje (przyrostkowo)

Operatory jednoargumentowe

Operatory jednoargumentowe mają jeden operand. Należące do tej grupy operatory inkrementacji (++) i dekrementacji (--) są operatorami *przedrostkowymi*, co oznacza, że występują przed swoim operandem w wyrażeniu. Operatory przedrostkowe tym różnią się od swoich odpowiedników przyrostkowych, że inkrementacja lub dekrementacja jest wykonywana przed zwróceniem wartości wyrażenia. Na przykład poniższy kod demonstruje, że wartość wyrażenia ++xNum jest zwracana po inkrementacji:

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

Wszystkie operatory jednoargumentowe wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
++	Inkrementuje (przedrostkowo)
--	Dekrementuje (przedrostkowo)
+	Jednoargumentowy +
-	Jednoargumentowy - (zmiana znaku)
!	Logiczne NOT
~	Bitowe NOT
delete	Usuwa właściwość
typeof	Zwraca informację o typie
void	Zwraca wartość undefined

Operatory multiplikatywne

Operatory multiplikatywne mają dwa operandy i wykonują mnożenie, dzielenie lub obliczają resztę z dzielenia (modulo).

Wszystkie operatory multiplikatywne wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
*	Mnożenie
/	Dzielenie
%	Modulo

Operatory addytywne

Operatory addytywne wykonują dodawanie lub odejmowanie na dwóch operandach. Wszystkie operatory addytywne wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
+	Dodawanie
-	Odejmowanie

Operatory przesunięcia bitowego

Operatory przesunięcia bitowego mają dwa operandy o przesuwają bity pierwszego operandu o liczbę pozycji określoną przez drugi operand. Wszystkie operatory przesunięcia bitowego wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
<<	Przesunięcie bitowe w lewo
>>	Przesunięcie bitowe w prawo
>>>	Przesunięcie bitowe w prawo bez znaku

Operatory relacyjne

Operatory relacyjne mają dwa operandy, porównują wartości tych operandów i zwracają wartość typu Boolean. Wszystkie operatory relacyjne wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
<	Mniejszy niż
>	Większy niż
<=	Mniejszy lub równy
>=	Większy lub równy
as	Sprawdza typ danych
in	Sprawdza właściwości obiektu
instanceof	Sprawdza łańcuch prototypów
is	Sprawdza typ danych

Operatory równości

Operatory równości mają dwa operandy, porównują wartości tych operandów i zwracają wartość typu Boolean. Wszystkie operatory równości wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
==	Równość
!=	Nierówność
===	Ścisła równość
!==	Ścisła nierówność

Bitowe operatory logiczne

Bitowe operatory logiczne mają dwa operandy i wykonują na nich operacje logiczne na poziomie bitów. Logiczne operatory bitowe mają różne priorytety i zostały wymienione w poniższej tabeli w kolejności malejących priorytetów:

Operator	Wykonywana operacja
&	Bitowe AND
^	Bitowe XOR
	Bitowe OR

Operatory logiczne

Operatory logiczne mają dwa operandy i zwracają wynik typu Boolean. Operatory logiczne mają różne priorytety i zostały wymienione w poniższej tabeli w kolejności malejących priorytetów:

Operator	Wykonywana operacja
&&	Logiczne AND
	Logiczne OR

Operator warunkowy

Operator warunkowy jest trójargumentowy, co oznacza, że ma trzy operandy. Operator ten jest skróconą formą instrukcji warunkowej `if...else`.

Operator	Wykonywana operacja
? :	Warunkowe

Operatory przypisania

Operatory przypisania mają dwa operandy i jednemu z tych operandów przypisują wartość drugiego operandu. Wszystkie operatory przypisania wymienione w poniższej tabeli mają równy priorytet:

Operator	Wykonywana operacja
=	Przypisanie
*=	Przypisanie z mnożeniem
/=	Przypisanie z dzieleniem
%=	Przypisanie z modułu
+=	Przypisanie z dodawaniem
-=	Przypisanie z odejmowaniem
<<=	Przypisanie z przesunięciem bitowym w lewo
>>=	Przypisanie z przesunięciem bitowym w prawo
>>>=	Przypisanie z przesunięciem bitowym w prawo bez znaku
&=	Przypisanie z bitowym AND
^=	Przypisanie z bitowym XOR
=	Przypisanie z bitowym OR

Instrukcje warunkowe

W języku ActionScript 3.0 istnieją trzy podstawowe instrukcje warunkowe służące do sterowania przepływem wykonania programu.

Instrukcja `if..else`

Instrukcja warunkowa `if..else` umożliwia sprawdzenie warunku i wykonanie jednego bloku kodu, jeśli warunek jest spełniony, albo innego bloku kodu, jeśli warunek nie jest spełniony. Poniższy przykładowy kod sprawdza, czy wartość `x` przekracza 20 i wywołuje funkcję `trace()`, jeśli ten warunek jest spełniony, albo wywołuje funkcję `trace()` z innym argumentem, jeśli nie jest spełniony:

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

Jeśli nie chcemy wykonywać alternatywnego bloku kodu, możemy użyć instrukcji `if` bez instrukcji `else`.

`if..else if`

Korzystając z instrukcji warunkowej `if..else if`, można sprawdzać więcej niż jeden warunek. Poniższy kod sprawdza nie tylko, czy wartość `x` przekracza 20, lecz również czy wartość `x` jest ujemna:

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

Jeśli po instrukcji `if` lub `else` następuje tylko jedna instrukcja, nie trzeba jej ujmować w nawiasy sześciennne. W poniższym przykładzie nie użyto nawiasów sześciennych:

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

Firma Adobe zaleca jednak, aby zawsze używać nawiasów sześciennych, ponieważ późniejsze dodanie instrukcji do instrukcji warunkowej bez nawiasów może doprowadzić do nieoczekiwanego zachowania programu. W poniższym przykładzie wartość `positiveNums` jest zwiększana o 1 niezależnie od tego, czy wartością warunku jest `true`:

```
var x:int;  
var positiveNums:int = 0;  
  
if (x > 0)  
    trace("x is positive");  
    positiveNums++;  
  
trace(positiveNums); // 1
```

switch

Instrukcja `switch` jest użyteczna, jeśli mamy kilka możliwych ścieżek wykonania, a wybór jednej z nich zależy od wartości tego samego wyrażenia warunkowego. Oferuje ona funkcjonalność podobną do długiego szeregu instrukcji `if..else if`, ale jest nieco bardziej czytelna. Zamiast sprawdzać wartość logiczną warunku, instrukcja `switch` oblicza wartość wyrażenia i na podstawie wyniku wybiera jeden z bloków kodu do wykonania. Bloki kodu rozpoczynają się od instrukcji `case` i kończą instrukcją `break`. Na przykład poniższa instrukcja `switch` wyświetla nazwę dnia tygodnia na podstawie numeru dnia zwróconego przez metodę `Date.getDay()`:

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

Pętle

Instrukcje pętli umożliwiają wielokrotne wykonywanie określonego bloku kodu z różnymi wartościami lub zmiennymi. Firma Adobe zaleca, aby zawsze ujmować blok kodu w nawiasy sześcienne (`{ }`). Mimo że nawiasy sześcienne można pominąć, jeśli kod zawiera tylko jedną instrukcję, nie zaleca się korzystania z tej możliwości. Powód jest ten sam, co w przypadku instrukcji warunkowych: brak nawiasów zwiększa ryzyko niezamierzonego wyłączenia z bloku kodu instrukcji dopisanych później. Jeśli później dodamy instrukcję, która powinna być uwzględniona w bloku, ale zapomnimy dodać potrzebne nawiasy, instrukcja nie zostanie wykonana w pętli.

for

Pętla `for` umożliwia iteracyjną zmianę wartości zmiennej w ramach określonego przedziału wartości. W instrukcji `for` należy podać trzy wyrażenia: zmienną z przypisaną wartością początkową, instrukcję warunkową, która decyduje o zakończeniu pętli, oraz wyrażenie zmieniające wartość zmiennej w każdej iteracji. Poniższa przykładowa pętla jest wykonywana pięć razy. Zmienna `i` ma na początku wartość 0, a na końcu wartość 4; wynikiem wykonania pętli są liczby od 0 do 4 wyświetlone w osobnych wierszach.

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

for..in

Pętla `for..in` iteracyjnie przegląda właściwości obiektu lub elementy tablicy. Pętli `for..in` można użyć do iteracyjnego przeglądania właściwości obiektu ogólnego (jednak właściwości obiektu nie są przechowywane w żadnym z góry określonym porządku, dlatego mogą pojawiać się w pętli w pozornie przypadkowej kolejności):

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

Możliwe jest także iteracyjne przeglądanie elementów tablicy:

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

Nie można natomiast przeprowadzić iteracji przez właściwości obiektu, jeśli jest to instancja klasy zapieczętowanej (zawiera wbudowane klasy oraz klasy zdefiniowane przez użytkownika). Możliwa jest tylko iteracja właściwości klasy dynamicznej. W przypadku instancji klas dynamicznych iteracja jest możliwa tylko za pośrednictwem właściwości dodawanych dynamicznie.

for each..in

Pętla `for each..in` iteracyjnie przegląda elementy kolekcji, które mogą być znacznikami w obiekcie XML lub XMLList, wartościami zapisanymi we właściwościach obiektu lub elementami tablicy. Poniższy fragment kodu ilustruje możliwość zastosowania pętli `for each..in` do iteracyjnego przeglądania właściwości obiektu ogólnego, jednak, inaczej niż w pętli `for..in`, zmienna sterująca pętli `for each..in` zawiera wartość przechowywaną we właściwości, a nie nazwę właściwości:

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

Możliwe jest iteracyjne przeglądanie obiektu XML lub XMLList, co ilustruje następujący przykład:

```
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
*/
```

Można także iteracyjnie przeglądać elementy tablicy, co zilustrowano poniżej:

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

Nie jest możliwe iteracyjne przeglądanie właściwości obiektu będącego instancją klasy zapieczętowanej. Nawet w instancjach klas dynamicznych nie jest możliwe iteracyjne przeglądanie właściwości ustalonych, czyli zdefiniowanych w ramach definicji klasy.

while

Pętla `while` działa jak instrukcja `if` powtarzana tak długo, jak długo warunek jest spełniony (`true`). Poniższy przykładowy kod generuje te same wyniki, co wcześniejszy przykład z pętlą `for`:

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

Pewną wadą pętli `while` w porównaniu z pętlą `for` jest większe ryzyko niezamierzonego stworzenia pętli nieskończonej `while`. Przykład z pętlą `for` nie zostałby skompilowany bezbłędnie, gdybyśmy pominęli wyrażenie inkrementujące zmienną sterującą, natomiast w przykładzie z pętlą `while` pominięcie tej operacji nie spowoduje błędu kompilatora. Bez wyrażenia inkrementującego `i` pętla jest wykonywana w nieskończoność.

do..while

Pętla `do..while` jest to pętla `while`, w której blok kodu zostanie zawsze wykonany co najmniej jeden raz, ponieważ warunek jest sprawdzany po wykonaniu bloku kodu. Poniżej przedstawiono prosty przykład pętli `do..while`, która wyświetla wynik nawet jeśli warunek jest spełniony:

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

Funkcje

Funkcje są to bloki kodu realizujące konkretne zadania i przystosowane do wielokrotnego wykorzystania w programie. W języku ActionScript 3.0 wyróżnia się dwa typy funkcji: *metody* i *zamknięcia funkcji*. To, czy funkcja jest nazywana metodą, czy zamknięciem funkcji, zależy od kontekstu, w jakim została zdefiniowana. Funkcja jest nazywana metodą, jeśli została zdefiniowana w ramach definicji klasy lub jest powiązana z instancją obiektu. Funkcja jest nazywana zamknięciem funkcji, jeśli jest zdefiniowana w inny sposób.

Funkcje były zawsze bardzo ważnym elementem języka ActionScript. Na przykład w języku ActionScript 1.0 nie istniało słowo `class`, dlatego „klasy” definiowało się za pomocą funkcji-konstruktorów. Mimo że od tamtej pory do języka dodano słowo `class`, pełne zrozumienie istoty funkcji nadal jest ważne dla programistów, którzy chcieliby w pełni wykorzystać potencjał języka. Początkowo może to stanowić pewien problem dla programistów, którzy oczekują, że funkcje w języku ActionScript będą zachowywały się podobnie, jak w języku C++ lub Java. Mimo że podstawowe zasady definiowania i wywoływania funkcji nie powinny sprawiać trudności doświadczonym programistom, pewne bardziej zaawansowane cechy funkcji języka ActionScript wymagają bliższego wyjaśnienia.

Podstawowe pojęcia związane z funkcjami

Wywoływanie funkcji

Wywołanie funkcji ma postać jej identyfikatora z dodanymi nawiasami `()`. Operator nawiasów obejmuje wszelkie parametry, które chcemy przekazać do funkcji. Na przykład: funkcja `trace()` jest funkcją najwyższego poziomu języka ActionScript 3.0:

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

W przypadku wywołania funkcji bez parametrów należy za identyfikatorem funkcji umieścić pustą parę nawiasów. Na przykład do wygenerowania liczby losowej możemy użyć funkcji `Math.random()`, która nie ma parametrów:

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

Definiowanie własnych funkcji

W języku ActionScript 3.0 istnieją dwa sposoby definiowania funkcji: użycie instrukcji funkcyjnej lub wyrażenia funkcyjnego. To, którą technikę wybierzemy, zależy od preferowanego stylu programowania: bardziej statycznego lub bardziej dynamicznego. Programiści, którzy preferują styl statyczny, ścisły, definiują funkcje za pomocą instrukcji funkcyjnych. Użycie wyrażeń funkcyjnych do definiowania funkcji jest uzasadnione specyficznymi potrzebami. Wyrażenia funkcyjne są częściej używane w stylu dynamicznym, w standardowym trybie kompilacji.

Instrukcje funkcyjne

Instrukcje funkcyjne to preferowana technika definiowania funkcji w trybie ścisłym. Instrukcja funkcyjna rozpoczyna się od słowa kluczowego `function`, po którym następują:

- nazwa funkcji;
- lista parametrów oddzielonych przecinkami, ujęta w nawiasy;
- treść funkcji — tzn. kod w języku ActionScript, który ma być wykonywany po wywołaniu funkcji — ujęta w nawiasy sześcienne.

Na przykład poniższy kod tworzy funkcję, w której zdefiniowany jest parametr. Następnie wywołuje funkcję, przekazując jako wartość parametru ciąg znaków "hello":

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

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

Wyrażenia funkcyjne

Drugi sposób deklarowania funkcji polega na użyciu instrukcji przypisania z wyrażeniem funkcyjnym, które czasami bywa nazywane literałem funkcyjnym lub funkcją anonimową. Jest to bardziej rozwlekła metoda zapisu, szeroko stosowana we wcześniejszych wersjach języka ActionScript.

Instrukcja przypisania z wyrażeniem funkcyjnym rozpoczyna się od słowa kluczowego `var`, po którym następują:

- nazwa funkcji;
- operator dwukropka (`:`);
- klasa `Function` wskazująca typ danych;
- operator przypisania (`=`);
- słowo kluczowe `function`;
- lista parametrów oddzielonych przecinkami, ujęta w nawiasy;
- treść funkcji — tzn. kod w języku ActionScript, który ma być wykonywany po wywołaniu funkcji — ujęta w nawiasy sześcienne.

W poniższym przykładzie zadeklarowano funkcję `traceParameter`, korzystając z wyrażenia funkcyjnego:

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

Należy zwrócić uwagę, że w tym przypadku nie określamy nazwy funkcji, tak jak miało to miejsce w instrukcji funkcyjnej. Inną ważną różnicą między instrukcjami funkcyjnymi a wyrażeniami funkcyjnymi polega na tym, że wyrażenie funkcyjne jest wyrażeniem, a nie instrukcją. Oznacza to, że wyrażenie funkcyjne nie może występować w kodzie autonomicznie, natomiast instrukcja funkcyjna — może. Wyrażenie funkcyjne może być użyte tylko w ramach instrukcji, zwykle instrukcji przypisania. Poniższy przykład ilustruje wyrażenie funkcyjne przypisane do elementu tablicy:

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

Wybór między instrukcjami a wyrażeniami

Co do zasady należy używać instrukcji funkcyjnych, chyba że szczególne okoliczności nakazują użycie wyrażenia. Instrukcje funkcyjne są bardziej zwarte i zachowują się podobnie w trybie ścisłym i w trybie standardowym, co odróżnia je od wyrażen funkcyjnych.

Instrukcje funkcyjne są bardziej czytelne niż instrukcje przypisania zawierające wyrażenia funkcyjne. Instrukcje funkcyjne sprawiają, że kod jest bardziej zwarty; są bardziej jednoznaczne niż wyrażenia funkcyjne, które wymagają zastosowania zarówno słowa kluczowego `var`, jak i `function`.

