helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

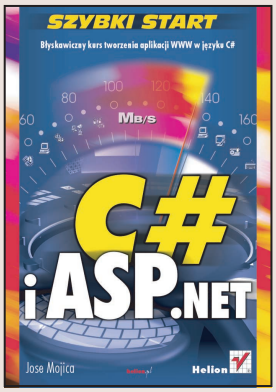
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C# i ASP.NET. Szybki start



Autor: Jose Mojica
Tłumaczenie: Wojciech Moch (rozdz. 1 - 6),
Radosław Meryk (rozdz. 7 - 13)
ISBN: 83-7361-389-7
Tytuł oryginału: C# Web Development with ASP.NET: Visual QuickStart Guide
Format: B5, stron: 456
Przykłady na ftp: 1127 kB

C# to całkiem nowy język programowania zaprojektowany przez firmę Microsoft. Wygląda on jak mieszanka C++ i Javy, jest jednak tak prosty jak Visual Basic. Z jego pomocą można tworzyć aplikacje WWW i programy wyposażone w graficzny interfejs użytkownika. Język ten związany jest z platformą .NET, umożliwiającą tworzenie zaawansowanych aplikacji biznesowych działających w środowisku sieciowym, w tym także z technologią budowania dynamicznych serwisów internetowych ASP.NET. Programy działające na platformie .NET można pisać w wielu językach programowania, ale wiele wskazuje na to, że właśnie C# stanie się najpopularniejszym z nich.

Książka „C# i ASP.NET. Szybki start” jest doskonałym podręcznikiem dla początkujących programistów. Jak każda pozycja z serii „Szybki start”, składa się z kilkudziesięciu rozdziałów, z których każdy przedstawia kolejne kroki, które należy wykonać, by osiągnąć zamierzony cel. Dodatkową zaletę stanowią liczne ilustracje.

Opisano między innymi:

- Instalację niezbędnego oprogramowania
- Składniki języka C#
- Instrukcje warunkowe i pętle
- Pracę z ciągami znaków
- Programowanie obiektowe w C#
- Korzystanie z tablic i kolekcji
- Delegaty, zdarzenie, obsługę błędów
- Tworzenie dynamicznych serwisów WWW w języku C#

Programista chcący tworzyć zaawansowane aplikacje internetowe ma wybór pomiędzy dwoma platformami: Java 2 EE Suna i .NET Microsoftu. Jeśli wybierze tę drugą, dzięki książce „C# i ASP.NET. Szybki start” szybko będzie mógł tworzyć funkcjonalne aplikacje WWW w nowym, ekscytującym języku C#.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp **11**

Rozdział 1. **Zaczynamy** **23**

Jak zdobyć C# ... 24
Instalowanie IIS ... 26
Tworzenie projektu aplikacji sieciowej w Visual Studio .NET ... 28
Uruchamianie projektów sieciowych za pomocą Visual Studio .NET ... 31
Pisanie prostej strony ASP .NET ... 33
Wyszukiwanie błędów w aplikacjach ASP .NET ... 35
Ręczne uruchamianie kompilatora ... 38
Kompilowanie i uruchamianie programów C# bez pomocy VS.NET ... 39
Wyszukiwanie błędów w programach bez pomocy VS.NET ... 41

Rozdział 2. **Składniki języka C#** **43**

Praca z elementami języka ... 44
Pisanie kodu C# ... 48
Deklarowanie zmiennych ... 50
Definiowanie stałych ... 53
Grupowanie stałych w typy wyliczeniowe ... 54
Deklarowanie funkcji ... 56
Deklarowanie funkcji z parametrami ... 58
Zwracanie wartości funkcji ... 62
Definiowanie klas ... 64
Dodawanie klas do przykładowej aplikacji ... 67
Tworzenie i stosowanie obiektów ... 68
Tworzenie obiektów w przykładowej aplikacji ... 70
Włączanie definicji klas ze źródeł zewnętrznych ... 71
Grupowanie klas w przestrzeniach nazw ... 74
Dodawanie pól do klas ... 78
Inicjalizowanie pól w momencie ich deklarowania ... 79
Dodawanie właściwości do klas ... 80

Dodawanie metod do klasy ... 86
Dodawanie składowych do klas przykładowego programu ... 89
Uruchamianie przykładowej aplikacji ... 90
Dodawanie komentarzy ... 95
Kierunki parametrów w typach referencyjnych ... 97

Rozdział 3. **Wyrażenia warunkowe i pętle** **99**
Praca z pętlami i warunkami ... 100
Porównywanie typów numerycznych ... 104
Porównywanie typów referencyjnych ... 105
Łączenie porównań ... 109
Instrukcje warunkowe ... 110
Sprawdzanie wielu warunków za pomocą instrukcji switch ... 112
Operator warunku ... 114
Dodawanie instrukcji warunkowych do przykładowego programu ... 115
Stosowanie pętli while ... 122
Stosowanie pętli do ... 123
Stosowanie pętli for ... 124
Zakończenie i kontynuacja pętli ... 126
Dodawanie pętli do przykładowego programu ... 129

Rozdział 4. **Ciągi znaków** **133**
Przygotowanie komputera do pracy z ciągami znaków ... 135
Praca z ciągami znaków ... 137
Inicjalizowanie ciągów znaków ... 140
Porównywanie ciągów znaków ... 142
Łączenie ciągów znaków ... 145
Określanie długości ciągu znaków ... 147
Porównywanie i łączenie ciągów znaków w przykładowej aplikacji ... 148
Tworzenie ciągów z pojedynczych znaków ... 151
Znaki specjalne ... 154
Stosowanie ciągów literałów ... 156
Dostęp do poszczególnych znaków ciągu ... 158
Wyszukiwanie podciągów wewnątrz ciągów znaków ... 159
Pobieranie wycinka ciągu ... 160
Dzielenie ciągu znaków ... 161
Składanie ciągów ... 163

Zamiana liter na małe lub wielkie ... 164
Formatowanie ciągów znaków ... 165
Uruchomienie przykładowej aplikacji ... 166
Zamiana obiektów na ciągi znaków ... 171
Tworzenie ciągów znaków za pomocą klasy StringBuilder ... 172

Rozdział 5. **Dziedziczenie klas** **173**
Praca z dziedziczeniem klas ... 174
Dziedziczenie z innej klasy ... 177
Udostępnianie składowych klas i ograniczanie dostępu do nich ... 180
Rozbudowa przykładowej aplikacji ... 183
Ukrywanie metod klasy bazowej ... 188
Pokrywanie funkcji w klasie wywiedzionej ... 190
Dodawanie standardowego klawisza do przykładowej aplikacji ... 194
Stosowanie nowego klawisza na formularzu WorkOrder ... 197
Dodawanie funkcji, które muszą zostać pokryte ... 200
Wymóg dziedziczenia ... 201
Blokowanie dziedziczenia ... 202

Rozdział 6. **Składowe specjalne** **203**
Dodawanie wielu funkcji o takich samych nazwach, czyli przeciążanie funkcji ... 204
Definiowane funkcji o zmiennej liczbie parametrów ... 207
Dodawanie konstruktorów ... 209
Wywoływanie konstruktora klasy bazowej ... 210
Dodawanie finalizerów ... 212
Tworzenie bibliotek kodu za pomocą składowych statycznych ... 213
Zmiana znaczenia operatora (przeciążanie operatorów) ... 216
Definiowanie równoważności klas przez przeciążenie operatora równości ... 220
Definiowanie równoważności klas przez pokrycie funkcji Equals ... 222
Praca ze składowymi specjalnymi ... 224

Rozdział 7. **Typy** **229**
Działania z typami ... 230
Uzyskiwanie typu klasy ... 234
Sprawdzanie zgodności typów ... 236

Konwersja jednego typu na inny 237
Rozwinięcie przykładowej aplikacji 240
Definiowanie reguł konwersji (przeciążanie operatora konwersji) 246

Rozdział 8. **Interfejsy** **249**
Działania z interfejsami 251
Definiowanie interfejsów 255
Niejawna implementacja składowych interfejsu 257
Jawna implementacja składowych interfejsu 260
Usprawnianie przykładowej aplikacji 261
Wykorzystanie obiektów poprzez interfejsy 268
Rozpoznawanie interfejsu 269
Wykorzystanie interfejsów do polimorfizmu 271
Interfejs będący pochodną innego interfejsu 272
Refaktoryzacja 274
Ponowna implementacja interfejsów w klasie pochodnej 275
Kończymy przykładową aplikację 277

