helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

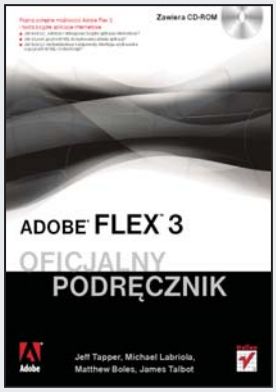
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Adobe Flex 3. Oficjalny podręcznik



Autorzy: Jeff Tapper, Michael Labriola, Matthew Boles, James Talbot
Tłumaczenie: Andrzej Gońda, Aleksander Lamża
ISBN: 978-83-246-1966-5
Tytuł oryginału: Adobe Flex 3: Training from the Source
Format: B5, stron: 648

Poznaj potężne możliwości Adobe Flex 3 i twórz bogate aplikacje internetowe

- Jak tworzyć, wdrażać i debugować bogate aplikacje internetowe?
- Jak używać języka MXML do budowania układu aplikacji?
- Jak tworzyć niestandardowe komponenty interfejsu użytkownika w językach MXML i ActionScript?

Adobe Flex 3 to odpowiedź na zapotrzebowanie programistów w zakresie tworzenia bogatych aplikacji internetowych za pomocą przyjaznych i intuicyjnych narzędzi. Na Flex składają się m.in.: ActionScript 3.0, zwiększający wydajność i prostotę programowania; Flash Player 9, pozwalający na szybszą pracę i wykorzystujący mniejszą ilość pamięci; Flex Builder 3, udostępniający m.in. świetne narzędzia do usuwania błędów i promujący najlepsze praktyki kodowania oraz programowania aplikacji.

Książka „Adobe Flex 3. Oficjalny podręcznik" poprowadzi Cię krok po kroku przez proces budowania dynamicznych, interaktywnych aplikacji. Zawiera ona szczegółowy, wyczerpujący, praktyczny kurs tworzenia i projektowania architektury bogatych aplikacji internetowych (RIA) w językach MXML i ActionScript 3.0. Dzięki temu podręcznikowi nauczysz się pracować ze złożonymi zbiorami danych, tworzyć własne zdarzenia niestandardowe, stosować style i skórki, a także instalować aplikacje w sieci lub komputerze. Poznasz wszystkie możliwości Fleksa i będziesz umiał zastosować w praktyce nowoczesne narzędzia, a także z łatwością korzystać z innowacyjnych technologii.

- Technologie bogatych aplikacji internetowych
- Pulpit programu Flex Builder 3
- Obsługa zdarzeń i struktury danych
- Używanie zdalnych danych z kontrolkami
- Tworzenie komponentów w języku MXML
- Repetery ze zbiorami danych
- Tworzenie niestandardowych komponentów w języku ActionScript 3.0
- Stosowanie siatek danych i generatorów elementów
- Wprowadzanie nawigacji
- Formatowanie i walidacja danych
- Zarządzanie historią i głębokie linkowanie
- Wizualizowanie danych
- Wdrażanie aplikacji Fleksa

Skorzystaj z indywidualnych warsztatów Flex 3 i zbuduj samodzielnie wyjątkową aplikację internetową!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Spis treści

O autorach **15**

Wstęp **17**

Wprowadzenie **19**

Lekcja 1 Wprowadzenie do bogatych aplikacji internetowych **29**

Rozwój aplikacji komputerowych ... 29
Odejście od architektury opartej na stronie ... 31
Zalety bogatych aplikacji internetowych ... 33
Menedżerowie przedsiębiorstw ... 33
Przedsiębiorstwa branży IT ... 33
Użytkownicy końcowi ... 33
Technologie bogatych aplikacji internetowych ... 33
Asynchroniczny JavaScript i XML (AJAX) ... 34
Flash ... 35
Flex ... 35
Windows Presentation Foundation, XAML, Silverlight i Expression ... 36
Czego się nauczyłeś? ... 37

Lekcja 2 Zaczynamy **39**

Wstęp do programowania aplikacji Fleksa ... 40
Tworzenie projektu i aplikacji MXML ... 40
Pulpit programu Flex Builder 3 ... 43
Uruchamianie aplikacji ... 46
Tworzenie drugiej strony aplikacji w trybie projektowania ... 52
Przygotowania do następnej lekcji ... 57
Czego się nauczyłeś? ... 58

Lekcja 3 Projektowanie interfejsu użytkownika **61**

Kontenery ... 61
Tworzenie układu aplikacji e-commerce w trybie projektowania ... 62
Praca z układami opartymi na więzach ... 68
Wiązanie do kontenera nadrzędnego ... 69
Użycie zaawansowanych więzów ... 74

Praca ze stanami widoku 76
Sterowanie stanami widoku 78
Projektowanie aplikacji w trybie edycji kodu źródłowego 80
Dodawanie stanów widoku i sterowanie nimi za pomocą MXML 82
Czego się nauczyłeś? 86

Lekcja 4 Stosowanie prostych kontrolek 89

Wprowadzenie do prostych kontrolek 90
Wyświetlanie obrazów 91
Tworzenie widoku szczegółów 94
Korzystanie z wiązania danych do powiązania struktury danych z prostą kontrolką 97
Wykorzystywanie kontenera układu formularzy do umieszczania prostych kontrolek w aplikacji 99
Dodawanie przycisków radiowych i pól daty do aplikacji Dashboard 103
Czego się nauczyłeś? 105

Lekcja 5 Obsługa zdarzeń i struktury danych 107

Zrozumienie obsługi zdarzeń 108
Prosty przykład 108
Obsługa zdarzenia przez funkcję języka ActionScript 109
Przekazywanie danych podczas wywołania funkcji uchwytu zdarzenia 110
Tworzenie struktury danych w zdarzeniu creationComplete 111
Używanie danych z obiektu Event 114
Tworzenie własnej klasy języka ActionScript 118
Tworzenie obiektu wartości 118
Tworzenie metody kreującej obiekt 123
Tworzenie klas koszyka na zakupy 125
Czego się nauczyłeś? 129

Lekcja 6 Używanie zdalnych danych XML z kontrolkami 131

Pobieranie danych XML za pośrednictwem HTTPService 132
Tworzenie obiektu HTTPService 132
Wywoływanie metody send() 133
Używanie zwróconych danych 133
Zagadnienia bezpieczeństwa 134
Pobieranie danych XML za pośrednictwem HTTPService 135
Wypełnianie ArrayCollection danymi z HTTPService 137
Używanie obiektów ArrayCollection 137
Używanie kolekcji w roli dostawców danych 138
Wypełnianie kontrolki ComboBox i programowe dodawanie opcji 140
Używanie danych XML z kontrolką Tree 143
Pojęcie operatorów E4X 143
Wypełnianie kontrolki Tree danymi XML 148
Pobieranie danych XML i przekształcanie ich w kolekcję ArrayCollection obiektów użytkownika 154

Stosowanie powiązania danych ze złożonymi strukturami danych 157
Przekształcanie danych koszyka na zakupy 159
Dodawanie elementów do koszyka 159
Sortowanie elementów kolekcji ArrayCollection 160
Dodawanie elementu lub aktualizowanie ilości 162
Korzystanie z kursora do umieszczania elementu koszyka na zakupy 164
Dodawanie przycisku Remove (usuń) 171
Czego się nauczyłeś? 172

Lekcja 7 Tworzenie komponentów w języku MXML 175

Wprowadzenie do komponentów języka MXML 176
Podstawy tworzenia niestandardowych komponentów 176
Tworzenie własnego komponentu krok po kroku 177
Wykorzystanie niestandardowych komponentów w architekturze aplikacji 179
Tworzenie komponentu aktualizującego i usuwającego informacje o produkcie oraz jego egzemplarza 180
Wyświetlanie informacji o produkcie po kliknięciu przycisków aktualizacji i usuwania 184
Tworzenie kolejnego obiektu wartości 189
Tworzenie komponentu zarządzającego danymi dla wszystkich trzech aplikacji 191
Używanie nowego komponentu zarządzającego danymi 197
Wprowadzanie funkcji dodawania produktu 200
Tworzenie i używanie komponentów dla aplikacji Dashboard 201
Czego się nauczyłeś? 203

Lekcja 8 Używanie kontrolek i komponentów Repeater ze zbiorami danych 205

Używanie zbiorów danych 206
Komponenty HorizontalList i TileList 207
Wprowadzenie labelFunction 208
Wprowadzenie właściwości itemRenderer 209
Wyświetlanie kategorii za pomocą elementów HorizontalList i itemRenderer 210
Wyświetlanie informacji o produktach spożywczych należących do określonej kategorii 214
Używanie komponentu Repeater do wykonania pętli przeszukującej zbiór danych 214
Pobieranie danych z komponentów Repeater 215
Adresowanie komponentów utworzonych przez Repeater 217
Wyjaśnienie różnic w wydajności komponentów TileList i Repeater 218
Wyświetlanie informacji o artykułach spożywczych należących do określonej kategorii 218
Kodowanie stanów wyświetlających szczegółowe informacje o produkcie 224
Umieszczanie produktów w koszyku na zakupy 227
Czego się nauczyłeś? 229

Lekcja 9 Używanie zdarzeń niestandardowych 232
Korzyści ze stosowania luźno powiązanej architektury 232
Wysyłanie zdarzeń 233
Deklarowanie zdarzeń komponentu 235
Kiedy używa się klas niestandardowych zdarzeń 236
Tworzenie i używanie CategoryEvent 237
Tworzenie i używanie klasy ProductEvent 240
Używanie ProductEvent do usuwania produktu z koszyka 244
Wykorzystanie ProductEvent do dodawania produktu do koszyka 246
Zrozumienie przepływu i bąbelkowania zdarzeń 247
Czego się nauczyłeś? 252

Lekcja 10 Tworzenie niestandardowych komponentów w języku ActionScript 3.0 255
Wprowadzenie do tworzenia komponentów w języku ActionScript 3.0 256
Tworzenie struktury klasy 257
Nadpisywanie metody createChildren() 259
Tworzenie przycisku w języku ActionScript 260
Używanie metody addChild() do dodawania przycisku do komponentu 260
Co to jest chrom i rawChildren 261
Używanie metod addChild() i rawChildren do dodawania elementów do chromu 262
Ustalanie rozmiarów i pozycji komponentów we Fleksie 265
Zrozumienie metody measure() 267
Nadpisywanie metody updateDisplayList() 267
Czego się nauczyłeś? 273

Lekcja 11 Stosowanie siatek danych i rendererów elementów 276
Wprowadzenie do siatek danych i rendererów elementów 277
Dodawanie standardowej siatki danych do komponentu ChartPod 277
Dodawanie wywołań HTTPService do aplikacji Dashboard 278
Wyświetlanie koszyka na zakupy w siatce danych 282
Dodawanie kontrolek edycji w siatce do DataGridColumn 284
Tworzenie renderera elementów służącego do wyświetlania informacji o produkcie 285
Tworzenie lokalnego renderera elementów w celu wyświetlenia przycisku Remove 287
Aktualizowanie ShoppingCartItem funkcjami Set i Get 292
Używanie zaawansowanej siatki danych 293
Sortowanie AdvancedDataGrid 294
Sortowanie w trybie zaawansowanym 295
Nadawanie stylów zaawansowanej siatce danych 296
Grupowanie danych 300
Wyświetlanie danych podsumowujących 305
Czego się nauczyłeś? 313