Instrukcje funkcyjne są traktowane podobnie w obu trybach kompilatora, tzn. zarówno w trybie ścisłym, jak i standardowym, dozwolone jest używanie kropki do wywoływania metod zadeklarowanych za pomocą instrukcji funkcyjnych. Nie zawsze dotyczy to metod zadeklarowanych za pomocą metod funkcyjnych. W poniższym przykładzie zadeklarowano klasę `Example` z dwiema metodami: `methodExpression()`, która jest zadeklarowana za pomocą wyrażenia funkcyjnego, oraz `methodStatement()`, która jest zadeklarowana za pomocą instrukcji funkcyjnej. W trybie ścisłym nie jest dozwolone użycie składni z kropką do wywołania metody `methodExpression()`.

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

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

Wyrażenia funkcyjne uważa się za lepiej przystosowane do programowania zorientowanego na zachowanie dynamiczne w czasie wykonywania. Programista, który preferuje pracę w trybie ścisłym, ale musi wywołać metodę zadeklarowaną za pomocą wyrażenia funkcyjnego, ma do wyboru dwie techniki. Po pierwsze, można wywołać metodę, używając nawiasów kwadratowych (`[]`) zamiast kropki (`.`). Poniższe wywołanie metody zostanie pomyślnie skompilowane zarówno w trybie ścisłym, jak i standardowym:

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

Po drugie — można zadeklarować całą klasę jako dynamiczną. Mimo że umożliwi to wywoływanie metody za pomocą operatora kropki, wadą takiego rozwiązania jest rezygnacja z części funkcjonalności trybu ścisłego w odniesieniu do wszystkich instancji tej klasy. Kompilator nie zgłosi na przykład błędu, gdy spróbujemy uzyskać dostęp do niezdefiniowanej właściwości w instancji klasy dynamicznej.

Istnieją pewne okoliczności, w których wyrażenia funkcyjne okazują się użyteczne. Jednym z typowych zastosowań wyrażen funkcyjnych są funkcje używane tylko raz, a następnie likwidowane. Rzadziej spotykane zastosowanie polega na powiązaniu funkcji z właściwością prototypową. Więcej informacji zawiera sekcja „[Obiekt prototypowy](#)” na stronie 119.

Istnieją także dwie subtelne różnice między instrukcjami funkcyjnymi a wyrażeniami funkcyjnymi, które należy wziąć pod uwagę, dokonując wyboru techniki. Pierwsza różnica polega na tym, że wyrażenia funkcyjne nie istnieją jako odrębne obiekty z perspektywy zarządzania pamięcią i czyszczenia pamięci. Innymi słowy, gdy przypisujemy wyrażenie funkcyjne do innego obiektu, np. elementu tablicy lub właściwości obiektu, tworzymy jedyne odwołanie do tego wyrażenia funkcyjnego, jakie istnieje w kodzie. Jeśli tablica lub obiekt, z którym powiązane jest wyrażenie, znajdzie się poza zasięgiem lub z innego powodu przestanie być dostępne, utracimy dostęp do wyrażenia funkcyjnego. Usunięcie tablicy lub obiektu spowoduje, że pamięć zajęta przez wyrażenie funkcyjne zostanie oznaczona jako przeznaczona do wyczyszczenia, a w konsekwencji będzie ją można odzyskać i wykorzystać w innym celu.

Poniższy przykład demonstruje fakt, że po usunięciu właściwości, do której przypisane jest wyrażenie funkcyjne, funkcja przestaje być dostępna. Klasa `Test` jest klasą dynamiczną, a zatem możemy do niej dodać właściwość o nazwie `functionExp` zawierającą wyrażenie funkcyjne. Funkcję `functionExp()` można wywoływać, posługując się kropką, jednak po usunięciu właściwości `functionExp` funkcja ta przestaje być dostępna.

```
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
```

Jeśli natomiast funkcja zostanie najpierw zdefiniowana za pomocą instrukcji funkcyjnej, to będzie istnieć jako autonomiczny obiekt, nawet po usunięciu właściwości, do której została przypisana. Operator `delete` działa tylko na właściwościach obiektów, a zatem nawet jawne usunięcie funkcji `stateFunc()` nie odniesie skutku.

```
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
```

Druga różnica między instrukcjami funkcyjnymi a wyrażeniami funkcyjnymi polega na tym, że instrukcje funkcyjne istnieją w całym zasięgu, w jakim są zdefiniowane, również we wszystkich instrukcjach poprzedzających taką instrukcję funkcyjną. Z kolei wyrażenia funkcyjne są zdefiniowane tylko w instrukcjach następujących po nich. Na przykład poniższy kod pomyślnie wywołuje funkcję `scopeTest()` zanim została ona zdefiniowana:

```
statementTest(); // statementTest
```

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

Wyrażenia funkcyjne są niedostępne, zanim nie zostaną zdefiniowane, a zatem następujący kod spowoduje zgłoszenie błędu w czasie wykonywania:

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

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

Zwracanie wartości z funkcji

W celu zwrócenia wartości z funkcji należy użyć instrukcji `return`, po której powinno nastąpić wyrażenie lub wartość literalna przeznaczona do zwrócenia. Poniższy kod przykładowy zwraca wyrażenie reprezentujące parametr:

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

Należy zwrócić uwagę, że instrukcja `return` kończy wykonywanie funkcji, a zatem instrukcje następujące po instrukcji `return` nie zostaną wykonane, co ilustruje poniższy przykład:

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

W trybie ścisłym, jeśli zadeklarowano typ wartości zwracanej przez funkcję, to funkcja musi zwracać wartość odpowiedniego typu. Na przykład poniższy kod spowoduje zgłoszenie błędu w trybie ścisłym, ponieważ nie zwraca właściwej wartości:

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

Funkcje zagnieżdżone

Istnieje możliwość zagnieżdżania funkcji, tzn. deklarowania funkcji wewnątrz innych funkcji. Funkcja zagnieżdżona jest dostępna tylko wewnątrz swojej funkcji nadrzędnej, chyba że odwołanie do funkcji zostanie przekazane do kodu zewnętrznego. W poniższym przykładowym kodzie zadeklarowano dwie funkcje zagnieżdżone wewnątrz funkcji `getNameAndVersion()`:

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

Funkcje zagnieżdżone są przekazywane do kodu zewnętrznego jako zamknięcia funkcji, tzn. zachowują wszelkie definicje pozostające w zasięgu, w którym funkcja była zdefiniowana. Więcej informacji zawiera sekcja „[Zasięg funkcji](#)” na stronie 87.

Parametry funkcji

Język ActionScript 3.0 oferuje w odniesieniu do parametrów funkcji pewne rozwiązania, które mogą stanowić nowość dla programistów, którzy dotychczas nie zetknęli się z tym językiem. Mimo że koncepcja przekazywania parametrów przez wartość lub przez odwołanie (referencję) powinna być znana większości programistów, to wielu czytelników prawdopodobnie nie zetknęło się wcześniej z obiektem `arguments` oraz parametrem `...` (rest).

Przekazywanie argumentów przez wartość lub przez odwołanie

W wielu językach programowania bardzo istotne jest rozróżnienie między przekazywaniem argumentów przez wartość a przekazywaniem ich przez odwołanie. Różnica ta wpływa na konstrukcję kodu.

Przekazanie przez wartość oznacza, że wartość argumentu jest kopiowana do zmiennej lokalnej używanej wewnątrz funkcji. Przekazanie przez odwołanie oznacza przekazanie wyłącznie odwołania (wskaźnika) do argumentu, a nie jego faktycznej wartości. Argument nie jest kopiowany. Tworzone jest natomiast odwołanie do zmiennej przekazanej jako argument i odwołanie to jest przypisywane zmiennej lokalnej używanej wewnątrz funkcji. Jako odwołanie do zmiennej znajdującej się poza funkcją, zmienna lokalna umożliwia zmianę wartości pierwotnej zmiennej.

W języku ActionScript 3.0 wszystkie argumenty są przekazywane przez odwołanie, ponieważ wszystkie wartości są przechowywane jako obiekty. Jednak obiekty należące do pierwotnych typów danych, czyli Boolean, Number, int, uint oraz String, mają specjalne operatory sprawiające, że zachowują się, jak gdyby były przekazywane przez wartość. Poniższy przykładowy kod tworzy funkcję o nazwie `passPrimitives()`, która definiuje dwa parametry o nazwach `xParam` i `yParam`, oba typu int. Parametry te są podobne do zmiennych lokalnych zadeklarowanych w treści funkcji `passPrimitives()`. Gdy funkcja zostaje wywołana z argumentami `xValue` i `yValue`, parametry `xParam` i `yParam` są inicjowane odwołaniami do obiektów int reprezentowanych przez argumenty `xValue` i `yValue`. Ponieważ argumenty należą do typu pierwotnego, zachowują się tak, jak gdyby były przekazywane przez wartość. Mimo że parametry `xParam` i `yParam` początkowo zawierały tylko odwołania do obiektów `xValue` i `yValue`, wszelkie modyfikacje zmiennych w treści funkcji powodują utworzenie nowych kopii wartości w pamięci.

```
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
```

Wewnątrz funkcji `passPrimitives()` wartości `xParam` i `yParam` są inkrementowane, ale nie wpływa to na wartość `xValue` i `yValue`, czego dowodzi wynik ostatniej instrukcji `trace`. Kod działałby tak samo nawet wtedy, gdyby parametry miały nazwy identyczne z nazwami zmiennych `xValue` i `yValue`, ponieważ nazwy `xValue` i `yValue` wewnątrz funkcji wskazywałyby na nowe miejsca w pamięci, autonomiczne względem zmiennych o tych samych nazwach poza funkcją.

Wszystkie pozostałe obiekty — tj. obiekty nienależące do pierwotnych typów danych — są zawsze przekazywane przez odwołanie, co umożliwia zmianę wartości oryginalnej zmiennej. Poniższy przykładowy kod tworzy obiekt o nazwie `objVar` z dwiema właściwościami: `x` i `y`. Obiekt ten jest przekazywany jako argument do funkcji `passByRef()`. Ponieważ obiekt nie należy do typu pierwotnego, jest nie tylko przekazywany przez odwołanie, lecz również pozostaje odwołaniem. Oznacza to, że modyfikacja parametrów wewnątrz funkcji wpłynie na właściwości obiektów poza funkcją.

```
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
```

Parametr `objParam` odwołuje się do tego samego obiektu, co zmienna globalna `objVar`. Jak wynika z wyników instrukcji `trace` w przykładowym kodzie, zmiany właściwości `x` i `y` obiektu `objParam` są odzwierciedlone w obiekcie `objVar`.

Domysłne wartości parametrów

W języku ActionScript 3.0 można zadeklarować *domysłne właściwości parametrów* dla funkcji. Jeśli w wywołaniu funkcji mającej wartości domyślne parametrów zostaną pominięte takie parametry, przyjmowane są wartości podane w definicji funkcji. Wszystkie parametry z wartościami domyślnymi muszą być umieszczone na końcu listy parametrów. Przypisywane wartości domyślne muszą być stałymi znanymi w czasie kompilacji. Istnienie wartości domyślnej parametru powoduje, że parametr ten automatycznie staje się *parametrem opcjonalnym*. Parametr bez wartości jest uznawany za *parametr wymagany*.

Poniższy przykładowy kod tworzy funkcję z trzema parametrami, z których dwa mają wartości domyślne. Gdy funkcja zostanie wywołana tylko z jednym parametrem, dla pozostałych dwóch parametrów przyjęte zostaną wartości domyślne.

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

Obiekt arguments

Wewnątrz funkcji, do której przekazywane są parametry, można skorzystać z obiektu `arguments`, aby uzyskać dostęp do informacji o przekazanych parametrach. Oto niektóre ważne aspekty użycia obiektu `arguments`:

- Obiekt `arguments` jest tablicą zawierającą wszystkie parametry przekazane do funkcji.
- Właściwość `arguments.length` zawiera liczbę argumentów przekazanych do funkcji.
- Właściwość `arguments.callee` zawiera odwołanie do samej funkcji, co bywa przydatne przy rekurencyjnych wywołaniach wyrażeń funkcyjnych.

Uwaga: Obiekt `arguments` nie jest dostępny, jeśli którykolwiek z parametrów ma nazwę `arguments` lub jeśli używany jest parametr ... (rest).

Jeśli do obiektu `arguments` istnieją odwołania w treści funkcji, wówczas w języku ActionScript 3.0 dozwolone jest wywoływanie funkcji z większą liczbą parametrów niż podana w definicji funkcji. Jednak w trybie dokładnym kompilator zgłosi błąd, jeśli liczba parametrów nie będzie zgodna z liczbą parametrów wymaganych (i ewentualnie parametrów opcjonalnych). Korzystając z obiektu `arguments` jak z tablicy, można odwołać się do dowolnego parametru przekazanego do funkcji, niezależnie od tego, czy został wymieniony w definicji funkcji. Poniższy przykład, który daje się skompilować bezbłędnie tylko w trybie standardowym, ilustruje użycie tablicy `arguments` oraz właściwości `arguments.length` do wyświetlenia wszystkich parametrów przekazanych do funkcji

```
traceArgArray():

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

traceArgArray(1, 2, 3);

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

Właściwość `arguments.callee` jest często używana w funkcjach anonimowych do uzyskania rekurencji. Takie rozwiązanie pozwala na bardziej elastyczne programowanie. Jeśli w toku prac nad programem nazwa funkcji rekurencyjnej ulegnie zmianie, nie będzie konieczne modyfikowanie wywołania rekurencyjnego w treści funkcji — wystarczy zamiast nazwy użyć odwołania `arguments.callee`. Właściwość `arguments.callee` została użyta w poniższym wyrażeniu funkcyjnym, umożliwiając rekurencję:

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

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

Jeśli deklaracja funkcji zawiera parametr ... (rest), to obiekt `arguments` nie jest dostępny. Do parametrów należy się wówczas odwoływać za pomocą nazw, pod jakimi zostały zadeklarowane.

Należy również unikać używania ciągu znaków "arguments" w charakterze nazwy parametru, ponieważ taki parametr przesłoni obiekt `arguments`. Przykładowo, jeśli funkcja `traceArgArray()` zostanie zmodyfikowana poprzez dodanie parametru `arguments`, odwołania do obiektu `arguments` w treści funkcji będą teraz dotyczyły tego parametru, a nie obiektu `arguments`. Poniższy kod nie spowoduje wyświetlenia żadnych wyników.

```
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
```

Obiekt `arguments` w poprzednich wersjach języka ActionScript zawierał też właściwość o nazwie `caller` będącą odwołaniem do funkcji, która wywołała bieżącą funkcję. Właściwość `caller` nie występuje w języku ActionScript 3.0, ale jeśli musimy odwołać się do funkcji wywołującej, możemy ją tak zmodyfikować, aby przekazywała odwołanie do samej siebie jako dodatkowy parametr.

Parametr ... (rest)

W języku ActionScript 3.0 wprowadzono możliwość zadeklarowania nowego rodzaju parametru o nazwie ... (rest). Umożliwia on podanie jako argumentu jednej tablicy zawierającej dowolną liczbę argumentów oddzielonych przecinkami. Parametr może mieć dowolną nazwę niebędącą słowem zastrzeżonym. Deklaracja takiego parametru musi być ostatnia na liście deklaracji. Użycie tego parametru powoduje, że obiekt `arguments` staje się niedostępny. Mimo że parametr... (rest) oferuje tę samą funkcjonalność, co tablica `arguments` i właściwość `arguments.length`, nie oferuje funkcjonalności właściwości `arguments.callee`. Decydując się na użycie parametru ... (rest), należy mieć świadomość, że uniemożliwi to korzystanie z właściwości `arguments.callee`.

W poniższym przykładzie funkcja `traceArgArray()` została zmodyfikowana w taki sposób, że zamiast obiektu `arguments` używany jest w niej parametr ... (rest):

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

```
traceArgArray(1, 2, 3);
```

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

Parametru ... (rest) można również używać z innymi parametrami, o ile będzie on ostatni na liście parametrów. W poniższym przykładzie funkcja `traceArgArray()` została zmodyfikowana w taki sposób, że pierwszy parametr, `x`, jest typu `int`, a drugi parametr jest parametrem ... (rest). Wyniki nie zawierają pierwszej wartości, ponieważ pierwszy parametr nie należy już do tablicy utworzonej przez parametr ... (rest).

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

```
traceArgArray(1, 2, 3);
```

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

Funkcje jako obiekty

W języku ActionScript 3.0 funkcje są obiektami. Utworzenie funkcji jest równoznaczne z utworzeniem obiektu, który nie tylko można przekazywać jako parametr do innej funkcji, lecz który może mieć również własne właściwości i metody.

Funkcje jako argumenty innych funkcji są przekazywane przez odwołanie, a nie przez wartość. Przekazując funkcję jako argument, podajemy tylko identyfikator, bez nawiasów używanych w wywołaniu metody. W poniższym przykładowym kodzie funkcja o nazwie `clickListener()` jest przekazywana jako argument do metody `addEventListener()`:

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

Mimo że programistom, którzy po raz pierwszy stykają się z językiem ActionScript, często wydaje się to dziwne, funkcje mogą mieć właściwości i metody — tak samo, jak każdy inny obiekt. W istocie każda funkcja ma właściwość o nazwie `length` przeznaczoną tylko do odczytu, w której przechowywana jest liczba parametrów zdefiniowanych dla funkcji. Należy odróżnić tę właściwość od właściwości `arguments.length`, która zawiera liczbę argumentów faktycznie przekazanych do funkcji. Jak pamiętamy, w języku ActionScript liczba argumentów przekazanych do funkcji może być większa od liczby parametrów zdefiniowanych dla tej funkcji. Poniższy przykład, który można skompilować bezbłędnie tylko w trybie standardowym (gdyż w trybie ścisłym wymagana jest dokładna zgodność liczby przekazanych argumentów z liczbą zdefiniowanych parametrów), ilustruje różnicę między tymi dwiema właściwościami:

```
// 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 */
```

W trybie standardowym możliwe jest definiowanie własnych właściwości funkcji poza treścią funkcji. Właściwości funkcji mogą pełnić rolę właściwości quasi-statycznych, pozwalając na zapisanie stanu zmiennej związanej z funkcją. Na przykład można w ten sposób rejestrować liczbę wywołań funkcji. Taka funkcjonalność byłaby przydatna w grze, w której chcemy rejestrować każde użycie określonego polecenia przez użytkownika. Jednak ten sam efekt uzyskalibyśmy, stosując właściwość klasy statycznej. Poniższy kod przykładowy, który da się skompilować bez błędów tylko w trybie standardowym (ponieważ tryb ścisły nie zezwala na dodawanie właściwości dynamicznych do funkcji) tworzy właściwość funkcji poza jej deklaracją i inkrementuje tę właściwość przy każdym wywołaniu funkcji:

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

someFunction.counter = 0;

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

Zasięg funkcji

Zasięg funkcji określa nie tylko to, w których miejscach kodu funkcję można wywoływać, lecz także do jakich definicji funkcja ma dostęp. Identyfikatory funkcji podlegają tym samym regułom zasięgu, co identyfikatory zmiennych. Funkcja zadeklarowana w zasięgu globalnym jest dostępna w całym kodzie. Na przykład w języku ActionScript 3.0 zdefiniowane są funkcje globalne, takie jak `isNaN()` i `parseInt()`, z których można korzystać w dowolnym miejscu kodu. Funkcja zagnieżdżona — czyli funkcja zadeklarowana wewnątrz innej funkcji — może być używana w dowolnym miejscu funkcji, w którym została zadeklarowana.

Łańcuch zasięgów

Za każdym razem, gdy rozpoczyna się wykonywanie funkcji, tworzona jest pewna liczba obiektów i właściwości. Po pierwsze, tworzony jest specjalny obiekt nazywany *obiektem aktywacji*, w którym przechowywane są parametry i wszelkie zmienne lokalne lub funkcje zadeklarowane w treści funkcji. Obiekt aktywacji nie jest dostępny bezpośrednio, ponieważ stanowi mechanizm wewnętrzny. Po drugie: tworzony jest *łańcuch zasięgów* zawierający uporządkowaną listę obiektów, w których poszukiwane są deklaracje identyfikatorów. Każda wykonywana funkcja ma swój łańcuch zasięgu przechowywany we właściwości wewnętrznej. W przypadku funkcji zagnieżdżonych łańcuch zasięgu rozpoczyna się od jej własnego obiektu aktywacji, po którym następuje obiekt aktywacji funkcji nadrzędnej. Łańcuch jest kontynuowany na tej zasadzie, aż do osiągnięcia obiektu globalnego. Obiekt globalny jest tworzony z chwilą rozpoczęcia wykonywania programu w języku ActionScript i zawiera wszystkie zmienne i funkcje globalne.

Zamknięcia funkcji

Zamknięcie funkcji to obiekt zawierający obraz stanu funkcji i jej *środowiska leksykalnego*. Środowisko leksykalne funkcji zawiera wszystkie zmienne, właściwości, metody i obiekty w łańcuchu zasięgów funkcji oraz ich wartości. Zamknięcia funkcji tworzone są za każdym razem, gdy funkcja jest wykonywana w oderwaniu od obiektu lub klasy. Fakt, że zamknięcia funkcji zachowują zasięg, w którym zostały zdefiniowane, prowadzi do interesujących skutków przekazania funkcji jako argumentu lub wartości zwracanej do innego zasięgu.

Przykładowo, w poniższym kodzie tworzone są dwie funkcje: `foo()`, która zwraca funkcję zagnieżdżoną o nazwie `rectArea()`, obliczającą pole powierzchni prostokąta, oraz `bar()`, która wywołuje funkcję `foo()` i zapisuje zwrócone zamknięcie funkcji w zmiennej o nazwie `myProduct`. Mimo że funkcja `bar()` definiuje własną zmienną lokalną `x` (o wartości 2), w wywołanym zamknięciu funkcji `myProduct()` zachowana jest zmienna `x` (o wartości 40) zdefiniowana w funkcji `foo()`. Funkcja `bar()` zwraca zatem wartość 160 zamiast wartości 8.

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

Metody zachowują się podobnie, tj. zachowują informacje o środowisku leksykalnym, w którym zostały utworzone. Ta cecha jest najbardziej zauważalna, gdy metoda zostanie wyodrębniona ze swojej instancji, stając się metodą powiązaną. Zasadniczą różnicą między zamknięciem funkcji a metodą powiązaną jest fakt, że wartość słowa kluczowego `this` w metodzie powiązanej jest zawsze odwołaniem do instancji, do której ta metoda pierwotnie należała, natomiast w zamknięciu funkcji wartość słowa kluczowego `this` może ulegać zmianie.

Rozdział 4: Programowanie zorientowane obiektowo w języku ActionScript

Wprowadzenie do programowania zorientowanego obiektowo

Programowanie zorientowane obiektowo jest sposobem porządkowania kodu w programie poprzez grupowanie kodu w obiekty. Termin *obiekt* w tym sensie oznacza pojedynczy element, który zawiera informacje (wartości danych) i funkcje. Jeśli programowanie zorientowane obiektowo jest wykorzystywane w celu uporządkowania programu, należy pogrupować ze sobą określone typy informacji lub operacje skojarzone z tymi informacjami. Na przykład: można informacje muzyczne, takie jak tytuły albumów, tytuły utworów i nazwy wykonawców pogrupować z funkcją, taką jak „dodaj utwór do listy odtwarzania” lub „odtwórz wszystkie utwory tego wykonawcy”. Elementy te można połączyć w jeden obiekt (na przykład „Album” albo „UtworMuzyczny”). Łączenie wartości i funkcji w grupy ma kilka zalet. Najważniejszą zaletą jest to, że wystarczy użyć jednej zmiennej zamiast wielu. Ponadto powiązane ze sobą funkcje są przechowywane razem. Łączenie informacji i funkcji umożliwia zdefiniowanie takiej struktury programu, która lepiej odzwierciedla struktury istniejące w świecie rzeczywistym.

Klasy

Klasa jest abstrakcyjną reprezentacją obiektu. W klasie zapisane są informacje o typach danych, jakie obiekt może przechowywać, oraz o możliwych zachowaniach obiektu. Użyteczność takiej abstrakcyjnej reprezentacji nie zawsze jest oczywista, gdy piszemy małe skrypty zawierające zaledwie kilka obiektów wchodzących we wzajemne interakcje. Jednak w miarę rozbudowywania programu wzrasta liczba obiektów, które wymagają zarządzania. W takim przypadku klasy zapewniają lepszą kontrolę nad tworzeniem obiektów oraz nad interakcjami między obiektami.

W języku ActionScript 1.0 programiści mieli do dyspozycji obiekty Function, które pozwalały im na budowanie struktur programistycznych przypominających klasy. W języku ActionScript 2.0 wprowadzono formalną obsługę klas oraz takie słowa kluczowe, jak `class` oraz `extends`. Język ActionScript 3.0 nie tylko obsługuje słowa kluczowe wprowadzone w języku ActionScript 2.0, ale również zapewnia nowe możliwości. Na przykład: język ActionScript 3.0 zawiera udoskonalone funkcje kontroli dostępu w postaci atrybutów `protected` i `internal`. Zapewnia również lepszą kontrolę nad dziedziczeniem dzięki słowom kluczowym `final` i `override`.

Programistom, którzy tworzyli klasy w językach, takich jak Java, C++ lub C#, język ActionScript będzie bardzo przypominał te języki. Język ActionScript wykorzystuje wiele takich samych słów kluczowych i nazw atrybutów, na przykład: `class`, `extends` i `public`.

Uwaga: W dokumentacji Adobe ActionScript termin *właściwość* oznacza dowolny element obiektu lub klasy, w tym zmienne, stałe i metody. Ponadto, mimo że terminy „zdefiniowany w klasie” i „statyczny” są często używane zamiennie, tutaj oznaczają dwa różne pojęcia. Na przykład: sformułowanie „właściwości klasy” oznacza wszystkie elementy klasy, a nie tylko jej elementy statyczne.

Definicje klas

Składnia definicji klas w języku ActionScript 3.0 jest podobna do obowiązującej w języku ActionScript 2.0. Prawidłowa definicja klasy składa się ze słowa kluczowego `class`, po którym następuje nazwa klasy. Po nazwie klasy następuje treść klasy, ujęta w nawiasy sześcienne (`{}`). Na przykład poniższy kod tworzy klasę o nazwie `Shape` zawierającą zmienną o nazwie `visible`:

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

Ważną zmianą reguł składni dotyczy definicji klas wewnątrz pakietów. W języku ActionScript 2.0 deklaracja klasy znajdującej się w pakiecie musi zawierać nazwę pakietu. W języku ActionScript 3.0, w którym wprowadzono instrukcję `package`, nazwa pakietu musi znajdować się w deklaracji pakietu, a nie w deklaracji klasy. Poniżej przedstawiono deklaracje klas ilustrujące sposób zdefiniowania klasy `BitmapData` (wchodzącej w skład pakietu `flash.display`) w językach ActionScript 2.0 oraz ActionScript 3.0:

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

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

Atrybuty klas

Język ActionScript 3.0 dopuszcza modyfikowanie definicji klas przy użyciu czterech atrybutów:

Atrybut	Definicja
<code>dynamic</code>	Umożliwia dodawanie właściwości do instancji w czasie wykonywania.
<code>final</code>	Zakazuje rozszerzania klasy przez inną klasę.
<code>internal</code> (atrybut domyślny)	Powoduje, że klasa jest widoczna wewnątrz bieżącego pakietu.
<code>public</code>	Powoduje, że klasa jest widoczna wszędzie.

Aby użyć każdego z tych atrybutów, z wyjątkiem `internal`, należy jawnie przypisać atrybut w definicji. Na przykład brak atrybutu `dynamic` w definicji klasy spowoduje, że w czasie wykonywania nie będzie można dodać właściwości do instancji tej klasy. Atrybut przypisuje się jawnie, umieszczając go na początku definicji klasy, co ilustruje poniższy kod:

```
dynamic class Shape {}
```

Niektórzy czytelnicy być może zwrócili uwagę, że na liście brak atrybutu `abstract`. W języku ActionScript 3.0 klasy abstrakcyjne nie są obsługiwane. Ponadto lista nie zawiera atrybutów `private` ani `protected`. Te atrybuty mają znaczenie wyłącznie wewnątrz definicji klasy i nie mogą być stosowane do samych klas. Jeśli klasa nie powinna być widoczna publicznie poza pakietem, należy umieścić ją w pakiecie i oznaczyć atrybutem `internal`. Można również pominąć atrybuty `internal` i `public`, a wówczas kompilator automatycznie dodaje atrybut `internal`. Można również zdefiniować klasę, która będzie widoczna tylko w pliku źródłowym, w którym została zdefiniowana. Należy umieścić taką klasę u dołu pliku źródłowego, poniżej zamykającego nawiasu sześciennej definicji pakietu.

Treść klasy

Treść klasy jest ujęta w nawiasy klamrowe. Definiuje zmienne, stałe oraz metodę klasy. Poniższy przykład przedstawia deklarację klasy `Accessibility` w języku ActionScript 3.0:

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

Wewnątrz treści klasy można także definiować przestrzenie nazw. Poniższy przykład ilustruje definiowanie przestrzeni nazw w treści klasy i wykorzystanie tej przestrzeni jako atrybutu metody klasy:

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

Język ActionScript 3.0 umożliwia uwzględnienie w treści klasy nie tylko definicji, lecz także instrukcji. Instrukcje w treści klasy, ale znajdujące się poza definicją klasy, są wykonywane dokładnie raz. Do wykonania dochodzi wówczas, gdy definicja klasy zostanie napotkana i gdy zostanie utworzony skojarzony obiekt klasy. Poniższy przykład zawiera wywołanie funkcji zewnętrznej `hello()` oraz instrukcję `trace`, która wyświetla komunikat potwierdzający zdefiniowanie klasy:

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

W języku ActionScript 3.0 dozwolone jest zdefiniowanie właściwości statycznej i właściwości instancji o tej samej nazwie w treści tej samej klasy. Poniższy przykładowy kod deklaruje zmienną statyczną o nazwie `message` oraz zmienną instancji o tej samej nazwie:

```
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
```

Atrybuty właściwości klas

W omówieniu modelu obiektów języka ActionScript termin *właściwość* oznacza każdy element, który może należeć do klasy, w tym zmienne, stałe i metody. Jednak w skorowidzu Adobe ActionScript 3.0 dla platformy Adobe Flash ten termin jest stosowany w węższym zakresie. W tym kontekście właściwość obejmuje tylko elementy klasy, które są zmiennymi lub które są zdefiniowane przez metodę pobierania/ustawiania. W języku ActionScript 3.0 istnieje zbiór atrybutów, które mogą być używane z dowolną właściwością klasy. W następującej tabeli wymieniono te atrybuty:

Atrybut	Definicja
<code>internal</code> (atrybut domyślny)	Powoduje, że właściwość jest widoczna wewnątrz tego samego pakietu.
<code>private</code>	Powoduje, że właściwość jest widoczna wewnątrz tej samej klasy.
<code>protected</code>	Powoduje, że właściwość jest widoczna wewnątrz tej samej klasy i jej klas potomnych.
<code>public</code>	Powoduje, że właściwość jest widoczna wszędzie.
<code>static</code>	Określa, że właściwość należy do klasy, a nie do instancji klasy.
<code>UserDefinedNamespace</code>	Niestandardowa przestrzeń nazw zdefiniowana przez użytkownika

Atrybuty przestrzeni nazw sterujące dostępem

Język ActionScript 3.0 udostępnia cztery atrybuty specjalne służące do sterowania dostępem do właściwości zdefiniowanych wewnątrz klasy: `public`, `private`, `protected` i `internal`.

Atrybut `public` powoduje, że właściwość będzie widoczna w każdym miejscu skryptu. Na przykład, aby udostępnić metodę dla kodu poza jej pakietem, należy zadeklarować ją z atrybutem `public`. Zasada ta dotyczy wszystkich właściwości, niezależnie od tego, czy są zadeklarowane za pomocą słowa kluczowego `var`, `const`, czy `function`.

Atrybut `private` powoduje, że właściwość jest widoczna tylko dla kodu wywołującego w klasie definiującej tę właściwość. Atrybut `private` działa zatem inaczej niż w języku ActionScript 2.0, gdzie podklasy miały dostęp do klas prywatnych nadklas. Inna ważna różnica w działaniu dotyczy dostępu w czasie wykonywania. W języku ActionScript 2.0 słowo kluczowe `private` uniemożliwiało dostęp wyłącznie w trybie kompilacji i dawało się łatwo obejść w czasie wykonywania. W języku ActionScript 3.0 nie jest to już możliwe. Właściwości oznaczone `private` są niedostępne zarówno w czasie kompilacji, jak i w czasie wykonywania.

Na przykład następujący kod tworzy prostą klasę o nazwie `PrivateExample`, zawierającą jedną zmienną prywatną, a potem próbuje uzyskać dostęp do tej zmiennej prywatnej spoza klasy.