Rozdział 9. **Tablice i kolekcje** **279**
Działania z tablicami i kolekcjami 281
Tworzenie tablic wartości 285
Tworzenie tablic typów referencyjnych 287
Poruszanie się po tablicy 291
Inicjowanie elementów tablicy w miejscu 294
Tworzenie tablic wielowymiarowych 296
Usprawnianie przykładowej aplikacji 298
Odnajdywanie elementów tablicy
za pomocą wyszukiwania liniowego 301
Sortowanie tablic 304
Odnajdywanie elementów tablicy
za pomocą wyszukiwania binarnego 307
Klasy działające jak tablice (definiowanie indeksatorów) 309
Wprowadzenie indeksatorów do przykładowej aplikacji 312
Kopiowanie tablic 313
Tworzenie list dynamicznych 314
Tworzenie kolejek 316

Tworzenie stosów 317
Tworzenie tabel mieszających 318
Przeglądanie elementów tabel mieszających 320
Kończymy przykładową aplikację 321
Testowanie kontrolki CodeGridWeb 323

Rozdział 10. **Delegaty i zdarzenia** **325**
Działania z delegatami i zdarzeniami 327
Deklarowanie delegatów 330
Tworzenie i wywoływanie delegatów 331
Łączenie delegatów 333
Odłączanie delegatów 334
Deklarowanie i wyzwalanie zdarzeń 336
Deklarowanie zdarzeń przyjaznych dla WWW 338
Subskrypcja zdarzeń 340
Asynchroniczne wyzwalanie delegatów 342
Oczekiwanie na zakończenie działania delegatów asynchronicznych 346
Pobieranie wyników działania delegatów asynchronicznych 348
Kończymy przykładową aplikację 350

Rozdział 11. **Obsługa błędów** **353**
Wykonywanie działań z wyjątkami 354
Przechwytywanie wyjątków 360
Przechwytywanie wybranych wyjątków 363
Uzyskiwanie informacji o wyjątku 366
Działania z łańcuchami wyjątków 368
Deklarowanie własnych wyjątków 372
Definiowanie komunikatów o błędach 374
Zgłaszanie wyjątku 375
Przechwytywanie i ponowne zgłaszanie wyjątków 376
Tworzenie łańcucha wyjątków 377
Wprowadzanie kodu,
który wykonuje się przed zakończeniem działania funkcji 378
Zastosowanie słowa kluczowego using 380
Wprowadzanie zabezpieczeń formularzy do przykładowej aplikacji 382
Obsługa nieobsłużonych błędów w aplikacjach WWW 385

Rozdział 12. **Odbicia i atrybuty** **389**
Działania z odbiciami i atrybutami ... 390
Identyfikacja kompilata ... 394
Działania z wyświetlanymi nazwami ... 395
Działania z ciągami znaków opisu ścieżek ... 398
Dynamiczne ładowanie programu na podstawie nazwy wyświetlanej ... 399
Dynamiczne ładowanie programu na podstawie ścieżki dostępu ... 400
Tworzenie egzemplarza klasy w kompilacie ... 401
Uzyskiwanie listy klas w kompilacie ... 403
Wyszczególnienie składowych klasy ... 404
Dynamiczne ustawianie lub pobieranie pól ... 406
Dynamiczne wywoływanie metody ... 409
Kończymy zadanie pierwsze w przykładowej aplikacji ... 411
Stosowanie atrybutów kodu ... 413
Definiowanie atrybutów ... 414
Wyszukiwanie atrybutów w kodzie ... 417
Kończymy zadanie drugie w przykładowej aplikacji ... 420

Rozdział 13. **Projekty WWW w języku C#** **423**
Tworzenie projektu DLL za pomocą Visual Studio .NET ... 424
Odwoływanie się do kodu DLL i jego wykonywanie ... 428
Udostępnianie bibliotek DLL do globalnego wykorzystania ... 429
Tworzenie serwisów WWW ... 432
Wykorzystywanie serwisów WWW ... 436

Skorowidz **439**

Interfejsy 8

Interfejsy to typy pozwalające na definiowanie wielu klas spełniających podobne funkcje. Wyobraźmy sobie klasę `Escort`. W klasie `Escort` są takie metody, jak: `Jazda`, `Zatrzymanie`, `OdtwarzanieMuzyki` itp. Użytkownik jest zadowolony z klasy `Escort`, ponieważ umożliwia mu ona dojazdy do pracy i z powrotem. W pewnym momencie jego koledzy zaczęli jednak żartować z klasy `Escort`, dlatego postanowił zastąpić ją klasą `Ferrari`. Przed zastąpieniem klasy `Escort` klasą `Ferrari` użytkownik chce mieć jednak pewność, że klasa ta spełnia co najmniej takie same funkcje, co klasa Escort. Musi więc oferować metody `Jazda`, `Zatrzymanie`, `OdtwarzanieMuzyki`. Wszystkie te działania mogą być wykonywane inaczej niż w klasie `Escort`, ale ważne jest, aby istniały co najmniej takie same metody. Ich zestaw nazywa się interfejsem. Interfejs może np. nosić nazwę `ISamochod`. Wszystkie samochody, aby mogły być tak nazywane, będą musiały zawierać implementację interfejsu `ISamochod` (przynajmniej z naszego punktu widzenia).

W świecie, w którym żyjemy, obowiązują zasady interfejsów. Implementacjami podobnych interfejsów są, na przykład, `Ludzie`. Wielu z nas oddycha i je (jeśli ktoś się z tym nie zgadza, proszę natychmiast przestać czytać tę książkę). Każdy egzemplarz klasy `Ludzie` może w inny sposób oddychać lub jeść, ale, niezależnie od tego, takie firmy, jak np. McDonald będą mogły odnosić sukcesy tylko wtedy, gdy zostanie wykonana metoda `Jedzenie` należąca do interfejsu `ICzlowiek`.

W terminologii języka C# interfejsy są typami. W stosunku do definicji interfejsów w języku C++ różnią się tym, że nie są to wyspecjalizowane klasy. Są to natomiast oddzielne typy. Zgodnie z podstawową definicją, interfejs jest zbiorem definicji funkcji (tylko definicji, bez kodu implementacji). Funkcje te same w sobie nie wykonują żadnych działań. Przypominają interfejs `ICzlowiek` bez konkretnego egzemplarza człowieka — to tylko pojęcie, a nie rzeczywista implementacja. Aby można było skorzystać z interfejsu, należy go zaimplementować w obrębie klasy. Kiedy klasa implementuje interfejs, ogłasza wszem i wobec, że obsługuje wszystkie zdefiniowane w nim metody. Jeżeli sprzedawca używanych samochodów twierdzi, że jego towar jest implementacją interfejsu `ISamochod`, to tak samo, jakby mówił, że zawiera co najmniej metody: `Jazda`, `Zatrzymanie`, `OdtwarzanieMuzyki`. Jeżeli któraś z metod interfejsu nie zostanie zaimplementowana, kompilator języka C# zgłosi błąd.

Zastosowanie interfejsów pozwala programistom pisać kod wykorzystujący funkcje i nie martwić się ich implementacją. Dzięki temu można rozpocząć od jakiejś implementacji, a potem całkowicie ją zastąpić bez konieczności przepisywania kodu. Sprawdza się to w przypadku interfejsu `ISamochod`. Kierowcy nie uczą się prowadzenia konkretnego samochodu — uczą się prowadzić dowolny samochód, ponieważ wszystkie samochody realizują ten sam, podstawowy interfejs.

Posłużmy się, jako przykładem rzeczywistego zastosowania interfejsów, technologią ADO .NET, która pozwala na realizację połączenia z bazą danych. Firma Microsoft, można w to wierzyć lub nie, nie ma zamiaru wymuszać użycia tylko jednej bazy danych. Zamiast zmuszać do użycia tego samego zestawu klas, zdefiniowała interfejs funkcji umożliwiających zrealizowanie połączenia z bazą danych. Producenci baz danych mają więc możliwość napisania klas będących implementacją tego interfejsu. Idea kodowania z wykorzystaniem interfejsów polega na tym, że można zmienić wykorzystywany pakiet bazy danych bez konieczności przepisywania aplikacji.