Lekcja 12 Używanie techniki „przeciągnij i upuść” 315
Wprowadzenie do menedżera przeciągania i upuszczania ... 316
Przeciąganie i upuszczanie pomiędzy dwiema siatkami danych ... 317
Przeciąganie i upuszczanie między siatką danych i listą ... 320
Używanie komponentu z wyłączonym przeciąganiem w operacji przeciągania i upuszczania ... 324
Przeciąganie produktu do koszyka na zakupy ... 329
Czego się nauczyłeś? ... 334

Lekcja 13 Wprowadzanie nawigacji 337
Wprowadzenie do nawigacji ... 337
Używanie komponentu TabNavigator w aplikacji DataEntry ... 340
Dodawanie strony głównej i strony płatności w aplikacji e-commerce ... 343
Przygotowywanie pierwszego etapu procesu płatności wyświetlanego w ViewStack ... 346
Wykorzystanie komponentu ViewStack do finalizacji płatności ... 353
Czego się nauczyłeś? ... 358

Lekcja 14 Formatowanie i walidacja danych 361
Podstawowe informacje o klasach formatujących i walidujących ... 361
Wykorzystanie klasy Formatter do wyświetlania informacji o walucie ... 362
Korzystanie z klas walidatorów ... 366
Sprawdzanie poprawności danych za pomocą wyrażeń regularnych (część 1.) ... 368
Sprawdzanie poprawności danych za pomocą wyrażeń regularnych (część 2.) ... 370
Tworzenie własnej klasy walidatora ... 373
Czego się nauczyłeś? ... 377

Lekcja 15 Zarządzanie historią i głębokie linkowanie 379
Wprowadzenie do zarządzania historią ... 380
Implementowanie zarządzania historią w kontenerze nawigacyjnym ... 382
Tworzenie własnego mechanizmu zarządzania historią ... 383
Wprowadzenie do głębokiego linkowania ... 388
Głębokie linkowanie we Fleksie 3 ... 388
Wykorzystanie techniki głębokiego linkowania w aplikacji ... 388
Czego się nauczyłeś? ... 393

Lekcja 16 Zmiana wyglądu aplikacji 396
Projekty oparte na stylach i skórkach ... 396
Stosowanie stylów ... 397
Nadawanie stylów przez atrybuty znaczników ... 398
Dziedziczenie stylów ... 400
Nadawanie stylów za pomocą znacznika <mx:Style> ... 400
Korzystanie z narzędzi wspomagających pracę ze stylami ... 403
Nadawanie stylów za pomocą plików CSS ... 404

Zmiana stylów w trakcie działania aplikacji 414
Korzyści płynące z wczytywania stylów 414
Tworzenie pliku SWF z arkusza CSS 414
Wczytywanie arkusza stylów za pomocą klasy StyleManager 414
Przesłanianie stylów we wczytanych plikach CSS 415
Tworzenie skórek dla komponentów 415
Skórki graficzne 416
Importowanie skórek utworzonych w narzędziach z pakietu CS3 416
Skórki programistyczne 419
Czego się nauczyłeś? 423

Lekcja 17 Korzystanie z usług sieciowych 426

Komunikacja z serwerem 427
Stosowanie wywołań zdalnego modelu zdarzeń 428
Konfigurowanie aplikacji do lokalnej pracy 428
Wykorzystanie usług sieciowych w aplikacji Dashboard 429
Obsługa wyników dostarczanych przez usługi sieciowe 432
Wywoływanie metod usług sieciowych 434
Wykorzystanie usług sieciowych w aplikacji DataEntry 436
Korzystanie z kreatora Web Service Introspection 441
Korzystanie z wygenerowanego kodu w aplikacjach 443
Refaktoryzacja we Flex Builderze 445
Finalizowanie łączenia wygenerowanego kodu z aplikacją 446
Uaktualnianie i usuwanie produktów 447
Czego się nauczyłeś? 449

Lekcja 18 Dostęp do obiektów po stronie serwera 451

Przesyłanie plików na serwer 452
Integrowanie komponentu FileUpload z aplikacją DataEntry 455
Korzystanie z klasy RemoteObject do zapisywania zamówienia 457
Aktualizowanie argumentów kompilatora Flex 459
Rozgłaszanie zdarzenia potwierdzającego proces zamawiania 460
Tworzenie i wywoływanie obiektów RemoteObject 460
Przekazywanie obiektu ShoppingCart do komponentu Checkout 464
Przywracanie aplikacji do stanu początkowego 464
Mapowanie obiektów ActionScript na obiekty serwera 464
Kreatory dostępu do danych 467
Tworzenie projektu serwera 468
Czego się nauczyłeś? 471

Lekcja 19 Wizualizowanie danych 473

Komponenty wykresów we Fleksie 474
Typy wykresów 474
Pakiety z komponentami wykresów 474
Elementy wykresu 475
Przygotowanie wykresów 476

Wypełnianie wykresów danymi 477
Określanie serii danych dla wykresu 478
Dodawanie poziomych i pionowych osi do wykresów liniowych i kolumnowych 483
Dodawanie legendy do wykresu 489
Ograniczanie liczby etykiet wyświetlanych na osi 490
Interaktywność wykresów 491
Zdarzenia związane ze wskaźnikiem myszy 492
Zdarzenia związane z klikaniem 492
Zdarzenia związane z zaznaczaniem 492
Dodawanie zdarzeń do wykresów 492
Dodawanie animacji do wykresów 496
Personalizowanie wyglądu wykresu za pomocą stylów 498
Czego się nauczyłeś? 500

Lekcja 20 Tworzenie aplikacji modułowych 503

Wprowadzenie do tworzenia modułowych aplikacji we Fleksie 503
Korzystanie z modułów Fleksa 505
Korzystanie z klasy modułu 505
Wprowadzenie znacznika ModuleLoader 507
Mechanizm RSL 508
Zadania konsolidatora 510
Przechowywanie bibliotek RSL 511
Cel przechowywania bibliotek 511
Podpisane i niepodpisane biblioteki RSL 512
Kontrolowanie bieżącego rozmiaru aplikacji 512
Dostosowanie aplikacji do korzystania z bibliotek RSL 513
Skontrolowanie zmian dokonanych przez zastosowanie bibliotek RSL 514
Tworzenie projektu biblioteki 515
Dodawanie do biblioteki klas i danych 515
Wykorzystanie biblioteki w aplikacji FlexGrocer 516
Czego się nauczyłeś? 517

Lekcja 21 Wdrażanie aplikacji Fleksa 519

Kompilowanie wdrożeniowej wersji aplikacji 520
Porzucenie okna przeglądarki i przejście do AIR 520
Rozpoczęcie pracy ze środowiskiem AIR 521
Instalowanie środowiska uruchomieniowego AIR 521
Instalowanie pierwszej aplikacji 522
Tworzenie aplikacji AIR 523
Tworzenie nowego projektu AIR 523
Przenoszenie aplikacji Dashboard do projektu AIR 525
Dostosowanie aplikacji za pomocą pliku XML 526
Eksportowanie pliku AIR 529
Czego się nauczyłeś? 533

Lekcja 22 Tworzenie przejść i zachowań 535
Wprowadzenie do zachowań i przejść 535
Wykorzystanie zachowań w komponentach 536
Wykorzystanie przejść w stanach aplikacji 538
Implementowanie efektów w komponencie 539
Dodawanie efektów do stanów aplikacji 541
Czego się nauczyłeś? 542

Lekcja 23 Drukowanie we Fleksie 546
Wprowadzenie do drukowania we Fleksie 547
Pierwszy wydruk z Fleksa 547
Korzystanie z klasy PrintDataGrid w niewidocznych kontenerach 549
Tworzenie widoku drukowania w osobnym komponencie 553
Skalowanie drukowanego dokumentu 555
Drukowanie potwierdzenia procesu zamawiania 559
Czego się nauczyłeś? 562

Lekcja 24 Stosowanie obiektów współdzielonych 565
Wprowadzenie do obiektów współdzielonych 565
Tworzenie obiektów współdzielonych 566
Odczytywanie obiektów współdzielonych 568
Tworzenie obiektu współdzielonego przechowującego zawartość koszyka 569
Odczytywanie danych z istniejącego obiektu współdzielonego 571
Czego się nauczyłeś? 574

Lekcja 25 Debugowanie aplikacji we Fleksie 577
Wprowadzenie do technik debugowania 577
Śledzenie wymiany danych między klientem a serwerem 578
Zaawansowane korzystanie z debugera 579
Więcej na temat ustawiania pułapek 579
Inspekcja zmiennych i powiązanych wartości 580
Obsługa błędów za pomocą instrukcji try-catch 586
Korzystanie z instrukcji try-catch 587
Występujące typy błędów 588
Korzystanie z wielu bloków catch 588
Przykład wykonywania tylko jednego bloku catch 589
Przykład niewłaściwego zastosowania klasy Error w pierwszym bloku catch 590
Korzystanie z instrukcji finally 590
Korzystanie z instrukcji throw 591
Tworzenie własnych klas błędów 592
Czego się nauczyłeś? 594

Lekcja 26 Profilowanie aplikacji Fleksa 597
Wykorzystanie pamięci przez Flash Player 598
Alokowanie pamięci we Flash Playerze 598
Przekazywanie przez referencję lub przez wartość 598

Mechanizm oczyszczający pamięć we Flash Playerze 599
Oczyszczanie pamięci 602
Profilowanie pamięci w aplikacjach Fleksa 604
Omówienie aplikacji ProfilerTest 605
Profilowanie aplikacji ProfilerTest 607
Poprawianie klasy ImageDisplay 611
Profilowanie wydajności aplikacji Fleksa 612
Profilowanie aplikacji ProfilerTest 613
Poprawianie klasy ProfilerTest 615
Czego się nauczyłeś? 615

Dodatek A Instalowanie oprogramowania 618

Skorowidz 621

5 Obsługa zdarzeń i struktury danych

Ważną częścią tworzenia bogatych aplikacji internetowych jest zbudowanie efektywnej architektury po stronie klienta. Za pomocą programu Flash Player do tworzenia aplikacji można użyć opartego na zdarzeniach modelu programowania, tworzyć bogate modele danych po stronie klienta i tworzyć logiczne aplikacje, postępując według dobrych praktyk programowania zorientowanego obiektowo. Ten typ projektowania różni się bardzo od metod stosowanych przez programistów aplikacji sieciowych, ponieważ nie korzysta się w nim z modelu projektowania opartego na stronie, sterowanego przepływem. Korzystanie z opartej na zdarzeniach architektury po stronie klienta skutkuje lepiej działającymi aplikacjami, w mniejszym stopniu obciążającymi sieć, ponieważ nie jest konieczne ciągłe odświeżanie strony. W tej lekcji zastosujemy techniki programowania opartego na zdarzeniach z niestandardowymi klasami ActionSriptu we Fleksie.

```
package valueObjects
{
    [Bindable]
    public class Product
    {
        public var catID:Number;
        public var prodName:String;
        public var unitID:Number;
        public var cost:Number;
        public var listPrice:Number;
        public var description:String;
        public var isOrganic:Boolean;
        public var isLowFat:Boolean;
        public var imageName:String;

        public function Product (_catID:Number, _prodName:String, _unitID:Number, _cost:
            catID = _catID;
            prodName =_prodName;
            unitID =_unitID;
            cost = _cost;
            listPrice = _listPrice;
            description = _description;
            isOrganic = _isOrganic;
            isLowFat =  isLowFat;
```

Ukończona struktura danych FlexGrocer, utworzona w języku ActionScript 3.0 i zintegrowana z aplikacją

Zrozumienie obsługi zdarzeń

Flex wykorzystuje model programowania oparty na zdarzeniach lub, inaczej, sterowany zdarzeniami. Oznacza to, że zdarzenia decydują o przebiegu działania aplikacji. Na przykład kliknięcie przez użytkownika przycisku lub dane zwracane z usługi sieciowej decydują o tym, co powinno nastąpić w aplikacji.