```
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.
```

W języku ActionScript 3.0 próba dostępu do właściwości prywatnej za pomocą kropki (`myExample.privVar`) powoduje błąd kompilacji w trybie ścisłym. W trybie standardowym błąd zostanie zgłoszony podczas wykonywania, podobnie jak przy próbie użycia operatora dostępu do właściwości (`myExample["privVar"]`).

W poniższej tabeli zamieszczono podsumowanie wyników, jakie da próba dostępu do właściwości prywatnej należącej do klasy zapieczętowanej (niedynamicznej):

	Tryb ścisły	Tryb standardowy
kropka (.)	błąd w czasie kompilacji	błąd w czasie wykonywania
nawiasy kwadratowe ([])	błąd w czasie wykonywania	błąd w czasie wykonywania

W klasach zadeklarowanych z atrybutem `dynamic` próba dostępu do zmiennej prywatnej nie powoduje błędu w czasie wykonania. Zmienna nie jest widoczna, dlatego zwracana jest wartość `undefined`. Jednak próba użycia kropki w trybie ścisłym spowoduje zgłoszenie błędu w czasie kompilacji. Poniższy przykład jest prawie identyczny z poprzednim, z tym że klasa `PrivateExample` jest zadeklarowana jako dynamiczna:

```
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
```

Klasy dynamiczne z reguły zwracają wartość `undefined`, a nie powodują zgłoszenia błędu przy próbie dostępu do właściwości prywatnej z kodu zewnętrznego względem klasy. Zgodnie z poniższą tabelą, błąd jest generowany tylko przy próbie dostępu do właściwości prywatnej za pomocą kropki w trybie ścisłym:

	Tryb ścisły	Tryb standardowy
kropka (.)	błąd w czasie kompilacji	undefined
nawiasy kwadratowe ([])	undefined	undefined

Atrybut `protected`, będący nowym elementem języka ActionScript 3.0, powoduje, że właściwość jest widoczna dla kodu wywołującego wewnątrz jej własnej klasy lub podklas. Innymi słowy, właściwość z atrybutem `protected` (chroniona) jest dostępna dla własnej klasy i dla wszystkich klas znajdujących się poniżej w hierarchii dziedziczenia. Zasada ta obowiązuje niezależnie od tego, czy podklasy znajdują się w tym samym, czy w różnych pakietach.

Dla użytkowników, którzy wcześniej korzystali z języka ActionScript 2.0, ta funkcja jest podobna do atrybutu `private` w języku ActionScript 2.0. Atrybut `protected` w języku ActionScript 3.0 jest również podobny do atrybutu `protected` w języku Java. Obydwa atrybuty różnią się jednak, ponieważ wersja Java zapewnia dostęp do modułów wywołujących w tym samym pakiecie. Atrybut `protected` jest użyteczny, jeśli mamy zmienną lub metodę potrzebną w podklasach, ale chcemy ukryć ją przed kodem zewnętrznym względem łańcucha dziedziczenia.

Atrybut `internal`, będący nowym elementem języka ActionScript 3.0, powoduje, że właściwość jest widoczna dla kodu wewnątrz jej własnego pakietu. Jest to domyślny atrybut dla kodu w pakiecie i obowiązuje dla każdej właściwości, której jawnie nie przypisano żadnego z poniższych atrybutów:

- `public`
- `private`
- `protected`
- przestrzeni nazw zdefiniowanej przez użytkownika

Atrybut `internal` działa podobnie jak domyślna kontrola dostępu w języku Java, z tym że w języku Java ten poziom dostępu nie ma jawnej nazwy i deklarowany jest po prostu przez pominięcie innych modyfikatorów dostępu. Atrybut `internal` w języku ActionScript 3.0 umożliwia jawne wyrażenie zamiaru uwidocznienia właściwości wyłącznie dla kodu w jej pakiecie.

Atrybut static

Atrybut `static`, którego można używać z właściwościami zadeklarowanymi za pomocą słów kluczowych `var`, `const` lub `function`, umożliwia powiązanie właściwości z klasą, a nie z instancjami tej klasy. Kod zewnętrzny względem klasy odwołuje się do właściwości statycznych za pośrednictwem nazwy klasy, a nie nazwy instancji.

Właściwości statyczne nie są dziedziczone przez podklasy, ale właściwości stanowią część łańcucha zasięgów podklas. Oznacza to, że w treści podklasy można używać zmiennej lub metody statycznej bez odwoływania się do klasy, w której została ona zdefiniowana.

Więcej tematów Pomocy

„[Właściwości statyczne — niedziedziczone](#)” na stronie 112

Atrybuty w postaci przestrzeni nazw zdefiniowanych przez użytkownika

Alternatywą dla wstępnie zdefiniowanych atrybutów kontroli dostępu jest utworzenie niestandardowych przestrzeni nazw i korzystanie z nich jak z atrybutów. W każdej definicji można użyć tylko jednego atrybutu w postaci przestrzeni nazw, a ponadto nie można używać przestrzeni nazw razem z którymkolwiek z atrybutów kontroli dostępu: (`public`, `private`, `protected`, `internal`).

Więcej tematów Pomocy

„[Przestrzeń nazw](#)” na stronie 40

Zmienne

Zmienne można deklarować za pomocą słów kluczowych `var` lub `const`. Wartości zmiennych zadeklarowanych za pomocą słowa kluczowego `var` można zmieniać wielokrotnie podczas wykonywania skryptu. Zmienne zadeklarowane za pomocą słowa kluczowego `const` są nazywane *stałymi* i można im przypisać wartość tylko raz. Próba przypisania nowej wartości do zainicjowanej stałej spowoduje błąd.

Zmienne statyczne

Zmienne statyczne deklaruje się za pomocą kombinacji słowa kluczowego `static` oraz instrukcji `var` albo `const`. Zmienne statyczne, powiązane z klasą, a nie z instancją klasy, są użyteczne do przechowywania i współużytkowania informacji należących do całej klasy obiektów. Zmienna statyczna będzie np. dobrym rozwiązaniem, jeśli chcemy rejestrować, ile instancji danej klasy utworzono, lub jeśli chcemy zapisać maksymalną dozwoloną liczbę instancji.

W poniższym przykładzie tworzona jest zmienna `totalCount`, która zlicza utworzone instancje klasy, oraz stała `MAX_NUM` zawierająca maksymalną liczbę instancji. Zmienne `totalCount` i `MAX_NUM` są statyczne, ponieważ zawierają wartości, które mają zastosowanie do klasy jako całości, a nie do konkretnej instancji.

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

Kod zewnętrzny względem klasy `StaticVars` i wszystkich jej podklas może odwoływać się do właściwości `totalCount` i `MAX_NUM` tylko za pośrednictwem samej klasy. Oto przykład poprawnego kodu:

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

Nie są dozwolone odwołania do zmiennych statycznych za pośrednictwem instancji klasy, dlatego następujący kod spowoduje zgłoszenie błędów:

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

Zmienne zadeklarowane za pomocą słów kluczowych `static` i `const` muszą być inicjowane w deklaracji stałej, co widzimy w deklaracji `MAX_NUM` w przykładowej klasie `StaticVars`. Nie można przypisać wartości do stałej `MAX_NUM` wewnątrz konstruktora lub metody instancji. Poniższy kod powoduje zgłoszenie błędu, ponieważ zastosowano w nim niepoprawny sposób inicjowania stałej statycznej:

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Zmienne instancji

Do zmiennych instancji zalicza się właściwości zadeklarowane za pomocą słów kluczowych `var` i `const`, ale bez słowa kluczowego `static`. Zmienne instancji, powiązane z instancjami klasy, a nie z całą klasą, są użyteczne do przechowywania wartości specyficznych dla instancji. Na przykład klasa `Array` ma właściwość instancji o nazwie `length`, która przechowuje liczbę elementów tablicy konkretnej instancji klasy `Array`.

Niedozwolone jest przesłanianie zmiennych instancji w podklasie, niezależnie od tego, czy zostały zadeklarowane jako `var`, czy jako `const`. Efekt zbliżony do przesłonięcia zmiennej można jednak uzyskać, przesłaniając metody pobierające i ustawiające.

Więcej tematów Pomocy

„[Stałe](#)” na stronie 66

„[Akcesory pobierające i ustawiające](#)” na stronie 98

Metody

Metody są funkcjami stanowiącymi część definicji klasy. Z chwilą utworzenia instancji klasy metoda zostaje powiązana z tą instancją. W przeciwieństwie do funkcji zadeklarowanych poza klasą, metody nie można używać w oderwaniu od instancji, z którą jest powiązana.

Do definiowania metod służy słowo kluczowe `function`. Podobnie, jak w przypadku każdej właściwości klasy, do metod można zastosować dowolne atrybuty właściwości klas, między innymi `private`, `protected`, `public`, `internal`, `static` lub `custom namespace`. Można również użyć instrukcji funkcji, takiej jak poniższa:

```
public function sampleFunction():String {}
```

Można też użyć zmiennej, której przypisane zostanie wyrażenie funkcyjne, w następujący sposób:

```
public var sampleFunction:Function = function () {}
```

W większości przypadków należy użyć instrukcji funkcyjnej, a nie wyrażenia, co wynika z następujących powodów:

- Instrukcje funkcyjne są bardziej zwarte i czytelne.
- Instrukcje funkcyjne umożliwiają stosowanie słów kluczowych `override` i `final`.

- Instrukcje funkcyjne tworzą silniejsze powiązanie między identyfikatorem — tj. nazwą funkcji — a kodem w treści metody. Ponieważ wartość zmiennej można zmienić za pomocą instrukcji przypisania, połączenie między zmienną a jej wyrażeniem funkcyjnym może zostać w dowolnej chwili zerwane. Mimo że istnieje obejście tego problemu, polegające na zadeklarowaniu zmiennej za pomocą słowa `const`, a nie `var`, takie postępowanie nie należy do dobrej praktyki programistycznych, ponieważ utrudnia zrozumienie kodu i uniemożliwia stosowanie słów kluczowych `override` i `final`.

Do sytuacji, w których konieczne jest użycie wyrażenia funkcyjnego, należy powiązanie funkcji z obiektem prototypowym.

Konstruktory

Konstruktory (nazywane też metodami-konstruktorami) to funkcje o tej samej nazwie, co klasa, w której są zdefiniowane. Kod zawarty w konstruktorze jest wykonywany za każdym razem, gdy za pomocą słowa kluczowego `new` tworzona jest nowa instancja klasy. Na przykład w poniższym kodzie zdefiniowano prostą klasę o nazwie `Example`, która zawiera jedną właściwość o nazwie `status`. Wartość początkowa zmiennej `status` jest przypisywana wewnątrz konstruktora.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Konstruktory mogą być tylko publiczne, ale użycie atrybutu `public` jest opcjonalne. W odniesieniu do konstruktora nie można używać żadnych innych specyfikatorów dostępu, w tym `private`, `protected` lub `internal`. Konstruktor nie można również zadeklarować w przestrzeni nazw zdefiniowanej przez użytkownika.

Konstruktor może jawnie wywołać konstruktor swojej bezpośredniej nadklasy. Służy do tego instrukcja `super()`. Jeśli konstruktor nadklasy nie zostanie wywołany jawnie, kompilator automatycznie wstawi wywołanie przed pierwszą instrukcją w treści konstruktora. Możliwa jest również wywoływanie metod nadklasy poprzez dodanie przedrostka `super` jako odwołania do nadklasy. Jeśli w tym samym konstruktorze ma być używana zarówno instrukcja `super()`, jak i przedrostek `super`, to instrukcja `super()` musi występować wcześniej. W przeciwnym razie odwołanie przez przedrostek `super` nie będzie działać zgodnie z oczekiwaniami. Konstruktor `super()` należy także wywołać przed ewentualną instrukcją `throw` lub `return`.

Poniższy przykład ilustruje skutki użycia odwołania `super` przed wywołaniem konstruktora `super()`. Nowa klasa, `ExampleEx`, rozszerza klasę `Example`. Konstruktor `ExampleEx` próbuje uzyskać dostęp do zmiennej `status` zdefiniowanej w jego nadklasie, ale próba ta ma miejsce przed wywołaniem `super()`. Instrukcja `trace()` w konstruktorze `ExampleEx` wyświetla wartość `null`, ponieważ zmienna `status` nie jest dostępna przed wykonaniem konstruktora `super()`.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Mimo że formalnie dozwolone jest użycie instrukcji `return` w konstruktorze, nie jest dozwolone zwrócenie wartości. Innymi słowy, instrukcja `return` nie może być skojarzona z wyrażeniem lub wartością. W związku z tym konstruktory nie mogą zwracać wartości i nie mogą mieć określonego typu zwracanej wartości.

Jeśli w klasie nie zostanie zdefiniowany konstruktor, kompilator automatycznie utworzy pustą metodę-konstruktor. Jeśli klasa rozszerza inną klasę, kompilator doda wywołanie `super()` do wygenerowanego konstruktora.

metody statyczne

Metody statyczne, nazywane także *metodami klas*, to metody zadeklarowane ze słowem kluczowym `static`. Metody statyczne, które są powiązane z klasą, a nie z instancją klasy, są przydatne do enkapsulacji funkcjonalności wpływającej na coś innego niż stan konkretnej instancji. Ponieważ metody statyczne są powiązane z klasą jako całością, dostęp do nich jest możliwy tylko za pośrednictwem klasy, a nie jej instancji.

Metody statyczne są przydatne do enkapsulacji funkcjonalności, która nie ogranicza się do zmiany stanu instancji klasy. Innymi słowy, metoda powinna być statyczna, jeśli jej działanie nie wpływa bezpośrednio na wartość instancji klasy. Na przykład klasa `Date` ma metodę statyczną o nazwie `parse()`, która konwertuje ciąg znaków na liczbę. Metoda jest statyczna, ponieważ nie wpływa na poszczególne instancje klasy. Metoda `parse()` analizuje ciąg znaków reprezentujący wartość daty i zwraca liczbę zgodną z wewnętrzną reprezentacją obiektu `Date`. Ta metoda nie jest metodą instancji, ponieważ nie ma sensu stosowanie jej do instancji klasy `Date`.

Porównajmy teraz metodę statyczną `parse()` z jedną z metod instancji klasy `Date`, np. z metodą `getMonth()`. `getMonth()` to metoda instancji, ponieważ operuje bezpośrednio na wartości instancji, pobierając konkretny składnik — miesiąc — instancji `Date`.

Ponieważ metody statyczne nie są powiązane z poszczególnymi instancjami, nie jest dozwolone użycie słów kluczowych `this` lub `super` w treści metody statycznej. Zarówno odwołanie `this`, jak i odwołanie `super`, mają znaczenie tylko w kontekście metody instancji.

Inaczej niż w niektórych innych językach programowania opartych na koncepcji klas, metody statyczne w języku ActionScript 3.0 nie są dziedziczone.

Metody instancji

Metody instancji są to metody zadeklarowane bez słowa kluczowego `static`. Metody instancji, które są powiązane z instancjami klasy, a nie z klasą jako całością, są użyteczne przy implementowaniu funkcjonalności wpływającej na poszczególne instancje klasy. Na przykład klasa `Array` zawiera metodę instancji o nazwie `sort()`, która operuje bezpośrednio na instancjach klasy `Array`.

Zarówno zmienne statyczne, jak i zmienne instancji znajdują się w zasięgu treści metody instancji, co oznacza, że do zmiennych zdefiniowanych w tej samej klasie można odwoływać się za pomocą prostych identyfikatorów. Na przykład poniższa klasa, `CustomArray`, rozszerza klasę `Array`. Klasa `CustomArray` definiuje zmienną statyczną o nazwie `arrayCountTotal`, która rejestruje łączną liczbę instancji klasy, zmienną instancji o nazwie `arrayNumber`, która rejestruje kolejność tworzenia instancji, oraz metodę instancji o nazwie `getPosition()`, która zwraca wartości tych zmiennych.