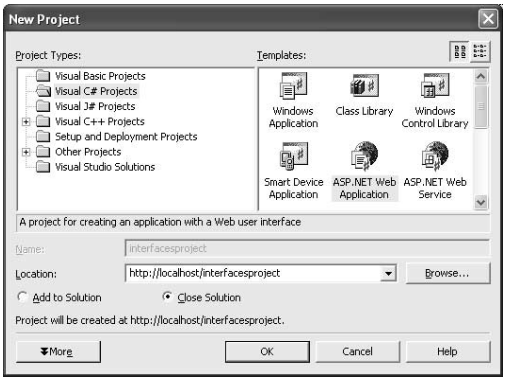
Działania z interfejsami

Jako przykładową aplikację w tym rozdziale napiszemy moduł ASP .NET. Nie będzie to aplikacja bardzo rozbudowana, ale za to będzie ją można wykorzystać w rzeczywistych aplikacjach (w tym rozdziale nie będzie aplikacji dotyczącej supermenów).

A zatem, czym jest moduł ASP .NET? Moduł jest klasą, która może przechwytywać żądania dowolnych stron w aplikacji. Moduły wykorzystuje się do wykonywania takich działań, jak np. testy zabezpieczeń. Mogą one przechwycić każde żądanie i albo je zatrzymać, albo kontynuować lub też zmodyfikować w pewien sposób. Moduły mogą także wykonywać działania po obsłudze żądania przez odpowiednią stronę. Programistom doświadczonym w programowaniu aplikacji WWW z pewnością pomoże wyjaśnienie, że moduły są odpowiednikiem filtrów IIS w środowisku .NET.

Celem modułu, który utworzymy, będzie dostarczenie wszystkim stronom informacji o rozmiarach obszaru klienta w przeglądarce. Obszar klienta w przeglądarce to miejsce, w którym jest wyświetlana strona WWW (wewnątrz okna przeglądarki, bez pasków menu i stanu). Interesujące, że środowisko .NET nie przekazuje tych wymiarów za pośrednictwem żadnej z właściwości. Można jednak uzyskać te wielkości za pomocą skryptów działających po stronie serwera (VBScript lub JavaScript działające na komputerze-kliencie, a nie na serwerze tak, jak strony ASP). Można zapytać, dlaczego interesują nas te wymiary? Otóż są nam potrzebne, ponieważ czasami chcemy zastosować grafikę o mniejszych wymiarach, jeśli wymiary okna są mniejsze lub większych wymiarach, jeśli wymiary okna są większe.

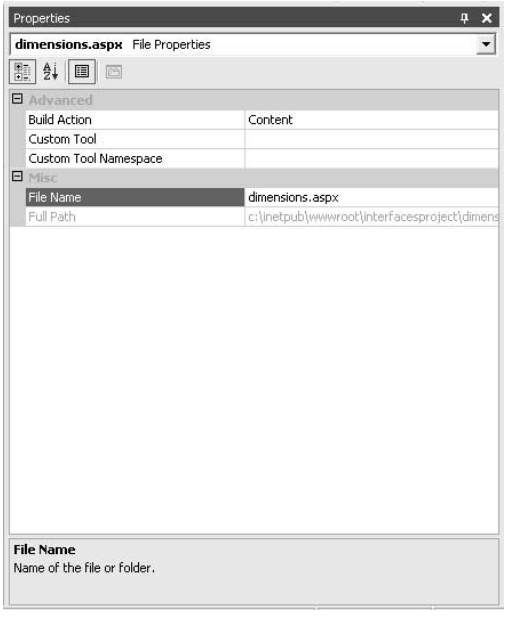
Rozwiązanie problemu jest następujące. Kiedy klient zażąda strony po raz pierwszy, moduł zatrzyma żądanie i zwróci stronę z odpowiednim skryptem działającym po stronie klienta. Zadaniem skryptu będzie określenie wymiarów obszaru klienta. Skrypt zapisze wymiary w ukrytym polu, a następnie zażąda natychmiast tej samej strony, której klient żądał poprzednio. Wysłanie skryptu działającego po stronie klienta i zwrócenie pierwotnie żądanej strony powinno nastąpić na tyle szybko, aby użytkownik niczego nie zauważył. Za drugim razem, gdy moduł otrzyma żądanie tej samej strony (teraz zażąda jej skrypt działający po stronie klienta), moduł odnajdzie ukryte pola zawierające wymiary, utworzy niewielki obiekt służący do przechowywania potrzebnych informacji i przekaże je do żądanej strony. Strona pobierze obiekt z informacjami i zwróci wynik do klienta, który wykorzysta informacje o rozmiarze obszaru klienta. W naszym przypadku na stronie wymiary wyświetlone zostaną w polach tekstowych.



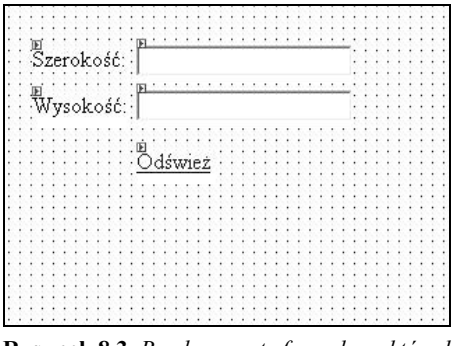
Rysunek 8.1. *Do utworzenia modułu potrzebny jest tylko projekt biblioteki, a nie cały projekt WWW ASP .NET. Jednak utworzenie projektu WWW ASP .NET pozwala na napisanie i przetestowanie modułu w jednym projekcie*

Aby napisać moduł:

1. Uruchom Visual Studio .NET (*Start/ Programy/Microsoft Visual Studio .NET*).
2. Wybierz polecenie *File/New/Project*. Na ekranie pojawi się okno dialogowe *New Project*.
3. W panelu *Project Types* po lewej stronie okna *New Project* kliknij folder *Visual C# projects*.
4. Zaznacz ikonę *ASP .NET Web Application* i zmień nazwę aplikacji na *interfacesproject* (rysunek 8.1).
5. Program *Visual Studio* utworzy nowy projekt i otworzy formularz *WebForm1.aspx*.
6. Zmień nazwę formularza na *dimensions.aspx*. W tym celu wybierz polecenie *View/Solution Explorer* z górnego paska menu.



Rysunek 8.2. *Zmiana nazwy pliku nie powinna sprawiać teraz żadnych problemów*



Rysunek 8.3. *Bardzo prosty formularz, który łatwo utworzyć od nowa. Zawiera dwie etykiety, dwa pola tekstowe oraz jedno łącze*

7. Kliknij prawym klawiszem myszy formularz *dimensions.aspx* i wybierz polecenie *Properties*. W tabelce właściwości, pokazanej poniżej na rysunku 8.2, zmień właściwość *FileName* z *WebForm1.aspx* na *dimensions.aspx*.

8. Zmodyfikuj formularz *dimension.aspx* w taki sposób, aby wyglądał tak, jak formularz pokazany na rysunku 8.3. Oczywiście, w ręczne wykonanie tego zadanie trzeba włożyć wiele pracy. Zamiast tego możesz wprowadzić kod HTML bezpośrednio w edytorze. Na rysunku 8.4, na następnej stronie, zaprezentowano kod HTML niezbędny do utworzenia formularza. Aby bezpośrednio wprowadzić kod HTML, kliknij przycisk *HTML* w oknie edytora. Alternatywnie możesz pobrać plik szkieletu dla tego projektu (patrz „Wskazówki” na następnej stronie).

Wskazówki

- To, co do tej pory zrobiliśmy w przykładowej aplikacji, nie ma nic wspólnego z modułem użytkownika. Formularz projektowany w tym podrozdziale zostanie użyty do przetestowania modułu. Kod modułu zostanie dodany natychmiast po tym, jak dowiemy się czegoś na temat implementacji interfejsów.
- Podobnie jak w przypadku innych projektów w tej książce, utworzenie projektu nie jest konieczne do poznania pojęć zaprezentowanych w niniejszym rozdziale.
- Szkielety wszystkich projektów można pobrać z witryny WWW Wydawnictwa Peachpit pod adresem *http://www.peachpit.com/vqs/csharp*.