Zdarzenia te reprezentują dwa typy: zdarzenia użytkownika (na przykład kliknięcie myszą lub wciśnięcie klawisza) i systemowe (uruchomienie aplikacji i wyświetlenie jej, wyświetlenie niewidocznego elementu). Programista Fleksa decyduje, co powinno wydarzyć się po wystąpieniu danego zdarzenia, obsługuje te zdarzenia i pisze kod, który ma zostać wykonany.



Wiele osób zajmujących się programowaniem po stronie serwera przywykło do modelu programowania sterowanego przepływem, w którym to programista decyduje o przebiegu działania aplikacji, a nie zdarzenia systemowe i użytkownika.

Oddziaływanie użytkownika na aplikację wywołuje szereg czynności:

1. Użytkownik oddziałuje na aplikację.
2. Obiekt, na który oddziałuje użytkownik, tworzy zdarzenie.
3. Zdarzenie zostaje odczytane i uchwycone.
4. Zostaje wykonany kod powiązany z odpowiednim uchwytem zdarzenia.

Prosty przykład

Aby ułatwić zrozumienie tematu, przyjrzyjmy się konkretnemu przykładowi. Po kliknięciu przez użytkownika przycisku w etykiecie pojawia się tekst:

```
<mx:Label id="myL"/>

   <mx:Button id="myButton"
      label="Click Me"
      click="myL.text='Button Clicked'"/>
```

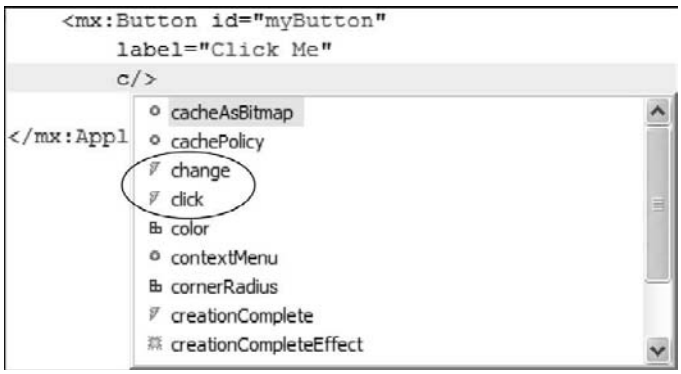
Zostaje wyświetlony przycisk z tekstem *Click Me*. Po kliknięciu przycisku przez użytkownika zostaje wysłane zdarzenie. W tym przypadku zostaje wykonany kod języka ActionScript — `myL.text=` ↪`'Button Clicked'`. Właściwości `text` etykiety zostaje przypisany łańcuch znakowy `Button Clicked`. Ten fragment kodu zawiera wartości zagnieżdżone i dlatego konieczne jest użycie zagnieżdżonych cudzysłowów pojedynczych i podwójnych. Cudzysłowy podwójne otaczają cały wiersz kodu, a pojedyncze wyodrębniają łańcuch.

Do tej pory przypisywaliśmy właściwościom wartości skalarne lub powiązania. Wartości skalarne to proste typy danych, takie jak łańcuchy znakowe, liczby lub wartości logiczne. Używaliśmy ich na przykład do określania wartości współrzędnych `x` i `y`, szerokości i wartości wyświetlanych w ety-

kietach. Stosowaliśmy także powiązania. Było tak zawsze, gdy wartość była umieszczona w nawiasach klamrowych. W ostatniej lekcji nawiasy pozwalały na wprowadzenie kodu ActionScript jako wartości właściwości.

Jeżeli właściwość zostaje przypisana do zdarzenia, jest ona rozumiana przez kompilator Fleksa jako kod języka ActionScript. Można więc wpisać kod ActionScript bezpośrednio jako wartość zdarzenia, pomijając nawiasy. Widać to we wcześniejszym przykładowym kodzie: `click="myL.text='Button Clicked'"`.

Podpowiadanie kodu pomaga nie tylko podczas wpisywania nazw właściwości, ale także nazw zdarzeń. Na poniższym rysunku widać zdarzenia `change` i `click` ze znajdującą się przed nimi ikoną błyskawicy, która oznacza zdarzenia.



Obsługa zdarzenia przez funkcję języka ActionScript

Po kliknięciu przycisku kod użyty do wpisania tekstu w etykiecie działa poprawnie. Pojawia się jednak problem, gdy trzeba wpisać więcej niż dwa wiersze kodu ActionScript, który ma zostać wykonany w celu obsługi zdarzenia. Po zdarzeniu `click` należałoby użyć wielu wierszy kodu oddzielonych średnikami. Taki zapis, chociaż prawidłowy, powoduje bałagan. Po wystąpieniu różnych zdarzeń może być wykonywany jednakowy kod. Aby zastosować to rozwiązanie, trzeba skopiować i wkleić ten kod w wielu miejscach, co w przypadku konieczności jego edycji wymaga wprowadzenia wielu zmian.

Lepszą metodą obsługiwania zdarzeń jest tworzenie funkcji w języku ActionScript. Zostanie ona umieszczona w bloku `<mx:Script>`, który informuje kompilator Fleksa, że kod w bloku `Script` jest kodem ActionScript. Zamiast więc umieszczać kod ActionScript, który ma zostać wykonany, jako wartość zdarzenia `click`, lepiej jest wywołać funkcję. Poniżej znajduje się kod wykonujący dokładnie to samo, co zaprezentowany wcześniej. Różnica polega na tym, że korzystano z zalecanej techniki umieszczania kodu, który ma zostać wykonany w odpowiedzi na zdarzenie, w funkcji.

```
<mx:Script>
   <![CDATA[
      private function clickHandler():void
      {
      myL.text="Button Clicked";
      }
   ]]>
</mx:Script>
```

```
<mx:Label id="myL"/>

<mx:Button id="myButton"
   label="Click Me"
   click="clickHandler()"/>
```



Blok `<![CDATA[]]>` wewnątrz bloku skryptu oznacza daną sekcję jako dane znakowe. Informuje on kompilator, że zamiast kodu XML w bloku są zawarte dane znakowe, co zabezpiecza przed wygenerowaniem błędów XML dla tego bloku.

Teraz, gdy zostanie kliknięty przycisk, zostaje wywołana funkcja `clickHandler()` i do etykiety zostaje wpisany łańcuch znakowy. Ponieważ nie ma w kodzie zagnieżdżonych cudzysłowów, do otoczenia tekstu mogły zostać użyte cudzysłowy podwójne.

Samej funkcji został nadany typ `void`. Oznacza to, że nie zwraca ona żadnej wartości. Bardzo dobrą praktyką jest przypisywanie funkcjom typu, nawet jeżeli jest to typ `void` wskazujący, że nie zostaje zwrócona żadna wartość. Kompilator wyświetli ostrzeżenie, jeżeli funkcji nie zostanie nadany żaden typ.

Przekazywanie danych podczas wywołania funkcji uchwytu zdarzenia

Podczas wywoływania funkcji może zaistnieć potrzeba przekazania danych. W ActionScripcie działa to tak, jak można się spodziewać. Dane, które mają zostać przekazane, są wpisywane wewnątrz nawiasów następujących po nazwie funkcji, następnie uchwyt zdarzenia musi zostać zmodyfikowany tak, by mógł przyjąć te dane. Parametry, tak jak funkcje, również powinny posiadać typ, który funkcja akceptuje.

W poniższym przykładzie aplikacja jest zmodyfikowana tak, że łańcuch znaków wyświetlany w etykiecie jest przekazywany do uchwytu zdarzenia jako parametr.

```
<mx:Script>
   <![CDATA[
      private function clickHandler(toDisplay:String):void
      {
         myL.text=toDisplay;
      }
   ]]>
</mx:Script>

<mx:Label id="myL"/>

<mx:Button id="myButton"
   label="Click Me"
   click="clickHandler('Value Passed')"/>
```

W tym przypadku po kliknięciu przycisku do funkcji uchwytu zdarzenia zostaje przekazany łańcuch `Value Passed`. Funkcja przyjmuje dane do parametru `toDisplay`, któremu został nadany typ `String`. Wartość przechowywana w zmiennej `toDisplay` zostaje następnie wyświetlona jako właściwość `text` etykiety.

Tworzenie struktury danych w zdarzeniu creationComplete

Zdarzenie `creationComplete` zostaje wysłane po utworzeniu egzemplarza elementu i jego poprawnym umieszczeniu w aplikacji. Element będzie widoczny w aplikacji po wystąpieniu zdarzenia `creation ↪Complete`, chyba że jego właściwość `visible` jest ustawiona na `false`. Zdarzenie `creationComplete` obiektu głównego `Application` zostaje wysłane po wysłaniu przez wszystkie jego elementy potomne (wszystkie elementy zawarte w kontenerze `Application`) zdarzeń `creationComplete`, które ich dotyczą.