```
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);
    }
}
```

Mimo że kod zewnętrzny względem klasy musi mieć dostęp do zmiennej statycznej `arrayCountTotal` za pośrednictwem obiektu klasy, w formie `CustomArray.arrayCountTotal`, kod rezydujący w treści metody `getPosition()` może odwoływać się bezpośrednio do zmiennej statycznej `arrayCountTotal`. Zasada ta obowiązuje nawet dla zmiennych statycznych w nadklasach. Chociaż w języku ActionScript 3.0 właściwości statyczne nie są dziedziczone, właściwości statyczne z nadklas należą do aktualnego zasięgu. Na przykład klasa `Array` zawiera kilka zmiennych statycznych, z których jedna jest stałą o nazwie `DESCENDING`. Kod rezydujący w podklasie `Array` może mieć dostęp do statycznej stałej `DESCENDING` za pomocą prostego identyfikatora:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

Wartość odwołania `this` w treści metody instancji jest odwołaniem do instancji, z którą ta metoda jest powiązana. Poniższy kod demonstruje, że odwołanie `this` wskazuje na instancję zawierającą metodę:

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Dziedziczeniem metody instancji można sterować za pomocą słów kluczowych `override` i `final`. Atrybut `override` umożliwia przededefiniowanie metody odziedziczonej, a atrybut `final` zapobiega przesłanianiu metody przez podklasy.

Akcesory pobierające i ustawiające

Funkcje-akcesory pobierające i ustawiające, nazywane także *metodami pobierającymi* i *metodami ustawiającymi*, umożliwiają realizację koncepcji ukrywania i enkapsulacji informacji, a jednocześnie stworzenie wygodnego interfejsu programistycznego dla samodzielnie tworzonych klas. Funkcje pobierające i ustawiające pozwalają na zachowanie prywatności właściwości klasy, ale jednocześnie umożliwiają użytkownikom klasy dostęp do tych właściwości w taki sposób, jak do zmiennych, a nie w formie wywołania metody klasy.

Zaletą takiej strategii jest możliwość uniknięcia tradycyjnych akcesorów o „nieprzyjaznych” nazwach, takich jak `getProperty()` i `setProperty()`. Inną zaletą funkcje pobierających i ustawiających jest możliwość obycia się bez pary funkcji publicznych przypadającej na każdą właściwość, która ma być dostępna i do odczytu, i do zapisu.

Poniższa klasa przykładowa o nazwie `GetSet` zawiera funkcje pobierającą i ustawiającą o nazwie `publicAccess()`, która umożliwia dostęp do zmiennej prywatnej o nazwie `privateProperty`:

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Próba bezpośredniego dostępu do właściwości `privateProperty` powoduje zgłoszenie błędu:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Użytkownik klasy `GetSet` ma natomiast do dyspozycji mechanizm, który z pozoru przypomina właściwość o nazwie `publicAccess`, ale faktycznie jest parą akcesorów — złożoną z funkcji pobierającej i ustawiającej — operujących na właściwości prywatnej `privateProperty`. W poniższym przykładzie tworzona jest instancja klasy `GetSet`, a następnie wartość właściwości `privateProperty` ustawiana jest za pomocą publicznego akcesora o nazwie `publicAccess`:

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Funkcje pobierające i ustawiające umożliwiają także przesłanianie właściwości dziedziczonych z nadklasy, co nie jest możliwe w przypadku zwykłych zmiennych należących do klasy. Zmienne należące do klasy zadeklarowane za pomocą słowa kluczowego `var` nie mogą być przesłaniane w podklasach. Jednak ograniczenie to nie obowiązuje w przypadku właściwości tworzonych za pomocą funkcji pobierających i ustawiających. W odniesieniu do funkcji pobierających i ustawiających odziedziczonych z nadklasy można używać atrybutu `override`.

Metody powiązane

Metoda powiązana, nazywana czasem *zamknięciem metody*, to po prostu metoda wyodrębniona z jej instancji. Przykładami metod powiązanych są metody przekazywane jako argumenty do funkcji lub zwracane jako wartości z funkcji. Metody powiązane, będące nowym elementem języka ActionScript 3.0, są podobne do zamknięć funkcji w tym, że zachowują swoje środowisko leksykalne nawet po wyodrębnieniu z instancji. Zasadnicza różnica między metodą powiązaną a zamknięciem funkcji polega na tym, że odwołanie `this` w metodzie powiązanej pozostaje powiązane z instancją implementującą tę metodę. Innymi słowy, odwołanie `this` w metodzie powiązanej zawsze wskazuje na oryginalny obiekt, który zaimplementował metodę. W przypadku zamknięć funkcji odwołanie `this` ma charakter ogólny, czyli wskazuje na obiekt lub funkcję, z którym jest skojarzone w momencie wywołania.

Zrozumienie koncepcji metod powiązanych jest ważne w przypadku posługiwania się słowem kluczowym `this`. Jak pamiętamy, słowo kluczowe `this` jest odwołaniem do obiektu nadrzędnego metody. Większość programistów używających języka ActionScript oczekuje, że słowo kluczowe `this` przedstawia obiekt lub klasę zawierającą definicję metody. Jednak bez mechanizm wiązania metod warunek ten nie zawsze byłby spełniony. Na przykład we wcześniejszych wersjach języka ActionScript odwołanie `this` nie zawsze wskazywało na instancję, która implementowała metodę. W języku ActionScript 2.0 w metodach wyodrębnionych z instancji nie tylko słowo kluczowe `this` nie było powiązane z oryginalną instancją, lecz również nie były dostępne zmienne i metody należące do klasy instancji. Nie stanowi to problemu w języku ActionScript 3.0, w którym metody powiązane są tworzone automatycznie podczas przekazywania metod jako parametrów. Metody powiązane gwarantują, że słowo kluczowe `this` zawsze odwołuje się do obiektu lub klasy, w którym metoda jest zdefiniowana.

Poniższy kod definiuje klasę o nazwie `ThisTest`, która zawiera metodę o nazwie `foo()` definiującą metodę powiązaną, a także metodę o nazwie `bar()`, która zwraca metodę powiązaną. Kod zewnętrzny względem klasy tworzy instancję klasy `ThisTest`, wywołuje metodę `bar()` i zapisuje zwróconą wartość w zmiennej o nazwie `myFunc`.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

Ostatnie dwa wiersze kodu demonstrują fakt, że odwołanie `this` w metodzie powiązanej `foo()` nadal wskazuje na klasę `ThisTest`, mimo że odwołanie `this` w wierszy bezpośrednio poprzedzającym wskazuje na obiekt globalny. Co więcej, metoda powiązana przechowywana w zmiennej `myFunc` wciąż ma dostęp do zmiennych klasy `ThisTest`. Gdyby ten sam kod został uruchomiony w języku ActionScript 2.0, odwołania `this` byłyby identyczne, a zmienna `num` miałaby wartość `undefined`.

Jednym z obszarów, w których nowa koncepcja metod powiązanych jest najbardziej zauważalna, są podprogramy obsługi zdarzeń, ponieważ metoda `addEventListener()` wymaga, aby jako argument przekazać do niej funkcję lub metodę.

Wyliczenia w oparciu o klasy

Wyliczenia to niestandardowe typy danych, które definiuje się w celu enkapsulacji małego zbioru wartości. W języku ActionScript 3.0 nie istnieje specjalny mechanizm wyliczeń, porównywalny ze słowem kluczowym `enum` w języku C++ lub interfejsem `Enumeration` w języku Java. Możliwe jest jednak tworzenie wyliczeń przy użyciu klas i stałych statycznych. Na przykład: klasa `PrintJob` w języku ActionScript 3.0 korzysta z wyliczenia o nazwie `PrintJobOrientation` zawierającego wartości: `"landscape"` i `"portrait"`, co ilustruje poniższy kod:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Przyjęto konwencję, że klasa wyliczeniowa powinna być zadeklarowana z atrybutem `final`, ponieważ nie ma potrzeby rozszerzania klasy. Klasa zawiera tylko elementy statyczne, co oznacza, że użytkownik nie tworzy instancji klasy. Dostęp do wartości wyliczenia odbywa się wprost przez obiekt klasy, co ilustruje poniższy fragment kodu:

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Wszystkie klasy wyliczeniowe języka ActionScript 3.0 zawierają tylko zmienne typu `String`, `int` lub `uint`. Korzyścią ze stosowania wyliczeń w miejsce literalnych ciągów lub wartości liczbowych jest możliwość łatwiejszego znajdowania błędów typograficznych. W wypadku pomyłki w zapisie wartości wyliczeniowej kompilator języka ActionScript zgłosi błąd. Gdy natomiast używane są literały, kompilator nie sygnalizuje obecności nieprawidłowo wpisanego słowa lub błędnej liczby. W poprzednim przykładzie kompilator generuje błąd, jeśli nazwa stałej wyliczeniowej jest nieprawidłowa, co ilustruje następujący fragment:

```
if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Jednak kompilator nie zgłosi błędu w razie pomyłki w zapisie literalnego ciągu znaków:

```
if (pj.orientation == "portrai") // no compiler error
```

Druga technika tworzenia wyliczeń również wymaga utworzenia osobnej klasy ze statycznymi właściwościami dla wartości wyliczeniowych. Jednak różni się od pierwszej tym, że każda z właściwości statycznych zawiera instancję klasy, a nie ciąg znaków lub wartość całkowitą. Poniższy przykładowy kod tworzy klasę wyliczeniową zawierającą dni tygodnia:

```
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();
}
```

Ta technika nie jest stosowana w języku ActionScript 3.0, ale wielu programistów korzysta z niej z uwagi na ściślejszą kontrolę typów. Na przykład metoda, która zwraca wartość wyliczeniową, może ograniczyć zbiór wartości zwracanych do konkretnego typu wyliczeniowego. Poniższy kod ilustruje nie tylko funkcję zwracającą dzień tygodnia, lecz również wywołanie funkcji, w której typ wyliczeniowy został użyty jako wskazany typ zadeklarowany:

```
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();
```

Możemy udoskonalić klasę Day w taki sposób, aby kojarzyła liczbę całkowitą z każdym dniem tygodnia i udostępniała metodę `toString()` zwracającą ciąg znaków z nazwą dnia.

Klasy zasobów osadzonych

W języku ActionScript 3.0 istnieją specjalne klasy, nazywane *klasami zasobów osadzonych*, służące do reprezentacji zasobów osadzonych. *Zasób osadzony* to zasób, taki jak dźwięk, obraz lub czcionka, włączony do pliku SWF w czasie kompilacji. Osadzenie zasobu, jako alternatywa dla dynamicznego ładowania go, gwarantuje dostępność zasobu w czasie wykonywania, jednak kosztem większej objętości pliku SWF.

Korzystanie z klas zasobów osadzonych w środowisku Flash Professional

Aby osadzić zasób, należy najpierw umieścić go w bibliotece pliku FLA. Następnie, korzystając z właściwości połączenia zasobu, należy podać nazwę klasy zasobu osadzonego. Jeśli klasy o tej nazwie nie ma w ścieżce klas, to zostanie automatycznie wygenerowana. Następnie tworzymy instancję klasy zasobu osadzonego i korzystamy z właściwości i metod zdefiniowanych lub odziedziczonych przez tę klasę. Poniższy przykładowy kod umożliwia odtworzenie osadzonego dźwięku połączonego z klasą zasobu osadzonego o nazwie PianoMusic:

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Można również wykorzystać znacznik metadanych `[Embed]` w celu osadzenia zasobów w projekcie Flash Professional, co zostało opisane w dalszej części niniejszego podręcznika. Jeśli w kodzie używany jest znacznik `[Embed]`, program Flash Professional wykorzystuje kompilator Flex w celu skompilowania projektu, nie korzysta natomiast z kompilatora Flash Professional.

Korzystanie z osadzonych klas zasobów, które wykorzystują kompilator Flex

W przypadku kompilowania kodu za pomocą kompilatora Flex w celu osadzenia zasobu w kodzie ActionScript należy użyć znacznika metadanych `[Embed]`. Zasób należy umieścić w głównym folderze źródłowym lub innym folderze w ścieżce budowania projektu. Gdy tylko kompilator Flex napotyka znacznik `Embed`, tworzy klasę zasobu osadzonego. Dostęp do tej klasy jest możliwy za pośrednictwem zmiennej o typie danych `Class`, którą deklaruje się bezpośrednio po znaczniku metadanych `[Embed]`. W poniższym przykładowym kodzie osadzony jest dźwięk o nazwie `sound1.mp3`, a odwołanie do klasy zasobu osadzonego powiązane z tym dźwiękiem jest przechowywane w zmiennej o nazwie `soundCls`. Następnie przykładowy kod tworzy instancję klasy obiektu osadzonego i wywołuje metodę `play()` tej instancji:

```
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

Aby użyć znacznika metadanych `[Embed]` w projekcie ActionScript w środowisku Flash Builder, należy zaimportować wszelkie potrzebne klasy ze środowiska Flex. Na przykład, aby osadzać dźwięki, należy zaimportować klasę `mx.core.SoundAsset`. Aby użyć środowiska Flex, należy umieścić plik `framework.swc` w ścieżce budowania projektu ActionScript. Spowoduje to zwiększenie wielkości pliku SWF.

Adobe Flex

We Flex alternatywny sposób osadzania zasobów polega na użyciu dyrektywy `@Embed()` w definicji znacznika MXML.

Interfejsy

Interfejs jest to kolekcja deklaracji metod umożliwiająca wzajemną komunikację między niepowiązanymi ze sobą obiektami. Na przykład w języku ActionScript 3.0 zdefiniowany jest interfejs `IEventDispatcher` zawierający deklaracje metod, których klasy mogą używać do obsługi obiektów zdarzeń. Interfejs `IEventDispatcher` udostępnia jednolity sposób przekazywania obiektów zdarzeń między obiektami. Poniższej przedstawiono definicję interfejsu `IEventDispatcher`:

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
        useCapture:Boolean=false, priority:int=0,
        useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
        useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Interfejsy bazują na koncepcji rozdzielenia interfejsu metody od jej implementacji. Interfejs metody zawiera wszystkie informacje potrzebne do wywołania metody, w tym jej nazwę, wszystkie parametry oraz typ zwracanej wartości. Implementacje metody zawiera nie tylko informacje podane w interfejsie, lecz również instrukcje wykonywalne, które realizują zachowanie metody. Definicja interfejsu zawiera tylko interfejsy metod, a każda klasa implementująca interfejs musi definiować implementacje metod.

W języku ActionScript 3.0 klasa `EventDispatcher` implementuje interfejs `IEventDispatcher`, definiując wszystkie metody interfejsu `IEventDispatcher` oraz dodając treści tych metod. Poniższy kod stanowi wycinek z definicji klasy `EventDispatcher`:

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

Interfejs `IEventDispatcher` pełni rolę protokołu, zgodnie z którym instancje klasy `EventDispatcher` przetwarzają obiekty zdarzeń i przekazują je do innych obiektów, które także implementują interfejs `IEventDispatcher`.

O interfejsie możemy też powiedzieć, że definiuje typ danych, podobnie jak klasa. W związku z tym nazwy interfejsu można użyć w charakterze wskazania typu danych, tak samo jak klasy. Jako typ danych interfejs może być również używany z operatorami, takimi jak `is` oraz `as`, które jako operandu wymagają typu danych. Jednak w odróżnieniu od klasy interfejs nie może posłużyć do utworzenia instancji. To rozróżnienie sprawia, że wielu programistów postrzega interfejsy jako abstrakcyjne typy danych, zaś klasy jako konkretne typy danych.

Definiowanie interfejsu

Struktura definicji interfejsu przypomina strukturę definicji klasy, z tym że interfejs może zawierać tylko deklaracje metod, a nie ich treści. Interfejsy nie mogą zawierać zmiennych ani stałych, ale mogą zawierać metody ustawiające i pobierające. Do definiowania interfejsów służy słowo kluczowe `interface`. Na przykład: interfejs `IExternalizable` jest częścią pakietu `flash.utils` w języku ActionScript 3.0. Interfejs `IExternalizable` definiuje protokół dla szeregowania obiektu, co oznacza konwertowanie obiektu na format odpowiedni od zapisu na urządzeniu lub do przesyłania w sieci.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

Interfejs `IExternalizable` został zadeklarowany ze specyfikatorem dostępu `public`. Definicje interfejsu można modyfikować wyłącznie poprzez dodanie specyfikatorów dostępu `public` albo `internal`. Deklaracje metod wewnątrz definicji interfejsu nie mogą mieć żadnych specyfikatorów dostępu.

W języku ActionScript 3.0 stosowana jest konwencja, zgodnie z którą nazwy interfejsu rozpoczynają się od wielkiej litery `I`, jednak jako nazwy interfejsu można użyć dowolnego poprawnego identyfikatora. Definicje interfejsów często umieszcza się na najwyższym poziomie pakietu. Definicji interfejsu nie można umieszczać wewnątrz definicji klas ani wewnątrz definicji innych interfejsów.

Interfejs może rozszerzać jeden lub większą liczbę innych interfejsów. Na przykład poniższy interfejs, `IExample`, rozszerza interfejs `IExternalizable`:

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Każda klasa, która implementuje interfejs `IExample`, musi zawierać implementacje nie tylko metod `extra()`, lecz także metod `writeExternal()` i `readExternal()` odziedziczonych z interfejsu `IExternalizable`.

Implementowanie interfejsu w klasie

Klasa jest jedynym elementem języka ActionScript 3.0, który może implementować interfejs. Aby zaimplementować jeden lub większą liczbę interfejsów, należy umieścić w deklaracji klasy słowo kluczowe `implements`. W poniższym przykładzie zdefiniowano dwa interfejsy, `IAlpha` oraz `IBeta`, a także klasę `Alpha`, która implementuje oba te interfejsy:

```
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 {}
}
```

W klasie, która implementuje interfejs, zaimplementowane metody muszą charakteryzować się następującymi cechami:

- Muszą być zadeklarowane ze specyfikatorem dostępu `public`.
- Muszą mieć te same nazwy, co odpowiednie metody w interfejsie.
- Muszą mieć tę samą liczbę parametrów o tych samych typach, co odpowiednie metody w interfejsie.
- Muszą zwracać wartości tego samego typu, co odpowiednie metody w interfejsie.