Rysunek 8.4. *Kod HTML pokazany na tym rysunku można wykorzystać jako wzorzec. Dzięki formularzowi możliwe jest uzyskanie informacji o wymiarach i pozycji wszystkich elementów sterujących*

Kod

```
<%@ Page language = "c#"
Codebehind = "dimensions.aspx.cs"
        AutoEventWireup="false"
        Inherits =
"interfacesproject.WebForm1"%>
<HTML>
    <HEAD>
        <title>WebForm1</title>
    </HEAD>
    <body MS_POSITIONING="GridLayout">
        <form id="Form1" method = "post"
        runat="server">
            <asp:label id="lblWidth"
                style="Z-INDEX: 101; LEFT:
                25px;
                POSITION: absolute; TOP: 37px"
                runat="server">
                Szerokość:
           </asp:label>
            <asp:label id="lblHeight"
                style="Z-INDEX: 102; LEFT:
                29px;
                POSITION: absolute; TOP: 70px"
                runat="server">
                Wysokość:
           </asp:label>
            <asp:textbox id="txtWidth"
                style="Z-INDEX: 103; LEFT:
                84px;
                POSITION: absolute; TOP: 36px"
                runat="server">
           </asp:textbox>
            <asp:textbox id="txtHeight"
                style="Z-INDEX: 104; LEFT:
                84px;
                POSITION: absolute; TOP: 68px"
                runat="server">
           </asp:textbox>
            <asp:hyperlink id="lnkSelf"
                style="Z-INDEX: 105; LEFT:
                89px;
                POSITION: absolute; TOP: 110px"
                runat="server"
                NavigateUrl="dimensions.aspx">
                Odśwież
            </asp:hyperlink>
        </form>
    </body>
</HTML>
```

Tabela 8.1. *Składowe interfejsu (składowe, które można wprowadzić do definicji interfejsu)*

Nazwa	Przykład
Metoda	`string Method();`
Właściwość	`int Age{ get; set}`
Właściwość tylko do odczytu	`int Age{get;}`
Indeks	`long this[int index]{get; set:}`
Zdarzenie	`event EventHandler OnClick;`

Rysunek 8.5. *Składowe interfejsu to tylko definicje, które nie zawierają kodu. W definicjach nie stosuje się także modyfikatorów dostępu — wszystkie metody interfejsu są publiczne*

Kod

```
public interface IHuman
{
   string Name { get; }   //właściwość tylko
                          // do odczytu
   void Eat(int Amount);  //metoda
   void Breathe();        //metoda
}
```

Definiowanie interfejsów

Interfejsy definiuje się za pomocą słowa kluczowego `interface`. Dla interfejsów można definiować metody, właściwości, delegaty, zdarzenia itp. Każdy z elementów wewnątrz interfejsu musi być jednak tylko deklaracją elementu i nie może zawierać implementacji.

Aby zdefiniować interfejs:

1. Wpisz słowo kluczowe `public` lub `internal`, w zależności od zasięgu, jaki ma mieć interfejs.
2. Wprowadź słowo kluczowe `interface`, a po nim spację.
3. Wpisz nazwę interfejsu.
4. Otwórz nawias klamrowy {.
5. Wprowadź definicje składowych (tabela 8.1).
6. Zamknij nawias klamrowy } (rysunek 8.5).

Wskazówki

- Wszystkie metody interfejsu z definicji są publiczne — w definicji metody nie można wprowadzić modyfikatorów dostępu (dotyczy to także modyfikatora `public`).
- Danych typu interfejsu można użyć w definicjach parametrów lub zmiennych (rysunek 8.6). W przypadku interfejsów nie można jednak tworzyć nowych egzemplarzy — na przykład nie można napisać `new IAccount`.
- W interfejsach można przeciążać metody (rysunek 8.7).
- Jeżeli jakiś programista użyje interfejsu, najlepiej nie zmieniać jego definicji. W przeciwnym razie istnieje ryzyko, że napisany program przestanie działać. W takiej sytuacji najlepiej zdefiniować nowy interfejs. Więcej informacji na ten temat znajdzie się w dalszej części tego rozdziału, w podrozdziale „Tworzenie interfejsów pochodnych od innych interfejsów".

Rysunek 8.6. *Po zdefiniowaniu interfejsu można go wykorzystać jako typ danych w deklaracjach zmiennych lub parametrów*

Kod

```
class DailyRoutine
{
    void Tasks()
    {
        IHuman man;
        IHuman woman;
    }

    public void GoToDinner(IHuman person)
    {
        person->Eat();
    }
}
```

Rysunek 8.7. *Przeciążanie metod to możliwość istnienia kilku metod o tej samej nazwie. Pamiętaj, że można to zrobić tylko pod warunkiem, że zmieni się liczbę parametrów lub zmodyfikuje typ jednego z nich*

Kod

```
public interface IHuman
{
    string Name {get; }
    //właściwość tylko do odczytu
    void Eat(int Amount); //metoda
    void Eat(string foodType, int Amount);
    //przeciążanie
}
```

Rysunek 8.8. *Implementacja interfejsu przypomina dziedziczenie. W przypadku metody niejawnej wystarczy zaimplementować składowe interfejsu jako metody publiczne*

Kod

```
public interface IHuman
{
    string Name {get; }
    void Sleep(short hours);
}

public class Person : IHuman
{
    public string Name
    {
        get
        {
            return "Paul";
        }
     }

    public void Sleep(short hours)
    {
        for(short counter=0;
        counter<hours; counter++)
        {
          HttpContext context =
          HttpContext.Current;
          context.Response.Write("Chrapie!");
        }
    }
}
```

Niejawna implementacja składowych interfejsu

Interfejsy są implementowane przez klasy. Zaimplementowanie interfejsu oznacza wprowadzenie kodu dla wszystkich zdefiniowanych w nim metod. Kompilator wymusza wprowadzenie kodu dla wszystkich metod bez wyjątku. Wynika to stąd, że programista wykorzystujący interfejs spodziewa się, że klasa implementująca interfejs będzie zawierać definicje wszystkich zawartych w nim metod. Istnieją dwa mechanizmy implementowania interfejsów w klasie: mechanizm niejawny i jawny. Najpierw zaprezentujemy mechanizm niejawny.

Aby zaimplementować interfejs w sposób niejawny:

1. Po nazwie klasy, która będzie implementować interfejs, wpisz dwukropek, a po nim nazwę interfejsu, który będzie implementowany.
2. Wprowadź składowe odpowiadające wszystkim składowym interfejsu.
3. Oznacz metody implementacyjne jako publiczne.
4. Wpisz kod składowych interfejsu (rysunek 8.8).

Wskazówki

- Ten mechanizm implementacji interfejsu cechuje wada polegająca na tym, że każdą metodę implementacyjną należy oznaczyć jako publiczną.
- Aby zaimplementować więcej niż jeden interfejs, należy oddzielić poszczególne nazwy interfejsu przecinkiem (rysunek 8.9).

Rysunek 8.9. *Klasa Person implementuje zarówno interfejs IHuman, jak i IManager. Zaimplementowanie interfejsu ustanawia relację „jest". Innymi słowy, osoba (Person) jest człowiekiem (Human) i menedżerem (Manager)*

Kod

```
public interface IHuman
{
    string Name {get; }
    void Sleep(short hours);
}

public interface IManager
{
    void SpyOnEmployee(string name, bool
    IsUnaware);
}

public class Person : IHuman, IManager
{
    public string Name
    {
        get
        {
            return "Paul";
        }
     }

    public void Sleep(short hours)
    {
        for(short counter=0;
        counter<hours; counter++)
        {
          HttpContext context =
          HttpContext.Current;
          context.Response.Write("Chrapie!");
        }
    }

    public void SpyOnEmployee(string name,
    bool IsUnaware)
    {
        while (IsUnaware)
        {
            LookOverShoulder(name);
        }
    }
}
```

Rysunek 8.10. *W powyższym kodzie zdefiniowano klasę Person, która implementuje interfejs IHuman. Następnie, biorąc pod uwagę to, że nie wszystkie osoby są menedżerami, zdefiniowano klasę opisującą menedżerów. Jednak, ze względu na to, że menedżerowie są ludźmi (persons), klasa ta dziedziczy wszystkie składowe klasy Person (zasada dziedziczenia). Na końcu znajduje się implementacja interfejsu IManager*

Kod

```
public interface IHuman
{
    string Name {get; }
    void Sleep(short hours);
}

public interface IManager
{
    void SpyOnEmployee(string name, bool
    IsUnaware);
}

public class Person : IHuman
{
    public string Name
    {
        get
        {
            return "Paul";
        }
     }

    public void Sleep(short hours)
    {
        for(short counter=0;
        counter<hours; counter++)
        {
          HttpContext context =
          HttpContext.Current;
          context.Response.Write("Chrapie!");
        }
    }
}
public class Manager : Person, IManager
    public void SpyOnEmployee(string name,
    bool IsUnaware)
    {
        while (IsUnaware)
        {
            LookOverShoulder(name);
        }
    }
}
```

- Jeżeli klasa jest pochodną innej klasy (nie interfejsu), najpierw należy wymienić klasę, która nie należy do interfejsu, a następnie implementowane interfejsy. Wszystkie te elementy trzeba oddzielić przecinkami (rysunek 8.10).