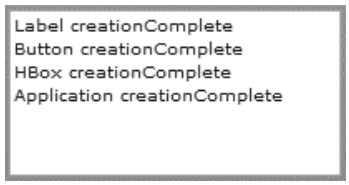
Przeanalizujmy poniższy fragment kodu:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   creationComplete="addToTextArea('Application creationComplete')">

   <mx:Script>
      <![CDATA[
         private function addToTextArea(eventText:String):void
         {
            var existingText:String=reportEvents.text;
            reportEvents.text=existingText+eventText+"\n";
         }
      ]]>
   </mx:Script>

   <mx:TextArea editable="false"
      height="100"
      width="200"
      borderStyle="solid"
      id="reportEvents" />

   <mx:HBox creationComplete="addToTextArea('HBox creationComplete')">
      <mx:Label creationComplete="addToTextArea('Label creationComplete')"/>
      <mx:Button creationComplete="addToTextArea('Button creationComplete')"/>
   </mx:HBox>

</mx:Application>
```

Spójrzmy najpierw na uchwyt zdarzenia nazwany `addToTextArea`. Przyjmuje on parametr o nazwie `eventText` i umieszcza go w obszarze tekstu, wprowadzając po nim łamanie wiersza. Dla każdego z elementów, czyli `Application`, `HBox`, `Label` i `Button`, zostaje wywołane wydarzenie `creationComplete`. Gdy proces tworzenia każdego z elementów zostaje zakończony, zdarzenie jest wysyłane i odpowiedni dla niego łańcuch znaków zostaje przekazany do uchwytu zdarzenia w celu wyświetlenia w obszarze tekstowym.

Flex nie tworzy elementów w kolejności od góry do dołu. Należy sobie wyobrażać ten proces raczej jako tworzenie od środka na zewnątrz. Zdarzenia `creationComplete` są więc wysyłane najpierw przez kontrolki `Label` i `Button`. Następnie, skoro tworzenie elementów potomnych kontenera `HBox` zostało zakończone, zdarzenie `creationComplete` zostaje wysłane przez `HBox`. W końcu, gdy wszystkie elementy potomne aplikacji zostały utworzone, sam element `Application` może wysłać zdarzenie `creationComplete`.

Wyniki wyświetlone na ekranie w obszarze tekstowym wyglądają tak, jak te zaprezentowane poniżej.



Teraz już rozumiesz, dlaczego zdarzenie `creationComplete` obiektu `Application` jest często wykorzystywane do pobierania danych. Kiedy wszystkie obiekty potomne obiektu `Application` zostaną już utworzone, nadchodzi odpowiedni moment na żądanie danych z zewnętrznego źródła i wykorzystanie ich.

W następnym ćwiczeniu wprowadzimy dwie duże zmiany: utworzymy odwołanie do pliku zewnętrznego będącego źródłem danych, które zostaną osadzone w aplikacji, oraz zastosujemy uchwyt zdarzenia do śledzenia niektórych danych w celu upewnienia się, że są one poprawnie pobierane.

Najpierw zostanie określony zewnętrzny model XML dla znacznika `<mx:Model>`. Następnie zdarzenie `creationComplete` zostanie wykorzystane do wywołania metody lub funkcji, która za pomocą własnej klasy ActionScript ostatecznie utworzy pojedyncze obiekty wartości.

1. W programie Flex Builder otwórz plik *src\assets\inventory.xml*.

   Plik ten jest zbiorem danych użytych w poprzedniej lekcji, tylko że obecnie jest plikiem zewnętrznym.

2. Otwórz plik *EComm.mxml*, z którym pracowałeś w poprzedniej lekcji. Jeżeli jej nie ukończyłeś, możesz go otworzyć z katalogu *Lekcja05\start* i zapisać go w swoim folderze *flexGrocer\src*.

3. Usuń węzły potomne ze znacznika `<mx:Model>` oraz znacznik zamykający `</mx:Model>`.

4. Dodaj na końcu znacznika `<mx:Model>` ukośnik zamykający:

   ```
   <mx:Model id="groceryInventory" />
   ```

5. Wewnątrz znacznika `<mx:Model>` wskaż zewnętrzny plik XML, określając wartość atrybutu `source` jako `"assets\inventory.xml"`:

   ```
   <mx:Model id="groceryInventory" source="assets/inventory.xml"/>
   ```

   Znacznik `<mx:Model>` automatycznie przeprowadzi analizę składniową danych z zewnętrznego pliku XML do pierwotnej postaci struktury danych ActionScriptu — w tym przypadku obiektu. W lekcji 6., „Używanie zdalnych danych XML z kontrolkami”, napiszemy więcej o bardziej złożonych strukturach danych.

6. Do znacznika `<mx:Application>` dodaj zdarzenie `creationComplete` i spraw, by wywoływało ono funkcję uchwytu zdarzenia o nazwie `prodHandler()`, tak jak poniżej:

   ```
   <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
      layout="absolute"
      creationComplete="prodHandler()">
   ```

Jak już wyjaśniliśmy, jeżeli zdarzenie `creationComplete` znajduje się w znaczniku `<mx:Application>`, zostaje ono wysłane dopiero po utworzeniu wszystkich elementów potomnych tego znacznika. Jest to korzystne, ponieważ oznacza, że wszystkie elementy aplikacji są już gotowe do użycia.

7. Prześlij do funkcji `prodHandler()` strukturę danych utworzoną w znaczniku `<mx:Model>` — `groceryInventory`:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute"
   creationComplete="prodHandler(groceryInventory)">
```

Identyfikator znacznika `<mx:Model>` jest równy `groceryInventory`. Znacznik ten utworzył automatycznie z pliku XML prosty obiekt języka ActionScript i może zostać teraz wykorzystany do wiązania danych.



Jeżeli została utworzona zmienna lub funkcja, podczas wpisywania znacznika dostępna jest pomoc. W tym przypadku, wprowadzając nazwę zmiennej `groceryInventory` *jako parametr funkcji, można wpisać* `gr`*, następnie wcisnąć* Ctrl+*spacja i wybrać nazwę zmiennej z listy.*

8. Zaraz po istniejącym znaczniku `<mx:Model>` dodaj blok `<mx:Script>`. Zauważ, że Flex Builder automatycznie dodaje znacznik `CDATA`. U góry bloku skryptu zdefiniuj funkcję prywatną typu `void` o nazwie `prodHandler()`. Musi ona przyjmować parametr typu `Object` o nazwie `theItems`.

```
<mx:Script>
      <![CDATA[
         private function prodHandler(theItems:Object):void
         {
         }
      ]]>
   </mx:/Script>
```

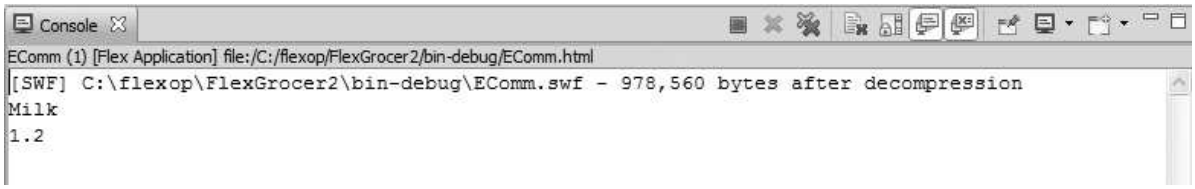
Funkcja `prodHandler()` jest uchwytem zdarzenia `creationComplete`. Została określona jako prywatna, co oznacza, że może być używana tylko wewnątrz klasy. Funkcja będzie przyjmowała pojedynczy parametr typu `Object` oraz nie będzie zwracała żadnej wartości, ponieważ jako typ zwracany został określony `void`.

9. Wewnątrz funkcji `prodHandler()` dodaj dwa wyrażenia `trace`, wyświetlające nazwę i koszt elementów przekazywanych do funkcji.

```
private function(prodHandler(theItems:Object):void
{
   trace (theItems.prodName);
   trace(theItems.cost);
}
```

10. Kliknij narzędzie *Debug*, aby skompilować aplikację. Wróć do Flex Buildera.

W widoku *Console* zostaną wyświetlone wyniki działania wyrażeń `trace`. W oknie konsoli powinna być widoczna nazwa produktu i jego koszt. Będziesz musiał zminimalizować wyświetlone okno przeglądarki oraz wybrać z menu opcję *Window/Console*, jeżeli widok *Console* nie jest wyświetlany. (Zobacz rysunek na następnej stronie).



Używanie danych z obiektu Event

Flex tworzy obiekt `Event` za każdym razem, gdy zostaje wysłane zdarzenie. Obiekt ten zawiera informacje o zdarzeniu, które nastąpiło. Jest tworzony automatycznie nawet wtedy, gdy nie zostanie wykorzystany. Będzie jednak często stosowany. Zwykle obiekt zdarzenia jest wysyłany do uchwytu zdarzenia i odczytywane są z niego właściwości.

Istnieje bardzo wiele rodzajów obiektów zdarzeń, których można użyć. Ogólny obiekt `Event` jest zdefiniowany w klasie języka ActionScript. Ta ogólna wersja nie będzie jednak działała we wszystkich zdarzeniach. Można się domyślić, że najważniejsza informacja zwrócona ze zdarzenia „przeciągnięcia i upuszczenia" będzie różna od najważniejszej informacji zwróconej po zakończeniu wywołania usługi sieciowej. Z tego powodu ogólna klasa `Event` stanowi podstawę dla wielu innych klas zdarzeń. Na poniższym rysunku widać, jak wiele innych klas jest opartych na ogólnej klasie `Event` lub stanowi jej podklasy.

Package	flash.events
Class	public class Event
Inheritance	Event → Object
Subclasses	ActivityEvent, ADGHeaderShiftEvent, ADGItemSelectEvent, AdvancedDataGridEvent, AIREvent, AutomationEvent, AutomationRecordEvent, AutomationReplayEvent, BrowserChangeEvent, CalendarLayoutChangeEvent, ChannelEvent, ChartSelectionChangeEvent, ChildExistenceChangedEvent, CloseEvent, CollectionEvent, ColorPickerEvent, ContextMenuEvent, CubeEvent, CuePointEvent, DataGridEvent, DateChooserEvent, DividerEvent, DRMAuthenticateEvent, DRMStatusEvent, DropdownEvent, DynamicEvent, EffectEvent, FileEvent, FileListEvent, FlexEvent, FocusEvent, HTMLUncaughtJavaScriptExceptionEvent, HTTPStatusEvent, IndexChangedEvent, InvokeEvent, ItemClickEvent, KeyboardEvent, ListEvent, ListItemSelectEvent, LogEvent, MenuShowEvent, MessageEvent, MessageFaultEvent, MetadataEvent, MouseEvent, MoveEvent, NativeWindowBoundsEvent, NativeWindowDisplayStateEvent, NetStatusEvent, NumericStepperEvent, OutputProgressEvent, ProgressEvent, PropertyChangeEvent, ResizeEvent, ScrollEvent, SliderEvent, SQLEvent, SQLUpdateEvent, StateChangeEvent, StatusEvent, SyncEvent, TextEvent, TextSelectionEvent, TimerEvent, ToolTipEvent, TreeEvent, TweenEvent, ValidationResultEvent, VideoEvent

Czy wszystkie te obiekty zdarzeń są potrzebne? Potrzebne są te zawierające wszystkie potrzebne informacje, ale nie ich nadmiar.

Ogólny obiekt `Event` zawiera właściwości użyte we wszystkich obiektach typu zdarzenie. Dwa łatwe do zrozumienia typy to `type` i `target`. Właściwość `type` zdarzenia jest łańcuchem zawierającym nazwę uchwyconego zdarzenia, na przykład `click` lub `creationComplete`. Właściwość `target` wskazuje element, który wysłał zdarzenie, na przykład element `Button`, jeżeli zdarzenie zostało wysłane przed kliknięciem przycisku.