```
public function foo(param:String):String {}
```

Programista ma jednak pewną swobodę w nadawaniu nazw parametrom implementowanych metod. Mimo że liczba parametrów i ich typy danych w zaimplementowanej metodzie muszą być identyczne, jak w metodzie zadeklarowanej w interfejsie, nazwy parametrów nie muszą być identyczne. W poprzednim przykładzie parametr metody `Alpha.foo()` ma nazwę `param`:

Jednak w metodzie `IAlpha.foo()` zadeklarowanej w interfejsie ten sam parametr ma nazwę `str`:

```
function foo(str:String):String;
```

Możliwe jest także swobodne określanie wartości domyślnych parametrów. Definicja interfejsu może zawierać deklaracje funkcji z domyślnymi wartościami parametrów. Metoda implementująca taką deklarację funkcji musi mieć wartość domyślną parametru należącą do tego samego typu danych, co wartość określona w definicji interfejsu, ale wartości te nie muszą być równe. Na przykład w poniższym kodzie zdefiniowano interfejs zawierający metodę z parametrem o wartości domyślnej 3:

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

Poniższa definicja klasy implementuje interfejs `IGamma`, ale z inną domyślną wartością parametru:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Swoboda ta wynika z faktu, że reguły implementacji interfejsów opracowano z myślą o zgodności typów danych, ale wymów identycznych nazw parametrów i domyślnych ich wartości nie jest niezbędny do zapewnienia takiej zgodności.

Dziedziczenie

Dziedziczenie to technika ponownego wykorzystania kodu, która umożliwia programistom opracowywanie nowych klas w oparciu o istniejące klasy. Te istniejące klasy są często nazywane *klasami bazowymi* lub *nadklasami*, natomiast nowe klasy nazywa się *podklasami*. Kluczową zaletą dziedziczenia jest możliwość ponownego użycia kodu z klasy bazowej, bez jego przerabiania. Co więcej, dziedziczenie nie wymaga zmian w sposobie interakcji innych klas z klasą bazową. Zamiast modyfikować istniejące klasy, które być może zostały już gruntownie przetestowane i są używane, można wykorzystać mechanizm dziedziczenia i potraktować istniejącą klasę jako zintegrowany moduł, który w razie potrzeby rozszerza się o dodatkowe właściwości i metody. Słowo kluczowe `extends` umożliwia zadeklarowanie, że dana klasa dziedziczy z innej klasy.

Dziedziczenie umożliwia także wykorzystanie *polimorfizmu* w programach. Polimorfizm to możliwość użycia jednej nazwy dla metody, która zachowuje się różnie w zależności od tego, do jakiego typu danych zostanie zastosowana. Za prosty przykład może posłużyć klasa o nazwie `Shape` z dwiema podklasami o nazwach `Circle` i `Square`. Klasa `Shape` definiuje metodę o nazwie `area()`, która zwraca pole powierzchni kształtu. W wypadku zaimplementowania polimorfizmu możliwe będzie wywoływanie metody `area()` w obiektach typu `Circle` i `Square`, a wynik obliczeń będzie za każdym razem poprawny dla danej figury. Dziedziczenie otwiera drogę do stosowania polimorfizmu, pozwalając na przedefiniowanie (*przesłanianie*) metod odziedziczonych z klasy bazowej w podklasach. W poniższym przykładzie metoda `area()` została przedefiniowana w klasach `Circle` i `Square`:

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Ponieważ każda klasa jest definicją typu danych, dziedziczenie tworzy szczególną relację między klasą bazową, a klasą, która ją rozszerza. Podklasa z definicji zawiera wszystkie właściwości swojej klasy bazowej, co oznacza, że w miejsce instancji klasy bazowej zawsze można użyć instancji podklasy. Na przykład, jeśli metoda definiuje parametr typu `Shape`, to dozwolone jest przekazanie do niej parametru typu `Circle`, ponieważ klasa `Circle` rozszerza klasę `Shape`, co ilustruje następujący kod:

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Właściwości instancji i dziedziczenie

Właściwość instancji, niezależnie od tego, czy została zdefiniowana za pomocą słowa kluczowego `function`, `var`, czy `const`, jest dziedziczona przez wszystkie podklasy, chyba że została w klasie bazowej zadeklarowana z atrybutem `private`. Na przykład klasa `Event` w języku ActionScript 3.0 ma szereg podklas dziedziczących właściwości wspólne dla wszystkich obiektów zdarzeń.

W przypadku niektórych typów obiektów klasa `Event` zawiera wszystkie właściwości potrzebne do zdefiniowania zdarzenia. Zdarzenia tego typu nie wymagają użycia właściwości instancji innych niż zdefiniowane w klasie bazowej `Event`. Do takich zdarzeń należą: zdarzenie `complete` wywoływane po pomyślnym załadowaniu danych i zdarzenie `connect` wywoływane po nawiązaniu połączenia sieciowego.

Poniższy przykład stanowi fragment klasy `Event`, który zawiera niektóre właściwości i metody dziedziczone przez podklasy. Ponieważ te właściwości są dziedziczone, instancja każdej podklasy ma do nich dostęp.

```
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 {}
    ...
}
```

Inne typy zdarzeń wymagają szczególnych właściwości, niedostępnych w klasie `Event`. Te zdarzenia zdefiniowane są przy użyciu podklas klasy `Event`, co umożliwia dodawanie nowych właściwości do zestawu zdefiniowanego w klasie `Event`. Przykładem takiej podklasy jest klasa `MouseEvent`, która zawiera dodatkowe właściwości unikatowe dla zdarzeń związanych z ruchem myszy lub kliknięciami myszą, takich jak zdarzenia `mouseMove` i `click`. Poniższy przykład przedstawia fragment klasy `MouseEvent` zawierający definicję właściwości istniejących w podklasie, ale nie w klasie bazowej:

```
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 {}
    ...
}
```

Specyfikatory dostępu a dziedziczenie

Właściwość zadeklarowana ze słowem kluczowym `public` jest widoczna w każdym miejscu kodu. Oznacza to, że słowo kluczowe `public`, w przeciwieństwie do słów `private`, `protected` i `internal`, nie nakłada żadnych ograniczeń na dziedziczenie właściwości.

Właściwość zadeklarowana ze słowem kluczowym `private` jest widoczna tylko w klasie, w której została zdefiniowana, a zatem nie jest dziedziczona przez podklasy. Ta reguła różni się od obowiązującej w poprzednich wersjach języka ActionScript, w których słowo kluczowe `private` miało znaczenie bliższe temu, które w języku ActionScript 3.0 ma słowo `protected`.

Słowo kluczowe `protected` oznacza, że właściwość jest widoczna nie tylko w klasie, w której została zdefiniowana, lecz także w jej podklasach. W przeciwieństwie do słowa kluczowego `protected` w języku programowania Java, słowo kluczowe `protected` w języku ActionScript 3.0 nie powoduje widoczności właściwości dla wszystkich innych klas w tym samym pakiecie. W języku ActionScript 3.0 tylko podklasy mogą mieć dostęp do właściwości zadeklarowanej ze słowem kluczowym `protected`. Ponadto właściwość zadeklarowana jako `protected` jest widoczna dla podklasy niezależnie od tego, czy ta podklasa znajduje się w tym samym pakiecie, co klasa bazowa, czy też w innym pakiecie.

Aby ograniczyć widoczność właściwości do pakietu, w którym jest zdefiniowana, należy użyć słowa kluczowego `internal` lub w ogóle pominąć specyfikator dostępu. Specyfikator dostępu `internal` jest specyfikatorem domyślnym, który obowiązuje także w wypadku pominięcia specyfikatora. Właściwość oznaczona słowem `internal` będzie dziedziczona tylko przez podklasy rezydujące w tym samym pakiecie.

Poniższy przykład pozwala przekonać się, w jaki sposób poszczególne specyfikatory dostępu wpływają na dziedziczenie między pakietami. W przedstawionym kodzie zdefiniowano główną klasę aplikacji o nazwie `AccessControl` oraz dwie pozostałe klasy o nazwach `Base` i `Extender`. Klasa `Base` znajduje się w pakiecie o nazwie `foo`, a klasa `Extender`, będąca podklasą klasy `Base`, znajduje się w pakiecie o nazwie `bar`. Klasa `AccessControl` importuje tylko klasę `Extender` i tworzy instancję klasy `Extender`, która próbuje uzyskać dostęp do zmiennej o nazwie `str` zdefiniowanej w klasie `Base`. Zmienna `str` jest zadeklarowana jako `public`, a zatem kod zostanie prawidłowo skompilowany i wykonany w postaci przedstawionej w poniższym wycinku:

```
// 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
        }
    }
}
```

Aby sprawdzić, jak inne specyfikatory dostępu wpływają na kompilację i wykonanie powyższego przykładu, zmień specyfikator dostępu zmiennej `str` na `private`, `protected` lub `internal`, a wcześniej usuń lub ujmij w komentarz następujący wiersz klasy `AccessControl`:

```
trace(myExt.str); // error if str is not public
```

Niedozwolone przesłanianie zmiennych

Właściwości zadeklarowane za pomocą słów kluczowych `var` lub `const` są dziedziczone, ale nie mogą być przesłaniane. Przesłonięcie właściwości oznacza przeddefiniowanie jej w podklasie. Jedynym rodzajem właściwości, jaki można przesłaniać, są metody pobierające i ustawiające (właściwości zadeklarowane za pomocą słowa kluczowego `function`). Mimo że niedozwolone jest przesłonięcie zmiennej instancji, można uzyskać podobną funkcjonalność, tworząc metodę pobierającą i ustawiającą dla zmiennej instancji i przesłaniając te metody.

Przesłanianie metod

Przesłonięcie metody oznacza przeddefiniowanie zachowania metody odziedziczonej. Metody statyczne nie są dziedziczone i nie mogą być przesłaniane. Jednak metody instancji są dziedziczone przez podklasy i mogą być przesłaniane, o ile spełnione są następujące dwa kryteria:

- Metoda instancji nie jest zadeklarowana w klasie bazowej ze słowem kluczowym `final`. Słowo kluczowe `final` w metodzie instancji oznacza, że programista chce zapobiec przesłanianiu tej metody w podklasach.
- Metoda instancji nie jest zadeklarowana w klasie bazowej ze specyfikatorem dostępu `private`. Jeśli metoda jest oznaczona w klasie bazowej jako `private`, można ją w podklasie zadeklarować pod tą samą nazwą nawet bez słowa kluczowego `override`, ponieważ metoda klasy bazowej nie jest widoczna w podklasie.

Aby przesłonić metodę instancji spełniającą te kryteria, należy w definicji metody w podklasie użyć słowa kluczowego `override`, a wersja metody w podklasie musi być zgodna z metodą w nadklasie pod następującymi względami:

- Metoda przesłaniająca musi mieć ten sam specyfikator dostępu, co metoda klasy bazowej. Metody oznaczone jako `internal` są uznawane za posiadające ten sam specyfikator dostępu, co klasy zadeklarowane bez specyfikatora.
- Metoda przesłaniająca musi mieć tę samą liczbę parametrów, co metoda klasy bazowej.
- Parametry metody przesłaniającej muszą mieć te same zadeklarowane typy danych, co parametry metody klasy bazowej.
- Metoda przesłaniająca musi mieć zadeklarowany ten sam typ wartości zwracanej, co metoda klasy bazowej.

Jednak nazwy parametrów w metodzie przesłaniającej nie muszą być identyczne z nazwami parametrów w klasie bazowej, jeśli tylko ich liczba i typy danych odpowiednich parametrów są zgodne z klasą bazową.

Instrukcja `super`

Przesłaniając metodę, programista często chce uzupełnić zachowanie przesłanianej metody nadklasy, a nie całkowicie je zmienić. Dlatego potrzebny jest mechanizm umożliwiający metodzie podklasy wywoływanie wersji samej siebie zdefiniowanej w nadklasie. Tym mechanizmem jest instrukcja `super`, która udostępnia odwołanie do bezpośredniej nadklasy. W poniższym przykładzie zdefiniowano klasę o nazwie `Base` zawierającą metodę o nazwie `thanks()` oraz podklasę klasy `Base` o nazwie `Extender`, która przesłania metodę `thanks()`. W metodzie `Extender.thanks()` używana jest instrukcja `super` w celu wywołania metody `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Przesłanianie metod pobierających i ustawiających

Wprawdzie nie jest możliwe przesłanianie zmiennych zdefiniowanych w nadklasie, można przesłaniać metody pobierające i ustawiające. Poniższy przykładowy kod przesłania metodę pobierającą o nazwie `currentLabel` zdefiniowaną w klasie `MovieClip` w języku ActionScript 3.0:

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

Instrukcja `trace()` w konstruktorze klasy `OverrideExample` wyświetla wynik `Override: null`, co oznacza, że program pomyślnie przesłonił odziedziczoną właściwość `currentLabel`.

Właściwości statyczne — niedziedziczone

Właściwości statyczne nie są dziedziczone przez podklasy. Oznacza to, że właściwości statyczne nie są dostępne dla instancji podklas. Właściwość statyczna jest dostępna tylko za pośrednictwem obiektu klasy, w którym jest zdefiniowana. Na przykład w poniższym kodzie zdefiniowano klasę bazową o nazwie `Base` oraz podklasę o nazwie `Extender` rozszerzającą klasę `Base`. W klasie `Base` zdefiniowana jest zmienna statyczna o nazwie `test`. Fragment kodu przedstawiony poniżej nie zostanie bezbłędnie skompilowany w trybie ścisłym i spowoduje błąd w czasie wykonywania w trybie standardowym.

```
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 { }
```

Dostęp do zmiennej statycznej `test` można uzyskać wyłącznie za pośrednictwem obiektu klasy, co ilustruje poniższy kod:

```
Base.test;
```

Dozwolone jest natomiast zdefiniowanie właściwości instancji o tej samej nazwie, co nazwa właściwości statycznej. Taka właściwość instancji może być zdefiniowana w tej samej klasie, co właściwość statyczna, lub w podklasie. Na przykład klasa `Base` z poprzedniego przykładu mogłaby zawierać właściwość instancji o nazwie `test`. Poniższy kod zostanie skompilowany i wykonany bez błędów, ponieważ właściwość instancji jest dziedziczona przez klasę `Extender`. Kod dałoby się również skompilować i wykonać, gdyby definicja zmiennej instancji `test` została przeniesiona, ale nie skopiowana, do klasy `Extender`.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test); // output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Właściwości statyczne i łańcuch zasięgów

Mimo że właściwości statyczne nie są dziedziczone, należą do łańcuchu zasięgów klasy, która je definiuje, oraz wszystkich podklas tej klasy. Dlatego mówimy, że właściwości statyczne są *w zasięgu* zarówno klasy, w której są zdefiniowane, jak i wszystkich jej podklas. Oznacza to, że właściwość statyczna jest bezpośrednio dostępna w treści klasy, w której została zdefiniowana oraz we wszystkich podklasach tej klasy.

Poniższy przykład modyfikuje klasy zdefiniowane w poprzednim przykładzie, aby zademonstrować, że statyczna zmienna `test` zdefiniowana w klasie `Base` jest w zasięgu klasy `Extender`. Innymi słowy, klasa `Extender` może odwoływać się do zmiennej statycznej `test` bez konieczności poprzedzania nazwy tej zmiennej nazwą klasy, w której zmienna `test` została zdefiniowana.

```
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
    }
}
```

W wypadku zadeklarowania właściwości instancji o tej samej nazwie, którą ma już właściwość statyczna w tej samej klasie lub nadklasie, właściwość instancji będzie miała wyższy priorytet w łańcuchu zasięgów. Mówimy, że właściwość instancji jest *cieniem* właściwości statycznej, co oznacza, że zamiast właściwości statycznej będzie używana wartość właściwości instancji. Poniższy przykładowy kod dowodzi, że w razie zdefiniowania w klasie Extender zmiennej instancji o nazwie test instrukcja `trace()` wyświetli wartość zmiennej instancji, a nie wartość zmiennej statycznej:

```
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
    }
}
```

Tematy zaawansowane

Historia elementów programowania zorientowanego obiektowo w języku ActionScript

Ponieważ język ActionScript 3.0 jest oparty na poprzednich wersjach języka ActionScript, warto przyrzeć się ewolucji modelu obiektowego w kolejnych wersjach języka. ActionScript był początkowo prostym mechanizmem skryptowym współpracującym z wczesnymi wersjami programu Flash Professional. Z czasem programiści zaczęli budować z jego pomocą coraz bardziej złożone aplikacje. W odpowiedzi na potrzeby takich programistów w kolejnych wersjach wprowadzano elementy języka usprawniające tworzenie skomplikowanych aplikacji.

ActionScript 1.0

ActionScript 1.0 to wersja języka używana w programie Flash Player 6 i wcześniejszych wersjach. Nawet na tym wczesnym etapie rozwoju model obiektowy języka ActionScript był oparty na koncepcji obiektu jako fundamentalnego typu danych. Obiekt w języku ActionScript jest złożonym typem danym składającym się z grupy *właściwości*. Przy omawianiu modelu obiektowego termin *właściwość* oznacza wszystkie elementy skojarzone z obiektem, takie jak zmienne, funkcje i metody.

Mimo że w pierwszej generacji języka ActionScript nie było możliwe definiowanie klas za pomocą słowa kluczowego `class`, można było definiować klasy przy użyciu specjalnego rodzaju obiektu, tzw. obiektu prototypowego. W językach opartych na klasach, takich jak Java i C++, programista za pomocą słowa kluczowego `class` tworzy abstrakcyjną definicję klasy, a następnie tworzy konkretne obiekty będące instancjami tej klasy. Natomiast w językach opartych na prototypach, takich jak ActionScript 1.0, używa się istniejącego obiektu jako modelu (prototypu) dla innych obiektów. Podczas gdy obiekty w języku opartym na klasach mogą wskazywać klasę będącą ich szablonem, obiekty w języku opartym na prototypach wskazują na inny obiekt, prototyp, pełniący rolę takiego szablonu.

Aby utworzyć klasę w języku ActionScript 1.0 należy zdefiniować funkcję będącą konstruktorem tej klasy. W języku ActionScript funkcje są faktycznie obiektami, a nie tylko abstrakcyjnymi definicjami. Utworzony konstruktor pełni rolę obiektu prototypowego dla instancji tej klasy. W poniższym kodzie utworzono klasę o nazwie `Shape` i zdefiniowano jedną właściwość o nazwie `visible`, która domyślnie przyjmuje wartość `true`:

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Ten konstruktor definiuje klasę `Shape`, której instancje można tworzyć za pomocą operatora `new`, w następujący sposób:

```
myShape = new Shape();
```

Obiekt funkcji-konstruktor `Shape()` jest prototypem instancji klasy `Shape`, ale może także pełnić rolę prototypu podklas klasy `Shape` — tj. innych klas rozszerzających klasę `Shape`.

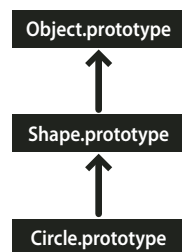
Tworzenie podklasy klasy `Shape` przebiega dwuetapowo. Najpierw należy utworzyć klasę, definiując jej konstruktor, w następujący sposób:

```
// child class
function Circle(id, radius)
{
    this.id = id;
    this.radius = radius;
}
```

Następnie za pomocą operatora `new` należy zadeklarować, że klasa `Shape` jest prototypem klasy `Circle`. Domyślnie prototypem każdej tworzonej kasy jest klasa `Object`, co oznacza, że właściwość `Circle.prototype` zawiera obecnie obiekt ogólny (instancję klasy `Object`). Aby wskazać, że prototypem klasy `Circle` jest klasa `Shape`, a nie `Object`, należy za pomocą poniższego kodu zmienić wartość właściwości `Circle.prototype`, tak aby zawierała obiekt `Shape`, a nie obiekt ogólny:

```
// Make Circle a subclass of Shape.  
Circle.prototype = new Shape();
```

Klasa `Shape` jest teraz powiązana z klasą `Circle` relacją dziedziczenia nazywaną *łańcuchem prototypów*. Poniższy diagram ilustruje relacje w łańcuchu prototypów:



Klasą bazową na końcu każdego łańcucha prototypów jest zawsze klasa `Object`. Klasa `Object` zawiera właściwość statyczną o nazwie `Object.prototype`, która wskazuje na bazowy obiekt prototypowy wszystkich obiektów tworzonych w języku ActionScript 1.0. Następnym obiektem w przykładowym łańcuchu prototypów jest obiekt `Shape`. Wynika to z faktu, że właściwości `Shape.prototype` nigdy w sposób jawny nie przypisano wartości, a więc nadal zawiera ona obiekt ogólny (instancję klasy `Object`). Ostatnim ogniwem tego łańcucha jest klasa `Circle` powiązana ze swoim prototypem, klasą `Shape` (właściwość `Circle.prototype` zawiera obiekt `Shape`).

Jeśli utworzymy instancję klasy `Circle`, tak jak w poniższym przykładzie, instancja odziedziczy łańcuch prototypów klasy `Circle`:

```
// Create an instance of the Circle class.  
myCircle = new Circle();
```

Przypominamy, że w przykładzie występowała właściwość o nazwie `visible` jako element klasy `Shape`. W tym przykładzie właściwość `visible` nie istnieje jako element obiektu `myCircle`, lecz jako element obiektu `Shape`, a mimo to następujący wiersz kodu daje w wyniku wartość `true`:

```
trace(myCircle.visible); // output: true
```

Środowisko wykonawcze, analizując łańcuch prototypów, jest w stanie ustalić, że obiekt `myCircle` dziedziczy właściwość `visible`. Podczas wykonywania tego kodu środowisko wykonawcze we właściwościach obiektu `myCircle` poszukuje właściwości o nazwie `visible`, ale takiej właściwości nie znajduje. Następnie szuka właściwości w obiekcie `Circle.prototype`, ale nadal nie znajduje tam właściwości o nazwie `visible`. Przechodząc coraz wyżej w łańcuchu prototypów, znajduje właściwość `visible` zdefiniowaną w obiekcie `Shape.prototype` i wyświetla wartość tej właściwości.

W celu zapewnienia prostoty wiele szczegółów i zawiłości łańcucha prototypów zostało pominiętych. Celem jest udostępnienie takiej ilości informacji, która umożliwia zrozumienie działania modelu obiektów ActionScript 3.0.

ActionScript 2.0

W języku ActionScript 2.0 wprowadzono nowe słowa kluczowe, takie jak `class`, `extends`, `public` i `private`, które pozwalały na definiowanie klas w sposób naturalny dla programistów posługujących się językami opartymi na klasach, takimi jak Java i C++. Należy jednak podkreślić, że zasadniczy mechanizm dziedziczenia w językach ActionScript 1.0 oraz ActionScript 2.0 nie uległ zmianie. W języku ActionScript 2.0 dodano jedynie nowe elementy składni służące do definiowania klas. Łańcuch prototypów działa tak samo w obu wersjach języka.

Nowe elementy składni wprowadzone w języku ActionScript 2.0, przedstawione w poniższym fragmencie kodu, umożliwiają definiowanie klas w sposób uważany przez wielu programistów za bardziej intuicyjny:

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

Ponadto w języku ActionScript 2.0 wprowadzono deklaracje typów pozwalające na sprawdzanie zgodności typów na etapie kompilacji. A zatem właściwość `visible` z poprzedniego przykładu można byłoby zadeklarować jako przyjmującą tylko wartości typu `Boolean`. Nowe słowo kluczowe `extends` upraszcza proces tworzenia podklasy. W poniższym przykładzie dwuetapowy proces wymagany w języku ActionScript 1.0 jest realizowany w jednym kroku przy użyciu słowa kluczowego `extends`:

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
    }
}
```

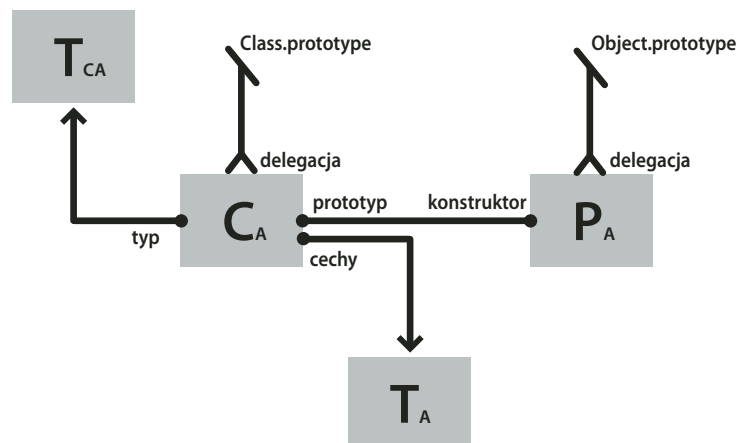
Konstruktor jest teraz deklarowany jako część definicji klasy, a właściwości klasy `id` oraz `radius` muszą być zadeklarowane jawnie.

W języku ActionScript 2.0 umożliwiono definiowanie interfejsów. Interfejsy pozwalają zoptymalizować strukturę programów zorientowanych obiektowo, ponieważ definiują formalnie protokoły komunikacji między obiektami.

Obiekt klasy w języku ActionScript 3.0

W powszechnie spotykanym modelu programowania zorientowanego obiektowo, zwykle kojarzonym z językami Java i C++, do definiowania typów obiektów używa się klas. W językach programowania opartych na tym modelu z reguły klasy służą także do budowania instancji typów danych definiowanych przez te klasy. W języku ActionScript klasy mają oba te zastosowania, ale ponieważ język wywodzi się z modelu opartego na prototypach, ma pewne interesujące cechy. W języku ActionScript definicja każdej klasy jest specjalnym obiektem klasy, który może udostępniać zarówno zachowania, jak i określone stany. Jednak dla wielu programistów korzystających z języka ActionScript to rozróżnienie nie ma praktycznego znaczenia. Język ActionScript 3.0 zaprojektowano w taki sposób, aby umożliwiał tworzenie zaawansowanych aplikacji zorientowanych obiektowo bez konieczności używania tych specjalnych obiektów klas, a nawet bez świadomości ich istnienia.

Poniższy schemat ilustruje strukturę obiektu klasy reprezentującego prostą klasę A zdefiniowaną za pomocą instrukcji `class A {}`:



Każdy prostokąt na diagramie reprezentuje jeden obiekt. Każdy obiekt na diagramie ma indeks A oznaczający, że należy on do klasy A. Obiekt klasy (CA) zawiera odwołania do szeregu innych ważnych obiektów. Obiekt cech instancji (TA) zawiera właściwości instancji zdefiniowane w definicji klasy. Obiekt cech klasy (TCA) reprezentuje wewnętrzny typ klasy i zawiera właściwości statyczne definiowane przez klasę (indeks C to skrót od słowa „class”). Obiekt prototypowy (PA) zawsze oznacza obiekt klasy, z którym był pierwotnie skojarzony, za pośrednictwem właściwości `constructor`.

Obiekt cech

Obiekt cech, który jest nowym elementem środowiska ActionScript 3.0, to rozwiązanie wprowadzone z myślą o wydajności działania kodu. W poprzednich wersjach środowiska ActionScript wyszukiwanie nazw było potencjalnie czasochłonne, ponieważ program Flash Player sprawdzał kolejne ogniwa łańcucha prototypów. W środowisku ActionScript 3.0 wyszukiwanie nazw przebiega znacznie wydajniej i szybciej, ponieważ właściwości dziedziczone są kopiowane z nadklas do obiektów cech podklas.

Obiekt cech nie jest bezpośrednio dostępny dla kodu pisanego przez programistę, ale jego obecność ujawnia się pośrednio poprzez większą szybkość działania kodu i optymalizację zużycia pamięci. Obiekt cech dostarcza maszynie AVM2 szczegółowych informacji o strukturze i zawartości klasy. Na podstawie tych informacji maszyna AVM2 jest w stanie znacznie przyspieszyć wykonywanie kodu, ponieważ często możliwe jest wygenerowanie instrukcji maszynowych uzyskujących dostęp do właściwości lub bezpośrednie wywoływanie metod bez czasochłonnego wyszukiwania nazw.

Dzięki obiektom cech obiekty aplikacji zajmują znacznie mniej miejsca w pamięci niż ich odpowiedniki w starszych wersjach środowiska ActionScript. Na przykład, jeśli klasa jest zapieczętowana (tj. nie jest zadeklarowana jako dynamiczna), to instancja tej klasy nie musi zawierać tabeli mieszania dla dynamicznie dodawanych właściwości, a obiekt tej klasy zajmuje niewiele więcej miejsca niż wskaźnik do obiektu cech i pola na właściwości ustalone zdefiniowane w klasie. W rezultacie obiekt, który w środowisku ActionScript 2.0 zajmował 100 bajtów pamięci, w środowisku ActionScript 3.0 zajmuje zaledwie 20 bajtów.

Uwaga: Obiekt cech jest wewnętrznym rozwiązaniem implementacyjnym i nie ma gwarancji, że w przyszłych wersjach środowiska ActionScript nie zostanie zmieniony lub nawet całkowicie wyeliminowany.

Obiekt prototypowy

Każdy obiekt klasy w języku ActionScript ma właściwość `prototype`, która zawiera odwołanie do obiektu prototypowego klasy. Obiekt prototypowy jest zaszłością wywodzącą się z korzeni ActionScript jako języka opartego na prototypach. Więcej informacji zawiera dokumentacja „[Historia elementów programowania zorientowanego obiektowo w języku ActionScript](#)” na stronie 115.

Właściwość `prototype` jest przeznaczona tylko do odczytu, co oznacza, że nie można jej zmodyfikować tak, aby wskazywała na inne obiekty. Zachowuje się zatem inaczej niż właściwość `prototype` klas w poprzednich wersjach języka ActionScript, które umożliwiały zmianę przypisania prototypu tak, aby właściwość ta wskazywała na inną klasę. Mimo że właściwość `prototype` jest przeznaczona tylko do odczytu, obiekt prototypowy, do którego ta właściwość się odwołuje, może być modyfikowany. Innymi słowy, do obiektu prototypowego można dodawać nowe właściwości. Właściwości dodawane do obiektu prototypowego są wspólne dla wszystkich instancji klasy.

Łańcuch prototypów, który w poprzednich wersjach języka ActionScript był jedynym mechanizmem dziedziczenia, pełni w języku ActionScript 3.0 jedynie drugorzędną rolę. Podstawowy mechanizm dziedziczenia, tj. dziedziczenie właściwości ustalonych, jest realizowany wewnętrznie przez obiekty cech. Właściwość ustalona do zmienna lub metoda zdefiniowana jako element definicji klasy. Dziedziczenie właściwości ustalonych nazywane jest także dziedziczeniem klas, ponieważ ten właśnie mechanizm dziedziczenia jest powiązany z takimi słowami kluczowymi, jak `class`, `extends` i `override`.

Łańcuch prototypów stanowi alternatywny mechanizm dziedziczenia — bardziej dynamiczny niż dziedziczenie właściwości ustalonych. Możliwe jest dodawanie właściwości do obiektu prototypowego klasy nie tylko w definicji klasy, lecz również za pośrednictwem właściwości `prototype` obiektu klasy. Należy jednak zwrócić uwagę, że w wypadku wybrania trybu ścisłego kompilatora nie będzie możliwe uzyskanie dostępu do właściwości dodanych do obiektu prototypowego, jeśli klasa nie będzie zadeklarowana ze słowem kluczowy `dynamic`.

Za przykład klasy z wieloma właściwościami powiązanymi z obiektem prototypowym jest klasa `Object`. Metody `toString()` i `valueOf()` klasy `Object` są w istocie funkcjami przypisanymi właściwościom obiektu prototypowego tej klasy. Poniżej przedstawiono teoretyczną deklarację tych metod (w praktyce deklaracja ta jest nieco inna z uwagi na szczegóły implementacyjne):

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Jak już wcześniej wspomniano, możliwe jest powiązanie właściwości z obiektem prototypowym klasy poza definicją klasy. Na przykład metoda `toString()` mogłaby być zdefiniowana poza definicją klasy `Object`, w następujący sposób:

```
Object.prototype.toString = function()
{
    // statements
};
```

Jednak w przeciwieństwie do dziedziczenia właściwości ustalonych dziedziczenie oparte na prototypach nie wymaga użycia słowa `override` przy przesłanianiu metody w podklasie. Na przykład, jeśli chcemy przeddefiniować metodę `valueOf()` w podklasie obiektu `Object`, mamy do wyboru trzy możliwości. Po pierwsze, możemy zdefiniować metodę `valueOf()` w obiekcie prototypowym podklasy, wewnątrz definicji tej podklasy. Poniższy kod tworzy podklasę klasy `Object` o nazwie `Foo` i zmienia definicję metody `valueOf()` w obiekcie prototypowym `Foo` w ramach definicji klasy. Ponieważ każda klasa dziedziczy z klasy `Object`, nie jest konieczne użycie słowa kluczowego `extends`.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Po drugie, możemy zdefiniować metodę `valueOf()` w obiekcie prototypowym `Foo` poza definicją klasy, co ilustruje poniższy kod:

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Po trzecie, możemy zdefiniować ustaloną właściwość o nazwie `valueOf()` jako element klasy `Foo`. Ta technika polega na jednoczesnym zastosowaniu dziedziczenia właściwości ustalonych i dziedziczenia opartego na prototypach. W każdej podklasie klasy `Foo`, w której chcielibyśmy zmienić definicję właściwości `valueOf()`, musimy użyć słowa kluczowego `override`. Poniższy kod przedstawia definicję `valueOf()` jako właściwości ustalonej w klasie `Foo`:

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

Przestrzeń nazw AS3

Istnienie dwóch odrębnych mechanizmów dziedziczenia, tj. dziedziczenia właściwości ustalonych i dziedziczenia opartego na prototypach, stwarza pewne potencjalne problemy ze zgodnością właściwości i metod klas podstawowych języka. Robocza specyfikacja ECMAScript, na której oparty jest język ActionScript, zakłada stosowanie dziedziczenia w oparciu o prototypy, co oznacza że właściwości i metody klas podstawowych języka są zdefiniowane w swoich obiektach prototypowych. Z drugiej strony, zgodność z wywołaniami języka ActionScript 3.0, w których stosowane jest dziedziczenie właściwości ustalonych, wymaga zdefiniowania właściwości i metod klas podstawowych za pomocą słów kluczowych `const`, `var` i `function`. Co więcej, użycie klas ustalonych zamiast ich wersji prototypowych może prowadzić do znacznego przyspieszenia wykonywania kodu.

W języku ActionScript 3.0 problem ten rozwiązano, stosując w klasach podstawowych języka zarówno dziedziczenie oparte na prototypach, jak i dziedziczenie właściwości ustalonych. Każda klasa podstawowa języka zawiera dwa zestawy właściwości i metod. Jeden zestaw jest zdefiniowany w obiekcie prototypowym w celu zapewnienia zgodności ze specyfikacją ECMAScript, a drugi zestaw jest zdefiniowany przy użyciu właściwości ustalonych w przestrzeni nazw AS3.0 w celu zapewnienia zgodności z językiem ActionScript 3.0.

Przestrzeń nazw AS3 udostępnia wygodny mechanizm wyboru jednego z tych zestawów właściwości i metod. Jeśli w programie nie jest używana przestrzeń nazw AS3, instancja klasy podstawowej języka dziedziczy właściwości i metody zdefiniowane w obiekcie prototypowym klasy podstawowej. Jeśli przestrzeń nazw AS3 jest używana, instancja klasy podstawowej dziedziczy wersje z tej przestrzeni nazw (AS3), ponieważ właściwości ustalone mają zawsze pierwszeństwo przed prototypowymi. Innymi słowy, gdy tylko dostępna jest właściwość ustalona, jest używana w miejsce właściwości prototypowej o tej samej nazwie.