Jawna implementacja składowych interfejsu

Podczas implementowania interfejsu przez wprowadzanie składowych publicznych powstaje problem polegający na tym, że czasami dwa interfejsy zawierają metodę o takiej samej nazwie i takich samych parametrach, ale konieczne jest zaimplementowanie tych metod na dwa różne sposoby. W przypadku zastosowania metody niejawnej jedna metoda publiczna będzie stanowiła implementację metod o takiej samej nazwie i tym samym zbiorze parametrów dla wszystkich interfejsów. Metoda jawna umożliwia poinformowanie kompilatora o tym, którą metodę, którego interfejsu mieliśmy zamiar zaimplementować.

Aby jawnie zaimplementować interfejs:

1. Po nazwie klasy, która będzie implementować interfejs, wpisz dwukropek, a po nim nazwę interfejsu, który będzie implementowany.
2. Wpisz typ zwracanej wartości dla składowej interfejsu.
3. Wpisz `INazwaInterfejsu.MetodaInterfejsu`. Mówiąc inaczej, wpisz nazwę interfejsu, po nim kropkę, a po niej nazwę metody interfejsu.
4. Wprowadź parametry składowej interfejsu.
5. Wpisz kod składowej interfejsu (rysunek 8.11).

Wskazówka

- Wszystkie metody implementowane w ten sposób są prywatne. Nie można wprowadzić modyfikatora dostępu na początku definicji.

Rysunek 8.11. *W przypadku jawnego implementowania interfejsu nie wprowadza się modyfikatorów dostępu. Należy wpisać nazwę interfejsu, po nim kropkę, a po niej nazwę metody*

Kod

```
public interface IHuman
{
    string Name {get; }
    void Sleep(short hours);
}

public class Person : IHuman
{
    string IHuman.Name
    {
        get
        {
            return "Paul";
        }
     }

    void IHuman.Sleep(short hours)
    {
        for(short counter=0;
        counter<hours; counter++)
        {
          HttpContext context =
          HttpContext.Current;
          context.Response.Write("Chrapie!");
        }
    }
}
```

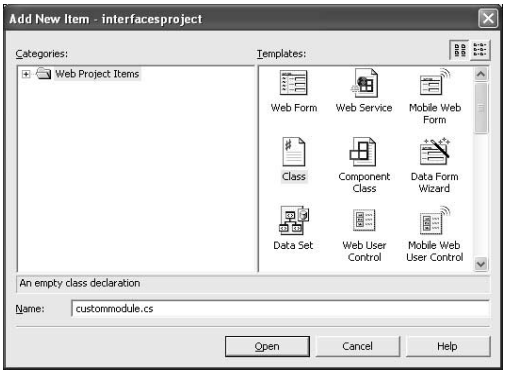
Usprawnianie przykładowej aplikacji

Teraz, kiedy wiemy już, w jaki sposób definiuje się interfejsy oraz jak należy implementować interfejsy za pomocą klas, czas usprawnić przykładową aplikację. Dokładniej rzecz ujmując, zdefiniujemy klasę, która będzie służyć jako moduł użytkownika. Aby przekształcić klasę w moduł, należy zaimplementować w niej interfejs `System.Web.IHttpModule`. Jego zaimplementowanie nie jest trudne — zawiera tylko dwie metody: `Init` i `Dispose`. Środowisko ASP .NET wywołuje metodę `Init` w momencie, gdy moduł jest po raz pierwszy ładowany do pamięci. Metoda `Dispose` wywoływana jest w momencie usuwania modułu z pamięci. Jedyna trudność polega na wykorzystaniu zdarzenia. Zdarzenia zostaną opisane w rozdziale 10. „Delegaty i zdarzenia".

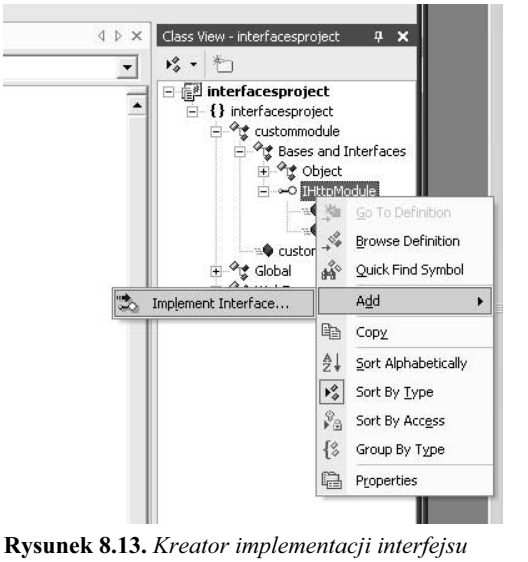
Nie zamierzamy poświęcać w tym rozdziale wiele miejsca na tłumaczenie, czym jest zdarzenie. Na razie wystarczy, abyśmy wiedzieli, że zdarzenie jest metodą wywoływaną w wyniku działania. Implementacja zdarzeń pod wieloma względami przypomina implementację interfejsów. Różnica polega na tym, że interfejs może zawierać kilka metod, natomiast zdarzenie zawiera tylko jedną metodę. Przykładem zdarzenia jest kliknięcie przycisku. Klasa żąda od zdarzenia `click` informacji od przycisku formularza. Kiedy użytkownik kliknie przycisk, serwer informuje klasę, że miało miejsce zdarzenie. Moduł można wykorzystać np. do nasłuchiwania wystąpienia zdarzenia `BeginRequest`. Środowisko ASP .NET wyzwala to zdarzenie za każdym razem, gdy przeglądarka klienta zażąda strony.

Aby zdefiniować klasę modułu użytkownika:

1. Z paska menu wybierz polecenie *Project/ Add class*. Wpisz *custommodule.cs* jako nazwę klasy i wciśnij *Enter* (rysunek 8.12).
2. W górnej części kodu modułu, poniżej wiersza o treści `using System`, wpisz wiersz `using System.Web`.
3. W wierszu w postaci: `public class custommodule` dodaj na końcu: `IHttpModule` tak, aby wiersz ten przyjął postać: `public class custommodule: IHttpModule`.
4. Teraz trzeba trochę „pooszukiwać”. W systemie Visual Studio .NET znajduje się kreator pozwalający na łatwą implementację interfejsu. Z menu wybierz polecenie *View/Class View*.
5. W oknie *Class View* rozwiń folder *interfacesproject/custommodule/Bases and interfaces*. Kliknij prawym klawiszem myszy *IHttpModule* i z rozwijanego menu wybierz *Add/Implement Interface* (rysunek 8.13).



Rysunek 8.12. *Wszystkie właściwości funkcjonalne modułu użytkownika zostaną zaimplementowane w klasie custommodule. Dzięki wykorzystaniu okna dialogowego pokazanego na rysunku użyjemy kreatora do wygenerowania szkieletu klasy*



Rysunek 8.13. *Kreator implementacji interfejsu wykorzystuje mechanizm niejawnej implementacji. Dla każdej składowej interfejsu definiowane są elementy publiczne*

Rysunek 8.14. *Interfejs IHttpModule spełnia dwie funkcje. Po pierwsze, informuje środowisko ASP .NET, że klasa jest modułem użytkownika. Po drugie, środowisko zyskuje możliwość poinformowania klasy, kiedy moduł jest ładowany po raz pierwszy (metoda Init) oraz kiedy nie jest już potrzebny (metoda Dispose)*

Kod

```
using System;
using System.Web;

namespace interfacesproject
{
   /// <summary>

   /// Ogólny opis modułu custommodule
   /// </summary>
   public class custommodule : IHttpModule
   {
      public custommodule()
      {
         //
         // DO ZROBIENIA: Tutaj należy
         //wprowadzić logikę konstruktora
         //
      }

      #region Implementation of IHttpModule
      public void
      Init(System.Web.HttpApplication context)
      {
      }

      public void Dispose()
      {
      }
      #endregion
    }
}
```

6. Kreator spowoduje dodanie namiastek metod `Init` i `Dispose`. Kod przyjmie postać widoczną na rysunku 8.14.
7. Dodaj do klasy metodę `BeginRequest` (rysunek 8.15).