`Target` (cel) może wydawać się dziwnym określeniem tej właściwości, skoro jest to zdarzenie, które wysłało inne zdarzenie, a nie cel czegokolwiek. Wyjaśni się to w lekcji 9., „Stosowanie zdarzeń użytkownika", i gdy dowiesz się czegoś o przepływie zdarzeń.

Przyjrzyjmy się poniższemu kodowi, który wysyła obiekt zdarzenia, w tym przypadku obiekt `MouseEvent`, do uchwytu zdarzenia.

```
<mx:Script>
   <![CDATA[
      private function clickHandler(event:MouseEvent):void
      {
         trace(event.type);
      }
   ]]>
</mx/Script>

<mx:Label id="myL"/>

<mx:Button id="myButton"
   label="Click Me"
   click="clickHandler(event)"/>
```

W powyższym kodzie zdarzenie zostaje wysłane do uchwytu zdarzenia i podczas debugowania aplikacji w widoku *Console* zostaje wyświetlone słowo `click`.

Chociaż obiekt `Event` nie jest używany w żadnym miejscu tworzonych obecnie aplikacji, zrozumienie, do czego służy, jest niezbędne. W związku z tym, chociaż obiekt ten nie jest zawarty w żadnej z trzech aplikacji tworzonych podczas pracy z książką, w poniższym ćwiczeniu zastosujemy go.

1. Wybierz z menu *File/New/Flex Project*. Nadaj projektowi nazwę *EventObject*.
2. Ustal lokalizację pliku projektu jako *flexop\Lekcja05\eventObject\start*.
3. Jako typ aplikacji wybierz *Web application*.
4. Ustaw technologię serwera jako `none`, a następnie kliknij *Next*.
5. Pozostaw typ folderu wyjściowego `bin-debug` i kliknij *Next*.
6. Pozostaw *src* jako główny folder źródłowy.
7. Przeszukaj opcję *Main application file* i zaznacz *EventTest.mxml*. Kliknij *Finish*.

   Właśnie utworzyłeś projekt. Możesz uruchomić aplikację, która będzie wykorzystywana do eksperymentowania z obiektem zdarzenia. Plik ten jest tylko szkieletem aplikacji z dodanym blokiem `Script`.
8. Dodaj przycisk pomiędzy zamykającym znacznikiem `</mx:Script>` a końcem aplikacji. Nadaj przyciskowi za pomocą właściwości `id` nazwę egzemplarza `myButton`. Dodaj właściwość `label` o wartości **`Click To Debug`**. Dodaj zdarzenie `click` i spowoduj, by wywoływało ono uchwyt zdarzenia o nazwie `doClick()`, przesyłający obiekt zdarzenia w formie parametru:
   `<mx:Button id="myButton" label="Click To Debug" click="doClick(event)"/>`

   Kiedy zostanie kliknięty przycisk, zostanie wywołana funkcja `doClick()` i zostanie do niej wysłany obiekt `event`.
9. W bloku skryptu dodaj prywatną funkcję typu `void` o nazwie `doClick()`. Przyjmij parametr typu `MouseEvent` o nazwie `event`, tak jak poniżej:

```
private function doClick(event:MouseEvent):void
{

}
```



Można przyjąć spójne nazewnictwo dla uchwytów zdarzeń. Na przykład w tej lekcji używane jest zdarzenie `click` obsługiwane przez uchwyty zdarzeń o nazwach `clickHandler()` i `doClick()`. Nie istnieje jedyna słuszna metoda nadawania nazw uchwytom zdarzeń, można jednak wybrać sobie konwencję nazewnictwa i trzymać się jej.

W następnym kroku użyjemy wbudowanego w program Flex Builder debugera w najprostszy sposób. Debuger zostanie dokładnie omówiony w lekcji 25., „Debugowanie aplikacji we Fleksie". Tutaj użyjemy go do przyjrzenia się obiektowi zdarzenia.

10. Dodaj punkt przerwania programu na zamykającym funkcję nawiasie klamrowym, dwukrotnie klikając pasek znaczników po lewej stronie kodu i numerów wierszy. Na pasku pojawi się mały niebieski punkt wskazujący miejsce, w którym zostanie wstrzymane wykonanie programu. W tym punkcie można przyjrzeć się aktualnym wartościom.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
3
4      <mx:Script>
5          <![CDATA[
6              private function doClick(event:MouseEvent):void
7              {
8
9  Line breakpoint: EventTest.mxml [line: 9]
10         ]]>
11     </mx:Script>
12
13     <mx:Button id="myButton"  click="doClick(event)"/>
14
15 </mx:Application>
16
```

Debuger jest niesamowicie pomocny w zrozumieniu tego, co dzieje się w aplikacji Fleksa, dlatego warto często go używać.

11. Kliknij przycisk *Debug* interfejsu programu Flex Builder.

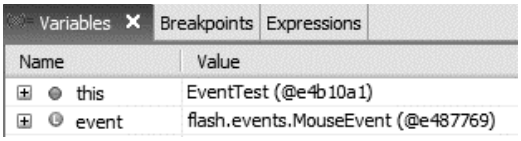


12. W przeglądarce kliknij przycisk z etykietą `Click To Debug`. We Flex Builderze zostaniesz poproszony o skorzystanie z perspektywy *Debugging*. Powinieneś to zrobić.

13. Kliknij dwukrotnie belkę widoku *Variables*, aby wyświetlić ją w trybie pełnoekranowym.

    Okno zawiera wiele informacji i przejście do widoku pełnoekranowego ułatwi ich przeglądanie.

14. Zostaną wyświetlone dwie zmienne: `this` i `event` (tak jak poniżej).



Zmienna `this` reprezentuje całą aplikację. Jeżeli klikniesz znajdujący się przed nią znak +, zobaczysz wiele właściwości i skojarzonych wartości. Zmienna `event` reprezentuje zmienną zdarzenia lokalną dla funkcji, w której został wstawiony punkt przerwania. Litera `L` umieszczona w ikonie po lewej stronie zmiennej `event` wskazuje, że jest to zmienna lokalna.

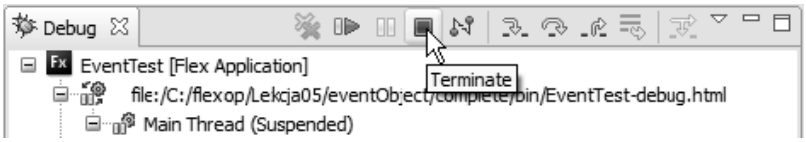
15. Kliknij znak + przed zmienną `event`, a następnie znak + przed zestawem właściwości `[inherited]`. Znajdź właściwość `target`. Zwróć uwagę, że `target` jest elementem wysyłającym zdarzenie `Button`. Zauważ także, że właściwość `type` ma wartość `click`.

    Wszystkie te wartości zostały już omówione.

16. Kliknij znak + przed `target`, następnie kliknij znak + przed zestawem właściwości `[inherited]` i znajdź właściwość `id`.

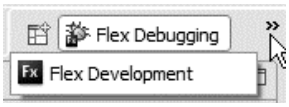
    Zauważ, że wartością właściwości jest `myButton`, tak jak zostało to wpisane w kodzie.

17. Przywróć widok *Variables*, klikając dwukrotnie jego belkę. Kliknij czerwony kwadracik w widoku *Debug* lub *Console*, by zakończyć debugowanie.



Nie zapominaj o tym, aby zakończyć sesję debugowania. Możliwe jest działanie jednej sesji debugowania w innej. Może być to przydatne w pewnych przypadkach, nie jest jednak zalecane.

18. Wróć do perspektywy *Development*, klikając znak **>>** w prawym górnym rogu ekranu i wybierając *Flex Development*.





Jeżeli umieścisz kursor z lewej strony ikony Open Perspective*, pojawi się podwójna strzałka. Możesz kliknąć i przeciągnąć okno w lewo w celu zwiększenia ilości miejsca przydzielonego perspektywom. Będzie widoczna zarówno perspektywa* Development*, jak i* Debugging*, i będzie można łatwo przełączać się między nimi, klikając ich zakładki.*



19. Zamknij projekt *EventObject* i wróć do projektu *FlexGrocer*.

Tworzenie własnej klasy języka ActionScript

Jak wspomniano pod koniec lekcji 2., książka ta nie jest podręcznikiem programowania zorientowanego obiektowo, każdy programista Fleksa musi jednak chociaż w podstawowym stopniu znać specjalistyczną terminologię. Jeżeli więc nie są Ci znane takie wyrażenia, jak *klasa*, *obiekt*, *właściwość* i *metoda*, nadszedł najwyższy czas, by skorzystać z setek, jeżeli nie tysięcy, publikacji wprowadzających do programowania zorientowanego obiektowo.

Tworzyliśmy już własne klasy języka ActionScript podczas pracy z tą książką, choć mogłeś o tym nie wiedzieć, ponieważ Flex ukrywa je przed programistą. Podczas tworzenia aplikacji w języku MXML tworzymy tak naprawdę nową klasę języka ActionScript. Kod MXML jest połączony z kodem ActionScript w bloku skryptu i jest tworzona czysta klasa języka ActionScript skompilowana do pliku *.swf*, gotowego do użycia przez program Flash Player. W ostatnim ćwiczeniu po utworzeniu *EventTest.mxml* powstał więc plik *EventTest-generated.as* zawierający poniższy kod:

```
public class EventTest extends mx.core.Application
```

W wyniku utworzenia *EventTest.mxml* została rozszerzona klasa `Application`, tak jak w wyniku utworzenia każdej z aplikacji w tej książce.



Aby widzieć tworzony kod języka ActionScript, należy dodać argument kompilatora we Flex Builderze. Przejdź do Project/Properties/Flex Compiler/Additional compiler arguments *(projekt/właściwości/kompilator Fleksa/dodatkowe argumenty kompilatora) i dodaj na końcu istniejących argumentów tekst* **`-keep-generated-actionscript`**. *W projekcie zostanie automatycznie utworzony katalog* generated*, a w nim — wiele plików języka ActionScript. Aktualne pliki aplikacji znajdują się w formularzu* Name-generated.as. *Nie zapomnij o usunięciu argumentu kompilatora po zakończeniu sprawdzania.*

W tej części lekcji zbudujemy klasy języka ActionScript bezpośrednio w tym języku, nie korzystając z konwersji kodu MXML na kod ActionScript przez Fleksa.

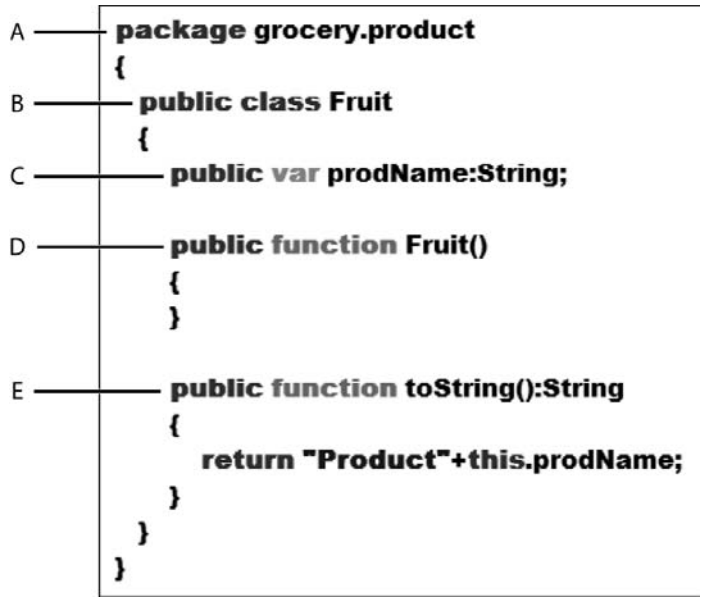
Po co się to robi? Tworzenie klas języka ActionScript jest niezbędne do wykonania niektórych ćwiczeń przedstawionych w tej książce, wliczając w to tworzenie własnych klas zdarzeń oraz obiektów wartości. Niedawno poznaliśmy obiekty zdarzeń. W lekcji 9. utworzymy obiekty zdarzeń do przechowywania określonych danych i tworzenie własnych klas języka ActionScript będzie wówczas konieczne.

Tworzenie obiektu wartości

Obiekty wartości (*value objects*, nazywane również *obiektami transferu danych* — DTO), lub po prostu *obiekty transferu*, przechowują dowolne dane związane z obiektem, nie zawierają szczegółowego kodu i logiki biznesowej oraz są wdrożone jako klasy ActionScript.

Nazwa obiektów transferu jest związana z tym, że często służą one do przesyłania danych do zaplecza aplikacji w celu zachowania ich na stałe w bazie danych. W tej lekcji utworzymy obiekty wartości dla produktów spożywczych, produktu przeniesionego do koszyka na zakupy oraz samego koszyka.

Najpierw jednak omówimy podstawy tworzenia klasy języka ActionScript. Poniżej została pokazana bardzo prosta klasa:



W linii A wyrażenie `package` reprezentuje strukturę katalogu, w którym jest przechowywana klasa. Plik jest przechowywany w katalogu *grocery/product* w strukturze plików aplikacji.

W linii B klasie zostaje nadana nazwa `Fruit`. Musi to odpowiadać nazwie pliku — *Fruit.as*.

W linii C zostały zadeklarowane własności klasy. Może ich być wiele.

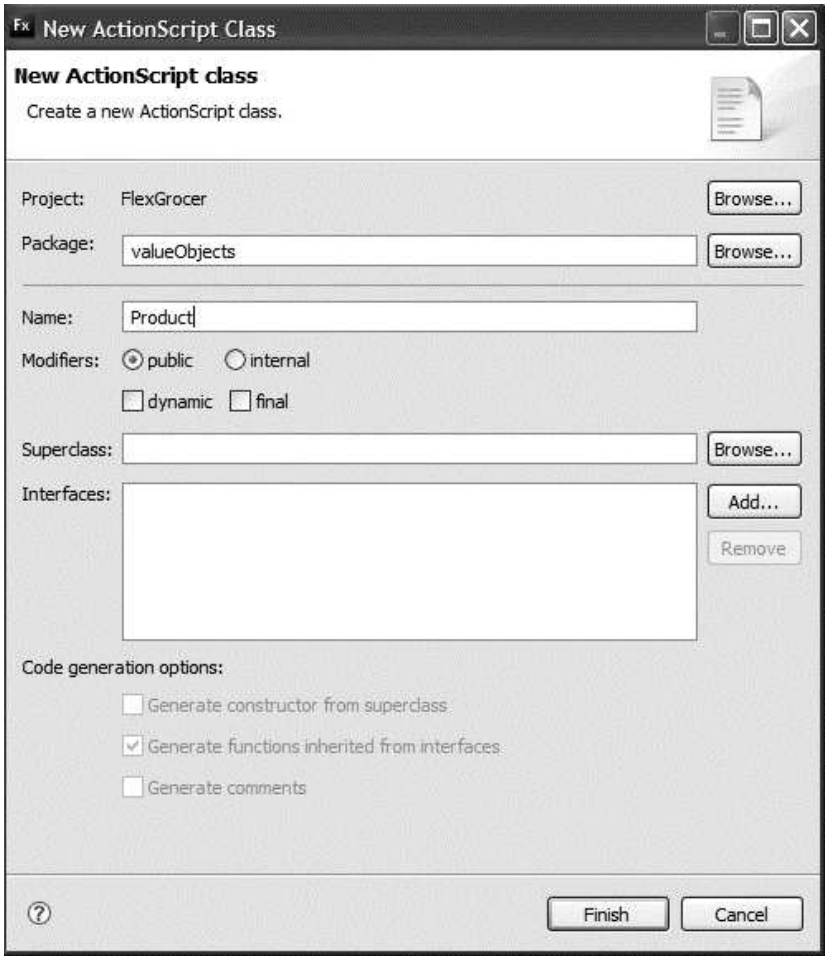
Linia D zawiera konstruktor klasy. Jest on wywoływany automatycznie, gdy tworzony jest nowy egzemplarz obiektu klasy. Nazwa funkcji konstruktora klasy musi być taka sama jak nazwa klasy, która z kolei jest taka sama jak nazwa pliku. Funkcja ta musi być funkcją publiczną i nie może mieć nadanego typu.

W linii E zostały zdefiniowane metody klasy. Może zostać zadeklarowanych wiele metod.

W aplikacji FlexGrocer trzeba zarządzać wielkimi ilościami danych i przesyłać je do innych aplikacji. W kolejnym ćwiczeniu utworzymy obiekt wartości przechowujący informacje o produkcie spożywczym.

1. Utwórz nowy plik klasy języka ActionScript, wybierając z menu *File/New/ActionScript class* (plik/nowy/klasa ActionScript). W polu *Package* wpisz **`valueObjects`**, co spowoduje automatyczne utworzenie przez Flex Builder katalogu o tej samej nazwie. Nazwij klasę **`Product`** i pozostaw wartości domyślne wszystkich pozostałych pól. Kliknij *Finish*, aby utworzyć plik.

   Zostaje utworzony plik o nazwie *Product.as*, zawierający podstawową strukturę klasy języka ActionScript. Wyrażenia `package` i `class` są słowami kluczowymi użytymi w definicji tej klasy. Należy pamiętać, że będzie ona podstawą dla wielu obiektów używanych później do opisywania każdego z produktów spożywczych. (Zobacz rysunek na następnej stronie).



2. W utworzonym pliku, przed słowem kluczowym `class`, ale za słowem kluczowym `package`, dodaj znacznik metadanych `[Bindable]`. Wewnątrz definicji klasy `Product` dodaj właściwość publiczną typu `Number` o nazwie `catID`.

```
package valueObjects{
   [Bindable]
   public class Product{
      public var catID:Number;
   }
}
```

Jeżeli znacznik metadanych `[Bindable]` zostanie wstawiony przed słowem kluczowym `class`, oznacza, że każda właściwość klasy może zostać użyta w powiązaniu (zostać powiązana z kontrolką lub inną strukturą danych). Zamiast określania całej klasy jako `[Bindable]` można określić pojedyncze właściwości. W przypadku aplikacji *FlexGrocer* każda właściwość powinna być możliwa do powiązania.

3. Utwórz właściwości publiczne o nazwach `prodName` (typ `String`), `unitID` (typ `Number`), `cost` (typ `Number`), `listPrice` (typ `Number`), `description` (typ `String`), `isOrganic` (typ `Boolean`), `isLowFat` (typ `Boolean`) oraz `imageName` (typ `String`). Klasa powinna wyglądać tak:

```
package valueObjects{
   [Bindable]
   public class Product{
      public var catID:Number;
```

```
        public var prodName:String;
        public var unitID:Number;
        public var cost:Number;
        public var listPrice:Number;
        public var description:String;
        public var isOrganic:Boolean;
        public var isLowFat:Boolean;
        public var imageName:String;
    }
}
```

Tworzymy strukturę danych przechowującą informacje magazynowe dla sklepu spożywczego. Powyżej utworzyliśmy wszystkie właściwości, które zostaną wykorzystane w tej klasie.

4. Wewnątrz nawiasów klasy `Product`, za właściwością `imageName`, zdefiniuj funkcję konstruktora klasy i określ parametry, które będą przekazywane do tej funkcji. Muszą się one zgadzać typem danych z typem już zdefiniowanych właściwości. Nazwy powinny być jednakowe, należy je jednak poprzedzić podkreślnikiem, aby uniknąć kolizji nazw parametrów i właściwości (występującej, gdy taka sama nazwa odnosi się do dwóch różnych zmiennych). Upewnij się, że nazwa funkcji odpowiada nazwie klasy oraz że konstruktor jest publiczny. Nie można nadać typu funkcji konstruktora.

```
public function Product(_catID:Number, _prodName:String, _unitID:Number,
↪_cost:Number, _listPrice:Number, _description:String, _isOrganic:Boolean,
↪_isLowFat:Boolean, _imageName:String){
}
```

Funkcja konstruktora jest wywoływana za każdym razem, gdy z klasy jest tworzony obiekt. Można utworzyć z klasy obiekt, używając słowa kluczowego `new` i przekazując klasie parametry. Na koniec przekażemy wartości ze znacznika `<mx:Model>` lub bazy danych do parametru konstruktora.

5. Wewnątrz funkcji konstruktora nadaj każdej właściwości wartość przesłaną do funkcji konstruktora.

```
public function Product(_catID:Number, _prodName:String, _unitID:Number,
↪_cost:Number, _listPrice:Number, _description:String, _isOrganic:Boolean,
↪_isLowFat:Boolean, _imageName:String)
{
    catID =_catID;
    prodName = _prodName;
    unitID = _unitID;
    cost = _cost;
    listPrice = _listPrice;
    description = _description;
    isOrganic = _isOrganic;
    isLowFat = _isLowFat;
    imageName = _imageName;
}
```

Powyższy kod ustawi wartość każdej właściwości klasy na równą odpowiedniemu parametrowi przesłanemu do konstruktora.



Każda właściwość wymieniona po lewej stronie znaku równości może mieć przedrostek `this.`, na przykład `this.catID = _catID;`. Taki przedrostek jest czasem dodawany przez programistów, którzy chcą używać takich samych nazw właściwości oraz parametrów i mimo to unikać kolizji nazw. Przedrostek `this` odwołuje się do samej klasy.

6. Utwórz bezpośrednio pod funkcją konstruktora nową metodę o nazwie `toString()`, zwracającą łańcuch `[Product]` i nazwę produktu.