Możliwe jest selektywne stosowanie właściwości i metod z przestrzeni AS3 poprzez kwalifikowanie nazw przestrzenią AS3. Na przykład w poniższym kodzie zastosowano wersję metody `Array.pop()` z przestrzeni AS3:

```
var nums:Array = new Array(1, 2, 3);
nums.AS3:pop();
trace(nums); // output: 1,2
```

Alternatywnym rozwiązaniem byłoby użycie dyrektywy `use namespace` w celu otwarcia przestrzeni nazw AS3 dla wszystkich definicji w bloku kodu. Na przykład w poniższym kodzie zastosowano dyrektywę `use namespace` w celu otwarcia przestrzeni nazw AS3 zarówno dla metody `pop()`, jak i metody `push()`:

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

Środowisko ActionScript 3.0 udostępnia opcje kompilatora właściwe dla każdego zestawu właściwości, pozwalając na zastosowanie przestrzeni nazw AS3 do całego programu. Opcja kompilatora `-as3` oznacza przestrzeń nazw AS3, a opcja kompilatora `-es` oznacza dziedziczenie w oparciu o prototypy (skrót `es` pochodzi od nazwy ECMAScript). Aby otworzyć przestrzeń nazw AS3 dla całego programu, należy ustawić opcję kompilatora `-as3` na `true`, a opcję kompilatora `-es` na `false`. Aby stosować wersje prototypowe, należy ustawić opcje kompilatora na przeciwne wartości. Domyślna konfiguracja kompilatora w programach Flash Builder i Flash Professional to: `-as3 = true` oraz `-es = false`.

Jeśli planowane jest rozszerzanie klas podstawowych języka i przesłanianie metod, należy wziąć pod uwagę wpływ przestrzeni nazw AS3 na wymagany sposób deklarowania metod przesłoniętych. Jeśli używana jest przestrzeń nazw AS3, każda metoda przesłaniająca metodę klasy podstawowej języka musi również deklarować przestrzeń nazw AS3 wraz z atrybutem `override`. Jeśli przestrzeń nazw AS3 nie jest używana i planowane jest przededefiniowanie metody klasy podstawowej w podklasie, nie należy deklarować przestrzeni nazw AS3 ani słowa kluczowego `override`.

Przykład: GeometricShapes

Przykładowa aplikacja `GeometricShapes` ilustruje szereg pojęć i mechanizmów programowania zorientowanego obiektowo w języku ActionScript 3.0, a w szczególności:

- Definiowanie klas.
- Rozszerzanie klas.
- Polimorfizm i użycie słowa kluczowego `override`.
- Definiowanie, rozszerzanie i implementowanie interfejsów.

Zawiera także „metodę fabrykującą”, która tworzy instancje klas, i demonstruje deklarowanie wartości zwracanej jako instancji interfejsu oraz uniwersalne traktowanie zwróconego w ten sposób obiektu.

Aby pobrać pliki tej przykładowej aplikacji, należy przejść na stronę www.adobe.com/go/learn_programmingAS3samples_flash_pl. Pliki aplikacji GeometricShapes znajdują się w folderze Samples/GeometricShapes. Aplikacja składa się z następujących plików:

File	Opis
GeometricShapes.mxml lub GeometricShapes fla	Główny plik aplikacji w formacie Flash (FLA) lub Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	Interfejs bazowy definiujący metody, które powinny być implementowane przez klasy aplikacji GeometricShapes.
com/example/programmingas3/geometricshapes/IPolygon.as	Interfejs definiujący metody, które powinny być implementowane przez klasy aplikacji GeometricShapes dla wieloboków.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Typ kształtu geometrycznego, który ma boki równej długości rozmieszczone symetrycznie wokół środka kształtu.
com/example/programmingas3/geometricshapes/Circle.as	Typ kształtu geometrycznego definiujący koło.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Podklasa klasy RegularPolygon definiująca trójkąt równoboczny.
com/example/programmingas3/geometricshapes/Square.as	Podklasa klasy RegularPolygon definiująca prostokąt o równych bokach.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Klasa zawierająca metodę fabrykującą kształty o zadanym typie i rozmiarze.

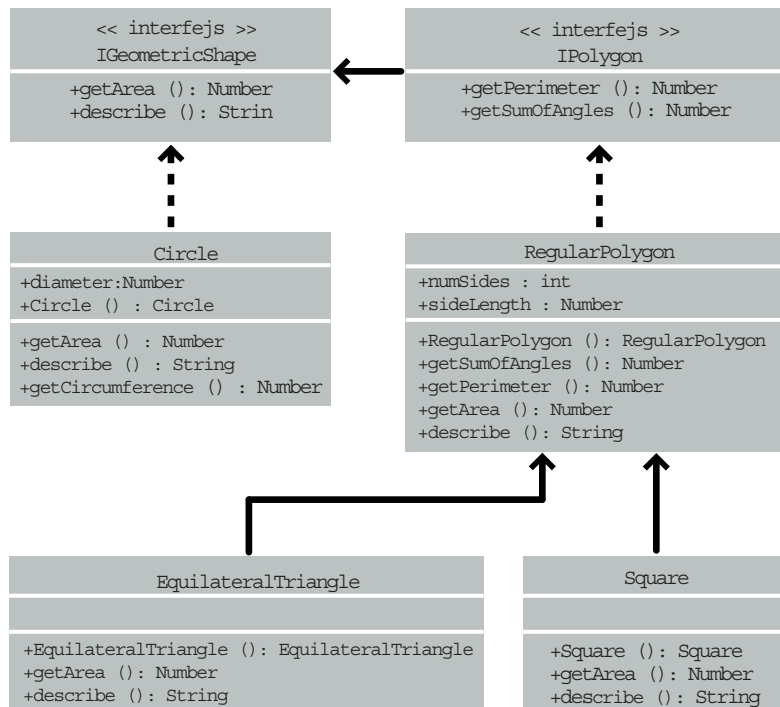
Definiowanie klas dla aplikacji GeometricShapes

Aplikacja GeometricShapes umożliwia użytkownikowi określenie typu i rozmiaru kształtu geometrycznego. W odpowiedzi wyświetla opis kształtu, jego pole powierzchni oraz obwód.

Interfejs użytkownika aplikacji jest trywialny i obejmuje kilka elementów sterujących do wybierania typu kształtu, określania rozmiaru i wyświetlania opisu. Najbardziej interesująca warstwa tej aplikacji pozostaje niewidoczna dla użytkownika — jest nią struktura klas i interfejsów.

Aplikacja operuje na kształtach geometrycznych, ale nie wyświetla ich graficznie.

Klasy i interfejsy, które definiują kształty geometryczne w tym przykładzie zostały przedstawione na następującym schemacie przy użyciu notacji UML (Unified Modeling Language):



Klasy z przykładu GeometricShapes

Definiowanie wspólnych zachowań przy użyciu interfejsów

Aplikacja GeometricShapes operuje na trzech typach kształtów: kołach, kwadratach i trójkątach równobocznych. Struktura klas aplikacji GeometricShapes zaczyna się od bardzo prostego interfejsu IGeometricShape, w którym zadeklarowane są metody wspólne dla wszystkich trzech typów kształtów:

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

Interfejs definiuje dwie metody: metodę `getArea()`, która oblicza i zwraca pole powierzchni kształtu, oraz metodę `describe()`, która komponuje opis tekstowy z właściwości kształtu.

Wskazana jest również znajomość długości obwodu każdego z kształtów. Jednak obwód koła oblicza się w nietypowy sposób, a zatem zachowanie koła odbiega od zachowania pozostałych kształtów, tj. trójkąta i kwadratu. Natomiast trójkąty, kwadraty i inne wieloboki są do siebie na tyle podobne, że celowe jest zdefiniowanie nowego interfejsu — IPolygon — specjalnie dla nich. Interfejs IPolygon także jest dość prosty:

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Ten interfejs definiuje dwie metody wspólne dla wszystkich wieloboków: metodę `getPerimeter()`, która sumuje długość wszystkich boków, oraz metodę `getSumOfAngles()`, która sumuje wszystkie kąty wewnętrzne.

Interfejs `IPolygon` rozszerza interfejs `IGeometricShape`, co oznacza, że każda klasa implementująca interfejs `IPolygon` musi zawierać deklaracje wszystkich czterech metod — dwóch z interfejsu `IGeometricShape` i dwóch z interfejsu `IPolygon`.

Definiowanie klas kształtów

Po zidentyfikowaniu metod wspólnych dla wszystkich kształtów możemy przystąpić do definiowania klas samych kształtów. Pod względem liczby klas, jakie należy zaimplementować, najprostszym kształtem jest koło — klasa `Circle`:

```
package com.example.programingas3.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;
        }
    }
}
```

Klasa `Circle` implementuje interfejs `IGeometricShape`, a więc musi zawierać zarówno kod metody `getArea()`, jak i kod metody `describe()`. Ponadto definiuje metodę `getCircumference()` występującą tylko w klasie `Circle`. Klasa `Circle` zawiera także deklarację właściwości `diameter`, której nie mają klasy wieloboków.

Pozostałe dwa typy kształtów — kwadraty i trójkąty równoboczne — mają kilka innych cech wspólnych: mają boki równej długości oraz wspólne wzory na obliczanie obwodu i sumy kątów wewnętrznych. W istocie te wspólne wzory będą miały zastosowanie do wszystkich innych wieloboków foremnych, które być może zdefiniujemy w przyszłości.

Klasa `RegularPolygon` jest nadklasą dla klasy `Square` i klasy `EquilateralTriangle`. Nadklasa pozwala na zdefiniowanie w jednym miejscu wszystkich wspólnych metod, przez co nie trzeba definiować ich osobno w każdej podklasie. Oto kod klasy `RegularPolygon`:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

W klasie `RegularPolygon` najpierw deklarowane są dwie właściwości wspólne dla wszystkich wieloboków foremnych: długość boku (właściwość `sideLength`) oraz liczba boków (właściwość `numSides`).

Klasa `RegularPolygon` implementuje interfejs `IPolygon` i zawiera deklaracje wszystkich czterech metod określonych w interfejsie `IPolygon`. Dwie z tych metod — `getPerimeter()` oraz `getSumOfAngles()` — implementuje przy użyciu wspólnych wzorów.

Ponieważ wzór używany w metodzie `getArea()` jest różny w poszczególnych kształtach, wersja tej metody w klasie bazowej nie może zawierać wspólnej logiki dziedziczonej przez metody w podklasach. Dlatego metoda ta zwraca wartość domyślną 0, która oznacza, że pole powierzchni nie zostało obliczone. Aby prawidłowo obliczać pola powierzchni poszczególnych kształtów, podklasy klasy `RegularPolygon` muszą przesłaniać metodę `getArea()`.

Poniższy kod klasy `EquilateralTriangle` ilustruje przesłanianie metody `getArea()`:

```
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;
        }
    }
}
```

Słowo kluczowe `override` oznacza, że metoda `EquilateralTriangle.getArea()` celowo przesłania metodę `getArea()` z nadklasy `RegularPolygon`. Po wywołaniu metody `EquilateralTriangle.getArea()` pole powierzchni jest obliczane według wzoru zawartego w powyższym przykładzie, a kod w metodzie `RegularPolygon.getArea()` nigdy nie jest wykonywany.

Natomiast klasa `EquilateralTriangle` nie definiuje własnej wersji metody `getPerimeter()`. Wywołanie metody `EquilateralTriangle.getPerimeter()` jest przekazywane w górę w łańcuchu dziedziczenia i wykonywany jest kod metody `getPerimeter()` z nadklasy `RegularPolygon`.

W konstruktorze `EquilateralTriangle()` użyto instrukcji `super()` w celu jawnego wywołania konstruktora `RegularPolygon()` z nadklasy. Gdyby oba konstruktory miały ten sam zestaw parametrów, można byłoby pominąć konstruktor `EquilateralTriangle()`, z zamiast niego wykonywany byłby konstruktor `RegularPolygon()`. Jednak konstruktor `RegularPolygon()` potrzebuje dodatkowego parametru `numSides`. Dlatego konstruktor `EquilateralTriangle()` zawiera wywołanie `super(len, )`, które przekazuje parametr wejściowy `len` oraz wartość 3 oznaczającą, że trójkąt ma 3 boki.

Metoda `describe()` również wykorzystuje instrukcję `super()`, ale w inny sposób. Wykorzystuje ją w celu wywołania wersji nadklasy `RegularPolygon` metody `describe()`. Metoda `EquilateralTriangle.describe()` najpierw przypisuje zmiennej `desc` typu `String` informację o typie kształtu. Następnie pobiera wynik metody `RegularPolygon.describe()`, wywołując `super.describe()`, po czym dołącza ten wynik do ciągu znaków `desc`.

Klasy `Square` nie będziemy tutaj szczegółowo omawiać, ale jest ona podobna do klasy `EquilateralTriangle` i zawiera konstruktor oraz własne implementacje metod `getArea()` i `describe()`.

Polimorfizm i metoda fabrykująca

Zestaw klas we właściwy sposób implementujących interfejsy i korzystających z dziedziczenia można zastosować na wiele różnych sposobów. Na przykład wszystkie opisane dotąd klasy kształtów albo implementują interfejs `IGeometricShape`, albo rozszerzają nadklasę, która implementuje ten interfejs. Jeśli zatem zdefiniujemy zmienną jako instancję `IGeometricShape`, to aby wywołać jej metodę `describe()` nie musimy wiedzieć, czy jest to faktycznie instancja klasy `Circle`, czy `Square`.

Ilustruje to poniższy przykład kodu:

```
var myShape:IGeometricShape = new Circle(100);  
trace(myShape.describe());
```

Wywołanie `myShape.describe()` powoduje wykonanie metody `Circle.describe()`, ponieważ, choć zmienna jest zdefiniowana jako instancja interfejsu `IGeometricShape`, należy do klasy `Circle`.

Przykład ten prezentuje istotę polimorfizmu: to samo wywołanie metody powoduje wykonanie różnego kodu w zależności od tego, do jakiej klasy należy obiekt wywoływanej metody.

W aplikacji `GeometricShapes` zastosowano polimorfizm oparty na interfejsach przy użyciu uproszczonej wersji wzorca projektowego znanego jako wzorzec metody fabrykującej. Termin *metoda fabrykująca* oznacza funkcję zwracającą obiekt, którego typ danych lub zawartość może być różna w zależności od kontekstu.

W przedstawionej tutaj klasie `GeometricShapeFactory` zdefiniowana jest metoda fabrykująca o nazwie `createShape()`:

```
package com.example.programingas3.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();
        }
    }
}
```

Metoda fabrykująca `createShape()` umożliwia konstruktorom podklas kształtów definiowanie szczegółów tworzonych przez nie instancji, a przy tym zwracanie nowych obiektów jako instancji interfejsu `IGeometricShape`, która aplikacja może traktować w sposób uniwersalny.

Metoda `describeShape()` w poprzednim przykładzie ilustruje zastosowanie metody fabrykującej do uzyskiwania ogólnych odwołań do ściślej sprecyzowanych obiektów. Aplikacja może uzyskać opis nowo utworzonego obiektu `Circle` w następujący sposób:

```
GeometricShapeFactory.describeShape("Circle", 100);
```

Następnie metoda `describeShape()` wywołuje metodę fabrykującą `createShape()` z tymi samymi parametrami i zapisuje nowy obiekt `Circle` w zmiennej statycznej `currentShape`, którą zadeklarowano jako obiekt `IGeometricShape`. Teraz wywoływana jest metoda `describe()` obiektu `currentShape`, a wywołanie to jest automatycznie tłumaczone na wywołanie metody `Circle.describe()`, która zwraca szczegółowy opis koła.

Udoskonalenia aplikacji przykładowej

Prawdziwy potencjał interfejsów i dziedziczenia ujawnia się przy udoskonalaniu lub modyfikowaniu aplikacji.

Założmy, że chcemy dodać do aplikacji przykładowej nowy kształt, pięciobok. Utworzylibyśmy wówczas klasę Pentagon rozszerzającą klasę RegularPolygon, z własną wersją metod `getArea()` i `describe()`. Następnie dodalibyśmy opcję Pentagon do odpowiedniego pola w interfejsie użytkownika aplikacji. I to wszystko. Klasa Pentagon automatycznie uzyskałaby funkcjonalność metod `getPerimeter()` oraz `getSumOfAngles()` w wyniku dziedziczenia z klasy RegularPolygon. Ponieważ klasa Pentagon dziedziczyłaby z klasy implementującej interfejs IGeometricShape, instancję klasy Pentagon można byłoby traktować jak instancję interfejsu IGeometricShape. Oznacza to, że aby dodać nowy typ kształtu, nie trzeba zmieniać sygnatur metod w klasie GeometricShapeFactory (a w konsekwencji nie trzeba modyfikować kodu, w którym klasa GeometricShapeFactory jest używana).

Czytelnik może w ramach samodzielnego ćwiczenia dodać klasę Pentagon do przykładowej aplikacji GeometricShapes, aby przekonać się, że interfejsy i dziedziczenie zmniejszają pracochłonność rozbudowywania aplikacji.