Rysunek 8.15. *Metoda BeginRequest nie jest częścią interfejsu IHttpModule. Jest to zdarzenie zdefiniowane dla klasy HttpApplication, informujące o tym, że występują żądania do dowolnej ze stron w aplikacji*

Kod

```
using System;
using System.Web;

namespace interfacesproject
{
/// <summary>

   /// Ogólny opis modułu custommodule
   /// </summary>
   public class custommodule : IHttpModule
   {
      public custommodule()
      {
         //
         // DO ZROBIENIA: Tutaj należy wprowadzić
         //logikę konstruktora
         //
      }

      #region Implementation of IHttpModule
      public void
      Init(System.Web.HttpApplication context)
      {
      }

      public void Dispose()
      {
      }
      #endregion

      void BeginRequest(object sender,
      EventArgs args)
      {
      }
   }
}
```

8. W metodzie `Init` wprowadź kod wiążący zdarzenie `BeginRequest` z metodą klasy `BeginRequest` (rysunek 8.16). Nie martw się, jeżeli na razie kod ten jest niejasny, jego szczegółowe wyjaśnienie znajdzie się w rozdziale 10. „Delegaty i zdarzenia".

Rysunek 8.16. *Zwróć uwagę, że w celu powiązania zdarzenia klasy z metodą do zdarzenia odwołujemy się w taki sposób, jakby było ono właściwością obiektu (obiekt.nazwazdarzenia), następnie używamy operatora += i przypisujemy mu nowy obiekt (delegat). Metodę przekazujemy jako parametr konstruktora obiektu*

Kod

```
using System;
using System.Web;

namespace interfacesproject
{
   /// <summary>

   /// Ogólny opis modułu custommodule
   /// </summary>
   public class custommodule : IHttpModule
   {
      public custommodule()
      {
         //
         //DO ZROBIENIA: Tutaj należy
         //wprowadzić logikę konstruktora
         //
      }

      #region Implementation of IHttpModule
      public void
      Init(System.Web.HttpApplication context)
      {
          context.BeginRequest += new
          EventHandler(BeginRequest);
      }

      public void Dispose()
      {
      }
      #endregion

      void BeginRequest(object sender,
      EventArgs args)
      {
      }
   }
}
```

Rysunek 8.17. *Doceń piękno literałów znakowych. Zwróć uwagę, jak łatwo — korzystając z literałów znakowych — można umieścić znaki końca wiersza w ciągu znaków. Po prostu formatujemy ciąg znaków, tak jak chcemy, umieszczając gdzie należy znaki końca wiersza*

Kod

```
public void Init(System.Web.HttpApplication
context)
{
    context.BeginRequest += new
EventHandler(BeginRequest);
    string html = @"
    <html>
    Ładowanie...
    <script language=vbscript>
    Sub GetDimensions()
        Dim clientWidth
        Dim clientHeight

        clientWidth = document.body.ClientWidth
        clientHeight =
        document.body.ClientHeight
        document.myform.txtDimensions.Value
        =clientWidth & "";"" & clientHeight &
        "";""
        document.myform.submit
    End Sub
    </script>
    <body onLoad =""GetDimensions"">
    <form name=""myform"" method=""POST""
    action = ""{0}"">
    <p><input type=""hidden""
    name=""txtDimensions""
    size=""20""></p>
    </form>
    </body>
    </html>";

context.Application["__DiscoverHTML"]=html;
}
```

9. Wprowadź kod w metodzie `Init` w celu utworzenia ciągu znaków stanowiącego skrypt działający po stronie klienta i zapisz go w obiekcie `Application`. Pamiętaj, że obiekt `Application` jest dostępny na wszystkich stronach i dla każdego klienta (rysunek 8.17).

10. Teraz, w treści metody `BeginRequest` wpisz kod pokazany na rysunku 8.18. Zadaniem tego kodu jest sprawdzenie, czy są dostępne informacje o wymiarach. Informacje te powinny być dostępne za pośrednictwem pola ukrytego. Jeżeli pole tekstowe nie istnieje, moduł zablokuje żądanie strony i wyśle skrypt działający po stronie klienta. Skrypt zarejestruje wymiary okna w ukrytym polu i natychmiast zażąda strony ponownie. Moduł sprawdzi, czy pole tekstowe istnieje i, jeżeli istnieje, wprowadzi wymiary do obiektu `Item` (więcej informacji na ten temat znajdzie się w ramce na następnej stronie).

Wskazówki

- Kod zaprezentowany w tym podrozdziale to kompletny kod potrzebny do utworzenia modułu użytkownika. Jednak moduł ten nie będzie działać dopóty, dopóki nie uzupełnimy przykładu kodem pokazanym w dalszej części tego rozdziału.
- Kreator implementujący interfejsy wprowadza kod `#region Implementation of IHttpModule` na początku bloku metod implementacyjnych oraz `#endregion` na końcu tego bloku. Te instrukcje nie dotyczą bezpośrednio kompilatora. Umożliwiają one uzyskanie rozwijanego kodu w edytorze. Jak można zauważyć, w edytorze pojawi się znak minus z lewej strony deklaracji regionu (rysunek 8.19). Kliknięcie znaku minus w edytorze spowoduje ukrycie kodu umieszczonego w obszarze i zastąpienie znaku minus plusem (rysunek 8.20). Kliknięcie znaku plus spowoduje ponowne pojawienie się kodu.

Rysunek 8.18. *Na tym rysunku znalazł się kod, który wymaga wyjaśnienia. Szczegółowe informacje na jego temat można znaleźć w ramce pt. „Szczegóły zdarzenia BeginRequest”*

Kod

```
void BeginRequest(object sender, EventArgs
args)
{
    Global g = (Global)sender;
    string dimensions =
    g.Request.Form["txtDimensions"];
    if (dimensions == null || dimensions ==
    "")
    {
        string html = (string)g.Context.
        ➥Application["__DiscoverHTML"];
        string newUrl = string.Format(html,
        g.Request.Url.AbsolutePath);
        g.Response.Write(newUrl);
        g.CompleteRequest();
    }
    else if
    (dimensions.IndexOf("ClientWidth") == -1)
    {
       string[] sizes = dimensions.Split(';');
       int clientWidth = System.Convert.
       ToInt32(sizes[0]);
       int clientHeight =
       System.Convert.ToInt32(sizes[1]);
        g.Context.Items["__ClientWidth"] =
        clientWidth;
        g.Context.Items["__ClientHeight"] =
        clientHeight;
    }
}
```

```
#region Implementacja modułu IHttpModule

public void Init(System.Web.HttpApplication context)
{
    context.BeginRequest +=new EventHandler(BeginRequest);
    string html = @"
    <html>
    Ładowanie...
```

Rysunek 8.19. *Instrukcja #region powoduje dodanie obszaru rozwijanego kodu. Kiedy edytor wykryje dyrektywę #region, umieszcza znak minus po lewej stronie okna kodu*

```
Implementacja modułu IHttpModule
void BeginRequest(object sender, EventArgs args)...
    }
}
```

Rysunek 8.20. *Kliknięcie znaku minus powoduje rozwinięcie kodu obszaru i wyświetlenie opisu za słowem #region, umożliwiającego identyfikację kodu. Słowo kluczowe #region w kodzie nie ma żadnego znaczenia funkcjonalnego. Jest ono wykorzystywane wyłącznie przez edytor, zatem, jeżeli komuś wydaje się niepotrzebne, może je usunąć*

Szczegóły zdarzenia BeginRequest

W pierwszym wierszu metody `BeginRequest` znajduje się instrukcja konwersji obiektu `sender` (pierwszy parametr metody) na obiekt `Global`. `Global` jest klasą generowaną przez kreator podczas tworzenia projektu ASP .NET. Jest to pochodna klasy `HttpApplication`. Obiekt tej klasy tworzy się za pierwszym razem, gdy dowolny klient skorzysta z dowolnej strony w aplikacji. Z obiektu `Global` można uzyskać dostęp do innych obiektów — np. `Response` lub `Request`.