```
public function toString():String
{
   return "[Product]"+this.prodName;
}
```

Metoda ta będzie zwracała nazwę aktualnego produktu i będzie przydatna przy pobieraniu nazwy. Tworzenie metod dających dostęp do właściwości jest dobrym zwyczajem, ponieważ jeżeli nazwa właściwości się zmieni, wciąż będzie można wywołać tę samą funkcję, wykorzystując kod dziedziczony. Metoda `toString()` jest wywoływana automatycznie przez szkielet Fleksa za każdym razem, gdy obiekt jest śledzony. Jest to bardzo przydatne podczas debugowania i wyświetlania struktur danych.

7. Wróć do pliku *EComm.mxml* i znajdź blok skryptu u góry strony. Poniżej znacznika `CDATA` zaimportuj klasę `Product` z katalogu *valueObjects*.

```
import valueObjects.Product;
```

W celu użycia klasy niestandardowej Flex Builder musi skorzystać z wyrażenia `import` odwołującego się do położenia lub pakietu, w którym klasa jest umieszczona. Kiedy tworzona jest własna klasa, należy ją zaimportować, aby była dostępna dla szkieletu Fleksa. Wykonuje się to za pomocą słowa kluczowego `import`.

8. W bloku skryptu następującym po wyrażeniu `import`, powyżej funkcji, zadeklaruj zmienną prywatną typu `Product` o nazwie `theProduct`. Dodaj znacznik metadanych `[Bindable]`.

```
[Bindable]
private var theProduct:Product;
```

Wszystkie pliki MXML są ostatecznie kompilowane do klasy języka ActionScript. Należy podczas tworzenia pliku MXML stosować tę samą konwencję, co podczas tworzenia klasy języka ActionScript. Należy na przykład zaimportować wszystkie klasy, które nie są macierzystymi dla języka ActionScript, tak jak utworzona klasa `Product`, i zadeklarować wszystkie właściwości, które zostaną użyte w pliku MXML. Zostaje to wykonane w bloku skryptu. Znacznik metadanych `[Bindable]` gwarantuje, że wartość tej właściwości może zostać użyta w wyrażeniach powiązań.

9. Wewnątrz funkcji `prodHandler()`, ale powyżej wyrażeń `trace`, utwórz nowy egzemplarz klasy `Product` o nazwie `theProduct`. Wypełnij i prześlij parametr do funkcji konstruktora z informacją ze znacznika `<mx:Model>`:

```
theProduct = new Product(theItems.catID, theItems.prodName, theItems.unitID,
➥theItems.cost, theItems.listPrice, theItems.description, theItems.isOrganic,
➥theItems.isLowfat, theItems.imageName);
```

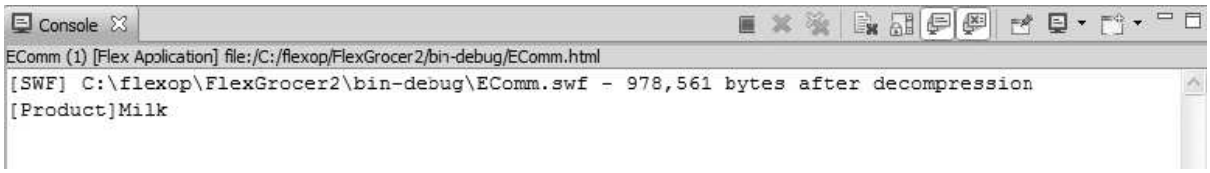
Używasz nowo utworzonej klasy i tworzysz egzemplarz jej obiektu. Dane ze znacznika `<mx:Model>` zostają przekazane jako wartości dla właściwości.

10. Usuń z funkcji `prodHandler()` dwa wyrażenia `trace` i zamień je na nowe wyrażenia `trace`. Będą one pobierały automatycznie informacje określone w metodzie `toString()`.

```
trace(theProduct);
```

11. Zapisz i włącz debugowanie aplikacji.

    W widoku konsoli powinien zostać wyświetlony tekst`[Product]Milk` wskazujący, że utworzyłeś obiekt wartości `Product`.



Tworzenie metody kreującej obiekt

Tak jak zrobiliśmy to w poprzednim ćwiczeniu, można utworzyć egzemplarz klasy `Product`, przekazując wszystkie wartości do metody konstruktora jako parametry. W tym zadaniu utworzymy metodę, która przyjmuje obiekt zawierający wszystkie pary właściwość – wartość i zwraca egzemplarz klasy `Product`. Trzeba zauważyć, że dla poprawnego działania tej metody obiekt przekazywany do niej musi zawierać nazwy właściwości dokładnie odpowiadające tym użytym w klasie.

1. Upewnij się, że klasa `Product` w katalogu *valueObjects* jest otwarta. Znajdź metodę `toString()` i dodaj za nią szkielet nowej metody o właściwościach `public` i `static`, nazwanej `buildProduct()`. Upewnij się, że zwracane przez nią dane są typu `Product` oraz że przyjmuje ona parametr `o` o typie danych `Object`, tak jak poniżej:

```
public static function buildProduct(o:Object):Product
{
}
```

   Metoda statyczna może być wywoływana bez potrzeby wcześniejszego tworzenia obiektu z klasy. Metody egzemplarzowe, tak jak te, których używaliśmy dotąd, mogą być stosowane tylko z obiektami, których egzemplarze są tworzone z klasy. Metody zadeklarowane jako statyczne mogą być używane bezpośrednio z klasy. Metody statyczne są przydatne w programach użytkowych, na przykład tworzona tutaj statyczna metoda `buildObject()`, do której chcemy mieć dostęp bez potrzeby tworzenia wcześniej obiektu. Odpowiednio użyte metody statyczne mogą podnieść wydajność, ponieważ nie muszą tworzyć najpierw obiektu zajmującego pamięć komputera. W celu odwołania się do statycznej metody `getName()` z klasy `Product` powinniśmy zastosować kod `Product.getName()`, który korzysta z nazwy klasy wstawionej przed metodą, a nie z nazwy obiektu, którego egzemplarz utworzono z klasy.

2. Wewnątrz metody `buildProduct()` utwórz egzemplarz klasy `Product` o nazwie `p`, używając słowa kluczowego `new`. Ustaw właściwości `catID`, `prodName`, `unitID`, `cost`, `listPrice`, `description`, `isOrganic`, `isLowFat` i `imageName` jako parametry konstruktora. Będziesz musiał rzutować zmienne `isOrganic` i `isLowFat` do typu boolowskiego.

```
var p:Product = new Product(o.catID, o.prodName, o.unitID, o.cost, o.listPrice,
↪o.description, Boolean(o.isOrganic), Boolean(o.isLowFat), o.imageName);
```

Pamiętaj, że użyte tutaj dane są pobierane ze znacznika `<mx:Model>`. Kiedy są pobierane w ten sposób, nie posiadają typu, więc wartości `true` i `false` są traktowane jak zwykłe łańcuchy znakowe. Rzutowanie zmiennej dostarcza kompilatorowi informacji, że ma traktować wartość jako daną określonego typu. W tym przykładzie informujesz kompilator, że Ty jako programista wiesz, że właściwości `isOrganic` i `isLowFat` będą zawierały dane logiczne i przez ich rzutowanie do tego typu nowo utworzony obiekt będzie posiadał wartości logiczne dla właściwości `isOrganic` i `isLowFat`.

3. Wróć do właśnie utworzonego obiektu, używając słowa kluczowego `return` z nazwą obiektu, czyli `p`. Gotowa metoda `buildProduct()` powinna mieć postać:

```
public static function buildProduct(o:Object):Product{
   var p:Product = new Product(o.catID, o.prodName, o.unitID, o.cost,
   ↪o.listPrice, o.description, Boolean(o.isOrganic), Boolean(o.isLowFat),
   ↪o.imageName);
   return p;
}
```

Powyższy kod zwróci obiekt wartości `Product` utworzony przez przekazanie ogólnego obiektu do metody.

4. Zapisz plik *Product.as*.

Plik klasy został zapisany z nową metodą. W widoku *Problems* nie powinny pojawić się żadne komunikaty o błędach.

5. Wróć do aplikacji *EComm.mxml*. Z metody `prodHandler()` usuń kod tworzący `theProduct` i wstaw zamiast niego kod używający metody statycznej do utworzenia `theProduct`. Pamiętaj o usunięciu słowa kluczowego `new`.

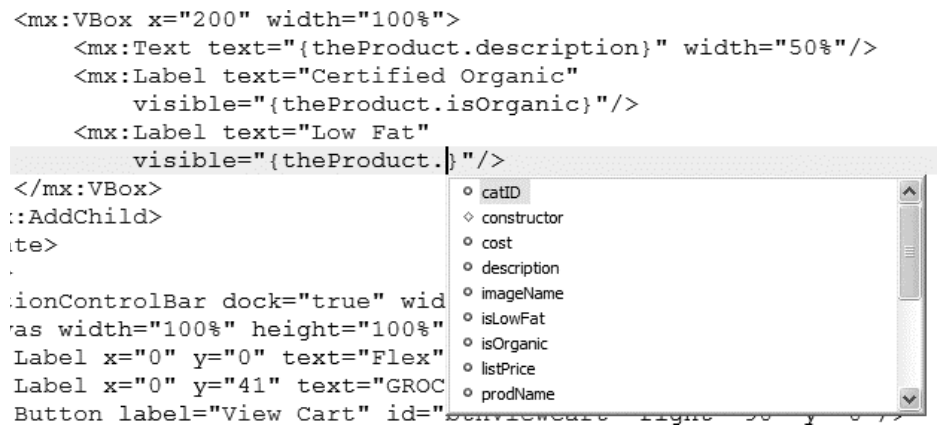
```
theProduct = Product.buildProduct(theItems);
```

Powyższy kod wywołuje statyczną metodę tworzącą egzemplarz klasy `Product`, która zwraca obiekt wartości `Product` o nadanym mocnym typie z obiektu, który nie posiadał nadanego typu.

6. Znajdź w stanie rozszerzonym pierwszy kontener układu `VBox`, który wyświetla opis produktu oraz informacje, czy jest on ekologiczny i dietetyczny. Zmień właściwość `text` znacznika `<mx:Text>` tak, by odwoływał się do obiektu `theProduct` utworzonego metodą `prodHandler()`. Dodaj także właściwość `visible` do obu etykiet oraz powiąż każdą z nich z odpowiednią właściwością obiektu `theProduct`, tak jak jest to pokazane w następnym fragmencie kodu. (Zobacz rysunek na następnej stronie).



Pamiętaj, że teraz `Product` jest klasą zaimportowaną, dostępne jest zatem podpowiadanie kodu zarówno dla nazwy klasy, jak i jej właściwości. Kiedy kursor znajduje się wewnątrz nawiasów tworzących powiązanie, naciśnij Ctrl+*spacja, aby uzyskać pomoc we wpisywaniu egzemplarza klasy `Product` — `theProduct`. Następnie, po wstawieniu kropki, zostaną wyświetlone właściwości.*