W drugim wierszu kodu użyto obiektu `Global` do uzyskania obiektu `Request`. Jak pamiętamy, obiekt `Request` umożliwia uzyskanie informacji na temat żądania klienta. Jeżeli skorzystamy z właściwości `Form`, możemy uzyskać dostęp do ukrytego pola, które zostało utworzone przez skrypt działający po stronie klienta. Nazwa tego ukrytego pola to `txtDimensions`.

W wierszu `g.Request.Form["txtDimensions"]` następuje pobranie tekstu z ukrytego pola. Jeżeli pole nie istnieje, zwrócony wynik będzie wartością `null`. W przeciwnym razie będą to współrzędne obszaru klienta. Skrypt działający po stronie klienta zapisuje współrzędne w postaci: *szerokość;wysokość* (np. 800;600). Jeżeli wynik wynosi `null`, do klienta wysyłany jest skrypt za pomocą funkcji `Response.Write`. Potem następuje wywołanie funkcji `CompleteRequest`, której działanie polega na zatrzymaniu żądania strony. Jeżeli wymiary są dostępne, następuje rozbicie ciągu znaków o formacie *szerokość;wysokość* na dwie liczby i konwersja tych liczb na typ `integer`. Wartości te są następnie umieszczone w obiekcie `Items`.

Do tej pory nie używaliśmy obiektu `Items`, ale przypomina on obiekty `Session` oraz `Application`. Ogólnie rzecz biorąc, jest to jeszcze jeden sposób utrwalania informacji. Różnica między tymi obiektami polega na czasie dostępności informacji. W obiekcie `Items` informacje są dostępne tylko na czas trwania żądania klienta. W przypadku obiektu `Application` informacje są dostępne tak długo, jak serwer WWW przetwarza aplikację i mają do nich dostęp wszystkie klienty korzystające z aplikacji. Informacje zapisane w obiekcie `Session` są dostępne na czas trwania programu, ale są one specyficzne dla każdego klienta. Obiekt `Items` istnieje tylko na czas żądania. Jeżeli zatem klient przechodzi z jednej strony na drugą, informacje zostają utracone. W ramach tego samego żądania moduł może jednak umieścić informacje w obiekcie `Items` i strona będzie miała do nich dostęp przez odwołanie `g.Context.Items`.

Wykorzystanie obiektów poprzez interfejsy

Kiedy programista zdefiniuje interfejs i zaimplementuje go jako konkretną klasę, może używać tej klasy poprzez interfejs.

Aby używać klasy poprzez interfejs:

1. Zdefiniuj zmienną typu interfejsu. Wpisz na przykład `ICar var`.
2. Ustaw zmienną typu interfejsu na wartość równą klasie, która ten interfejs implementuje. Wpisz na przykład `= new Escort()` (rysunek 8.21).

Wskazówka

- Nie można tworzyć egzemplarzy interfejsu. Interfejs jest typem abstrakcyjnym, co oznacza, że nie jest to klasa, której obiekty można utworzyć. Zamiast tego tworzy się egzemplarze klasy implementującej interfejs.

Rysunek 8.21. *Chociaż typem zmiennej jest ICar, nie można napisać new ICar. Trzeba utworzyć egzemplarz klasy implementującej interfejs. Nie można tworzyć egzemplarzy interfejsu*

Kod

```
interface ICar
{
   string Run();
   string Stop();
   string PlayMusic();
}

class Escort : ICar
{
   public string Run()
   {
      return "Jadę tak szybko, jak umiem!";
   }
   public string Stop()
   {
      return "Jak dobrze. Wreszcie się mogę
      zatrzymać";
   }
   public string PlayMusic()
   {
      return "Jesteśmy piratami, którzy nic
      nie robią...";
   }
}

class Driver
{
   public string GoToWork()
   {
      ICar car = new Escort();
      string msg = "";
      msg+= car.Run();
      msg+= car.PlayMusic();
      msg+= car.Stop();
      return msg;
   }
}
```

Rysunek 8.22. *Zmienna obj jest typu object. Oznacza to, że może wskazywać na obiekt Cat lub na obiekt Dog. Powyższy kod testuje zawartość zmiennej obj przez sprawdzenie, czy obiekt obj obsługuje interfejs IDog*

Kod

```
interface ICat
{
   string IgnoreOwner();
}

interface IDog
{
   string ListenToOwner();
}

class Poodle : IDog
{
}

class Siamese : ICat
{
}

class Owner
{
   void FeedAnimal(object obj)
   {
      if (obj is IDog)
      {
         IDog dog = (IDog)obj;
         string cmd = dog.ListenToOwner();
      }
   }
}
```

Rozpoznawanie interfejsu

Przed użyciem obiektu poprzez interfejs warto sprawdzić, czy obiekt rzeczywiście obsługuje interfejs. Oczywiście, jeżeli mamy dostęp do definicji klasy, możemy na nią spojrzeć i przekonać się, czy klasa implementuje interfejs. Czasami jednak mamy do czynienia z danymi typu `object`, dla których trzeba zweryfikować, czy obiekt, na który wskazuje zmienna, jest zgodny z interfejsem, którego chcemy użyć. Istnieją dwa sposoby sprawdzenia, czy obiekt obsługuje interfejs.

Aby sprawdzić, czy obiekt obsługuje interfejs:

1. Wpisz `if (obj is IAnything)`, gdzie `obj` jest zmienną wskazującą na obiekt, który chcesz sprawdzić, natomiast `IAnything` jest nazwą interfejsu, dla którego chcesz przeprowadzić test (rysunek 8.22).

   lub

   Wpisz `IAnything var = obj as IAnything`, gdzie `IAnything` jest interfejsem, dla którego chcesz przeprowadzić test, `var` jest dowolną zmienną służącą do przechowywania odwołania do obiektu, a `obj` jest obiektem, który chcesz sprawdzić.

2. Wpisz instrukcję `if (var!=null)`. Jeżeli po wykonaniu pierwszego kroku zmienna `var` jest równa *null*, obiekt nie obsługuje interfejsu. Jeśli test zwróci wartość różną od *null*, obiekt obsługuje interfejs, a zmienna `var` wskazuje na odwołanie do interfejsu (rysunek 8.23).

Wskazówki

- Podczas korzystania z pierwszego mechanizmu, po uzyskaniu informacji, że obiekt obsługuje interfejs, aby użyć obiektu poprzez interfejs, należy zadeklarować zmienną typu interfejsu, po czym dokonać konwersji obiektu na interfejs.
- W przypadku drugiego mechanizmu rozpoznawania interfejsu, zarówno konwersja obiektu, jak i rozpoznanie następuje w jednym kroku. Jeżeli obiekt obsługuje interfejs, zmienna `var` będzie wskazywać na obiekt, w przeciwnym razie zmienna `var` przyjmie wartość *null*.
- Istnieje trzeci mechanizm rozpoznawania, czy obiekt obsługuje interfejs. Można spróbować dokonać konwersji obiektu na interfejs. Jeżeli obiekt nie obsługuje interfejsu, konwersja spowoduje zgłoszenie wyjątku (błędu). Wyjątki zostaną omówione w rozdziale 11. „Obsługa błędów". Wykorzystywanie wyjątków nie jest zalecane jako sposób rozpoznawania obsługi interfejsu przez obiekty, ponieważ wyjątki mogą wpłynąć niekorzystnie na wydajność działania programu.