```
<mx:VBox x="200" width="100%">
   <mx:Text text="{theProduct.description}"
      width="50%"/>
   <mx:Label
      text="Certified Organic"
      visible="{theProduct.isOrganic}"/>
   <mx:Label
      text="LowFat"
      visible="{theProduct.isLowFat}"/>
</mx:VBox>
```

Odwołujesz się teraz do obiektu wartości, który utworzyliśmy wcześniej.

7. Zapisz plik i uruchom debugowanie.

   Powinno być widoczne, że `trace` działa tak jak wcześniej, a powiązanie danych powinno działać po umieszczeniu kursora na obrazku.

Tworzenie klas koszyka na zakupy

W tym ćwiczeniu utworzymy nową klasę dla produktów dodawanych do koszyka na zakupy. Jej zadaniem będzie śledzenie, jaki produkt został dodany oraz w jakiej ilości. Utworzymy metodę, która będzie liczyła koszt zakupu tego produktu. Zbudujemy także szkielet klasy `ShoppingCart`, która będzie obsługiwała cały układ logiczny koszyka na zakupy, wliczając w to dodawanie do niego elementów.

1. Utwórz nowy plik klasy języka ActionScript, wybierając z menu *File/New/ActionScript class*. Ustaw wartość *Package* równą `valueObjects`, dodając automatycznie tę klasę do utworzonego wcześniej katalogu. Wprowadź **`ShoppingCartItem`** w polu nazwy i pozostaw wartości domyślne dla wszystkich pozostałych pól.

   W tej klasie będzie sprawdzana liczba produktów oraz ich cena.

2. Wewnątrz definicji `class` wstaw właściwość publiczną o nazwie `product` i typie danych `Product`:

```
package valueObjects {
   public class ShoppingCartItem {
      public var product:Product;
   }
}
```

Ważnym elementem koszyka na zakupy jest informacja, który produkt został do niego dodany. Utworzyliśmy już klasę `Product` służącą śledzeniu tych danych, jest więc jak najbardziej logiczne wykorzystanie egzemplarza tej klasy w klasie `ShoppingCartItem`.

3. Zdefiniuj właściwość publiczną o nazwie `quantity` i typie danych `uint`:

```
package valueObjects {
   public class ShoppingCartItem {
      public var product:Product;
      public var quantity:uint;
   }
}
```

Typ danych `uint` oznacza całkowitą liczbę nieujemną (0, 1, 2, 3, …). Ilość dodanego do koszyka produktu może wynosić zero lub zostać określona liczbą dodatnią, zastosowanie typu `uint` jest więc logiczne.

4. Zdefiniuj właściwość publiczną typu `Number` o nazwie `subtotal`:

```
package valueObjects {
   public class ShoppingCartItem {
      public var product:Product;
      public var quantity:uint;
      public var subtotal:Number;
   }
}
```

Za każdym razem, gdy użytkownik doda produkt do koszyka, będzie potrzebna aktualizacja sumy wartości produktów. Ostatecznie dane te zostaną wyświetlone w kontrolce.

5. Zaraz po właściwości `subtotal` zdefiniuj nagłówek funkcji konstruktora klasy i określ parametry przekazywane do tej funkcji. Zaliczają się do nich: `product`, parametry typu `Product` i `quantity` oraz parametry typu `uint`. Ponieważ konstruktor jest wywoływany tylko podczas tworzenia produktu, ustaw wartość parametru `quantity` równą `1`.

```
public function ShoppingCartItem(product:Product, quantity:uint=1)
{
}
```

Pamiętaj, że funkcje konstruktora muszą być funkcjami publicznymi i nie mogą posiadać typu.

6. W funkcji konstruktora przypisz właściwościom klasy wartości parametryczne. W tym przypadku zostały użyte jednakowe nazwy, należy więc poprzedzić nazwy właściwości po lewej stronie znaków równości przedrostkiem `this`. Przypisz również właściwości `subtotal` wartość wyrażenia właściwości `listPrice` produktu pomnożonego razy `quantity`.

```
public function ShoppingCartItem(product:Product, quantity:uint=1){
   this.product = product;
   this.quantity = quantity;
   this.subtotal = product.listPrice * quantity;
}
```

Pamiętaj, że funkcja konstruktora jest wywoływana automatycznie za każdym razem, gdy jest tworzony obiekt z klasy. Konstruktor ustawi właściwości, które zostały do niego przekazane — w tym przypadku będzie to egzemplarz klasy `Product` oraz ilość, która otrzymuje domyślną wartość równą `1`. Metoda ta zostanie użyta tylko podczas dodawania produktu do koszyka, logiczne jest więc zdefiniowanie ilości początkowej równej `1`.

7. Utwórz metodę publiczną o nazwie `recalc()`, która będzie przeliczała sumę każdego produktu (mnożyła wartość `listPrice` produktu razy `quantity`):

```
public function recalc():void{
   this.subtotal = product.listPrice * quantity;
}
```

   Kiedy użytkownik dodaje produkt do koszyka, należy przeprowadzić kalkulację w celu zaktualizowania całkowitej sumy. Należy także sprawdzić, czy element został już wcześniej dodany do koszyka — jeżeli tak, należy zaktualizować ilość. W następnej lekcji wyjaśnimy, jak to wykonać.

8. Utworzymy teraz nową klasę. Wybierz z menu *File/New/ActionScript class*. Ustaw wartość *Package* równą `valueObjects`, dodając automatycznie tę klasę do utworzonego wcześniej katalogu. Wprowadź `ShoppingCart` w polu nazwy i pozostaw wartości domyślne dla wszystkich pozostałych pól.

   Tworzymy nową klasę, która będzie samym koszykiem wypełnionym obiektami `shoppingCartItem`. Jej zadaniem będzie obróbka danych dla koszyka z zakupami. Zaprojektowaliśmy już wygląd i styl graficzny koszyka, możemy więc wprowadzić ekonomiczny układ logiczny do klasy `ShoppingCart`. W skład układu logicznego wchodzą: dodawanie produktów do koszyka, usuwanie ich z koszyka, aktualizacja produktów w koszyku itp.

9. Dodaj wyrażenie `import`, które pozwoli używać narzędzi Fleksa, takich jak wyrażenie `trace`, wewnątrz klasy:

```
package valueObjects{
   import flash.utils.*
   public class ShoppingCart{
   }
}
```

   Z tego samego powodu, z jakiego należy zaimportować klasy niestandardowe, aby mogły zostać wykorzystane, konieczne jest też zaimportowanie odpowiednich klas szkieletu. Będziemy używali funkcji `trace()` i innych narzędzi, co wymaga zaimportowania tych klas.

10. Utwórz szkielet publicznej metody `addItem()` typu `void` przyjmującej parametr o nazwie `item` typu `ShoppingCartItem`. Wewnątrz metody dodaj wyrażenie `trace`, które będzie śledziło produkt dodany do koszyka:

```
package valueObjects{
   import flash.utils.*
   public class ShoppingCart{
      public function addItem(item:ShoppingCartItem):void{
         trace(item.product);
      }
   }
}
```

   W metodzie tej będziemy dodawali nowe elementy do koszyka. W późniejszych lekcjach dodamy do niej logikę. W tej chwili metoda będzie tylko śledziła produkty dodawane do koszyka. Pamiętaj, że napisana wcześniej funkcja `toString()` jest wywoływana automatycznie, zawsze gdy jest śledzony egzemplarz klasy `Product`.

11. Otwórz w programie Flex Builder plik *EComm.mxml* i znajdź blok skryptu. Pod wyrażeniem `import` klasy `Product` zaimportuj klasy `ShoppingCartItem` i `ShoppingCart` z katalogu *valueObjects*, tak jak poniżej:

```
import valueObjects.ShoppingCartItem;
import.valueObjects.ShoppingCart;
```

Aby użyć klasy, Flex Builder potrzebuje wyrażenia `import` odwołującego się do adresu lub pakietu, w którym mieści się klasa.

12. Pod wyrażeniami `import` utwórz egzemplarz publiczny klasy `ShoppingCart` i nadaj mu nazwę `cart`, a następnie dodaj znacznik metadanych `[Bindable]`:

```
[Bindable]
public var cart:ShoppingCart = new ShoppingCart();
```

Kiedy użytkownik klika przycisk *Add To Cart*, należy wywołać metodę klasy `ShoppingCartItem` o nazwie `addItem()`, którą właśnie utworzyliśmy. Dzięki utworzeniu egzemplarza klasy w tym miejscu mamy pewność, że będzie ona dostępna w całej aplikacji.

13. Znajdź w bloku `<mx:Script>` metodę `prodHandler()`. Zaraz za nią dodaj szkielet nowej prywatnej metody typu `void` o nazwie `addToCart()`. Spraw, by przyjmowała parametr typu `Product` o nazwie `product`, tak jak poniżej:

```
private function addToCart(product:Product):void {
}
```

Powyższa metoda będzie wywoływana, gdy użytkownik kliknie przycisk *Add* i zostanie przesłany wybrany przez niego obiekt wartości `Product`. Jest to metoda pliku MXML i nie będzie wywoływana spoza tego pliku. W ten sposób można używać prywatnego identyfikatora oznaczającego, że do danej metody nie można uzyskać dostępu spoza jej klasy, co sprzyja lepszej ochronie danych.

14. Wewnątrz metody `addToCart()` utwórz nowy egzemplarz klasy `ShoppingCartItem` o nazwie `sci` i przekaż do jego konstruktora parametr `product`:

```
var sci:ShoppingCartItem = new ShoppingCartItem(product);
```

15. Wewnątrz metody `addToCart()` wywołaj metodę `addItem()` egzemplarza `cart` klasy `ShoppingCart`. Prześlij do metody utworzony obiekt `sci`:

```
cart.addItem(sci);
```

Kod ten wywoła metodę `addItem()` utworzonej wcześniej klasy `ShoppingCart`. W kolejnej lekcji pokażemy, jak wykonywać pętle przez strukturę danych w celu sprawdzenia, czy element został do niej dodany. Obecnie metoda ta po prostu śledzi nazwę produktu dodanego do koszyka.

16. Dodaj zdarzenie `click` do przycisku *Add To Cart*. Wywołaj metodę `addToCart()`, przekazując do niej egzemplarz `theProduct`:

```
<mx:Button id="add" label="Add To Cart"
   <click="addToCart(theProduct)"/>
```

17. Zapisz aplikację i uruchom debugowanie.

    Po każdym kliknięciu przycisku *Add To Cart* powinieneś widzieć tekst `[Product]Milk` pojawiający się w widoku *Console*.

Czego się nauczyłeś?

Podczas tej lekcji:

- zrozumiałeś, na czym polega obsługa zdarzeń (s. 108 – 110),
- obsługiwałeś zdarzenie `creationComplete` znacznika `<mx:Application>` w celu utworzenia struktury danych (s. 111 – 114),
- zbadałeś obiekt zdarzenia, aby zrozumieć, czym jest, i korzystałeś z dwóch właściwości tego obiektu (s. 114 – 117),
- zrozumiałeś podstawową strukturę klasy języka ActionScript (s. 118),
- tworzyłeś klasy języka ActionScript będące obiektami wartości (s. 118 – 123),
- tworzyłeś właściwości, metody i metody statyczne klas języka ActionScript (s. 123 – 129).