Rysunek 8.23. *Kiedy do sprawdzenia, czy obiekt obsługuje interfejs, użyjemy polecenia as, uzyskamy wynik null wtedy, gdy obiekt go nie obsługuje*

Kod

```
interface ICat
{
   string IgnoreOwner();
}

interface IDog
{
   string ListenToOwner();
}

class Poodle : IDog
{
}

class Siamese : ICat
{
}

class Owner
{
   void FeedAnimal(object obj)
   {
       IDog dog = obj as IDog;
       if (dog != null)
         string cmd = dog.ListenToOwner();
   }
}
```

Rysunek 8.24. *Zwróć uwagę, że typem parametru metody Communicate jest IHuman. Do metody można przekazać dowolny obiekt, który implementuje interfejs. Każda z trzech klas implementuje interfejs w inny sposób. Metoda Communicate zwraca inny ciąg znaków w zależności od przekazanego obiektu*

Kod

```
interface Ihuman
{
   string Speak();
}

class Baby : IHuman
{
   public string Speak()
   {
      return "Goo-goo gaa-gaa";
   }
}

class Spouse : IHuman
{
   public string Speak()
   {
      return "Czy mogę kupić nowy komputer?";
   }
}

class Friend : IHuman
{
   public string Speak()
   {
      return "Czy mógłbyś pożyczyć mi trochę
      pieniędzy?";
   }
}

class Person
{
   string Communicate(IHuman person)
   {
      return person.Speak();
   }
   void DailyLiving()
   {
      string answer;

      Spouse sp = new Spouse();
      answer = Communicate(sp);
      Friend fr = new Friend();
      answer = Communicate(fr);
      Baby ba = new Baby();
      answer = Communicate(ba);
   }
}
```

Wykorzystanie interfejsów do polimorfizmu

Polimorfizm to mechanizm polegający na tym, że dwie powiązane ze sobą klasy mają nieznacznie różniące się implementacje tej samej metody. W przypadku wykorzystania interfejsów jako typów, programista może zdefiniować funkcje, które akceptują jako parametry dowolne obiekty implementujące interfejs. Następnie, w zależności od obiektu przesłanego do funkcji, wykonywany jest nieco inny kod.

Aby wykorzystać polimorfizm:

1. W dowolnej klasie zdefiniuj metodę, w której jeden z parametrów wejściowych jest typu interfejsu.
2. Wywołaj metody interfejsu.
3. Przekaż do metody dowolny obiekt obsługujący interfejs (rysunek 8.24).

Wskazówka

- Przed przekazaniem obiektu do metody należy się upewnić, czy obiekt rzeczywiście obsługuje interfejs (więcej informacji na ten temat znalazło się w podrozdziale „Rozpoznawanie interfejsu” we wcześniejszej części tego rozdziału).

Interfejs będący pochodną innego interfejsu

Jeżeli inni programiści wykorzystują nasz obiekt poprzez określony interfejs, a niezbędne okaże się ulepszenie interfejsu, wówczas najlepiej pozostawić definicję oryginalnego interfejsu bez zmian. W języku C# istnieje mechanizm umożliwiający rozszerzanie interfejsu bez dodawania metod do interfejsu oryginalnego. Działanie tego mechanizmu polega na tworzeniu interfejsów pochodnych.

Aby utworzyć interfejs będący pochodną innego interfejsu:

1. Zakładając, że wcześniej zdefiniowałeś jeden interfejs, zdefiniuj drugi.
2. Za nazwą interfejsu wpisz dwukropek, następnie `NameOfFirstInterface`, gdzie `NameOfFirstInterface` jest nazwą interfejsu, który chcesz rozszerzyć.
3. Oba interfejsy: oryginalny i rozszerzony możesz zaimplementować w ramach tej samej klasy (rysunek 8.25).

Rysunek 8.25. *Interfejs IPresident stanowi rozszerzenie interfejsu IParent. Oznacza to, że przy implementacji interfejsu IPresident należy zaimplementować nie tylko metody interfejsu IPresident, ale także metody interfejsu IParent*

Kod

```
interface IParent
{
   void SendKidsToPlay();
}

interface IPresident : IParent
{
   void DispatchArmy();
}

class George : IPresident
{
   //implementacja metod dla obu interfejsów
   //Iparent oraz IPresident
   public void SendKidsToPlay()
   {
   }

   public void DispatchArmy()
   {
   }
}
```

Rysunek 8.26. *Zaimplementowanie obu interfejsów: nadrzędnego i pochodnego jest tym samym, co zaimplementowanie tylko interfejsu pochodnego, ponieważ interfejs pochodny zawiera wszystkie składowe interfejsu nadrzędnego*

Kod

```
interface IParent
{
   void SendKidsToPlay();
}

interface IPresident : IParent
{
   void DispatchArmy();
}

class George : IPresident
{
   //implementacja metod dla obu interfejsów:
   //Iparent oraz IPresident
   public void SendKidsToPlay()
   {
   }

   public void DispatchArmy()
   {
   }
}
```

Rysunek 8.27. *Mimo że klasa Telemarketer implementuje zarówno interfejs IPerson, jak ISendGifts, klasa ta nie jest zgodna z interfejsem IGivingPerson*

Kod

```
interface IGivingPerson : IPerson,
ISendsGifts
{
}

class Grandma : IGivingPerson
{
}
class Telemarketer : IPerson, ISendsGifts
{
}
```

Wskazówki

- Definicje klas zaprezentowanych na rysunku 8.26 są sobie równoważne. Zaimplementowanie interfejsu oryginalnego i pochodnego jest tym samym, co zaimplementowanie tylko interfejsu pochodnego. W obu przypadkach uzyskuje się klasę zgodną z obydwoma interfejsami.
- Definicje klas przedstawionych na rysunku 8.27 nie są sobie równoważne. Chociaż interfejs `IGivingPerson` jest kombinacją interfejsów: `IPerson` oraz `ISendsGifts`, to zaimplementowanie interfejsu `IGivingPerson` nie jest tym samym, co oddzielne zaimplementowanie interfejsów `IPerson` i `ISendsGiftf`. Interfejs `IGivingPerson` może zawierać inne metody niż kombinacja metod interfejsów: `IPerson` oraz `ISendsGifts`.

Refaktoryzacja

Refaktoryzacja (ang. *refactoring*) to mechanizm wykorzystywany w programowaniu obiektowym do działań z klasami i interfejsami. Polega on na tym, że jeśli dwie klasy spełniające podobne funkcje mają wspólny kod, wówczas ze wspólnego kodu tworzy się klasę bazową, a następnie, na podstawie tej klasy bazowej, tworzy się podklasy. Podklasy zawierają tylko ten kod, który je od siebie odróżnia[1].

Aby dokonać refaktoryzacji klas z wykorzystaniem interfejsów:

1. Jeżeli kilka klas implementuje ten sam interfejs w podobny sposób (kod implementacyjny jest identyczny dla wszystkich klas), utwórz klasę bazową.
2. Zaimplementuj interfejs w klasie bazowej.
3. Wprowadź w klasie bazowej kod implementacyjny dla każdej metody interfejsu.
4. Utwórz klasy pochodne na podstawie klasy bazowej.
5. Teraz możesz wykorzystać klasy pochodne poprzez interfejs (rysunek 8.28).

Wskazówki

- Jeżeli klasa bazowa implementuje interfejs, klasy pochodne są także zgodne z interfejsem.
- W czasie projektowania klas, najpierw należy zdefiniować interfejsy, później utworzyć klasę bazową implementującą te interfejsy, a następnie utworzyć podklasy dziedziczące cechy klasy bazowej. Klasy pochodne wykorzystuje się poprzez interfejs. Właśnie taki mechanizm często wykorzystują doświadczeni programiści podczas pisania aplikacji.

Rysunek 8.28. *Obiektu Checking można użyć poprzez interfejs IAccount, ponieważ klasa Checking jest pochodną klasy AccountImpl, a klasa AccountImpl implementuje interfejs IAccount. Innymi słowy, jeżeli dowolna klasa w hierarchii nadrzędnej implementuje interfejs, to klasa pochodna także obsługuje ten interfejs*

Kod

```
interface IAccount
{
   void MakeDeposit(int Amount);
   void MakeWithdrawal(int Amount);
}

class AccountImpl : IAccount
{
   public void MakeDeposit(int Amount)
   {
   }

   public void MakeWithdrawal(int Amount)
   {
   }
}

class Checking : AccountImpl
{
}

class Savings : AccountImpl
{
}
class Bank
{
   void OpenAccount()
   {
      IAccount acct = new Checking();
      acct.MakeDeposit(100);
   }
}
```

[1] Z treści tego akapitu nie wynika, czym naprawdę jest refaktoryzacja. Jest to po prostu upraszczanie kodu, czyli między innymi wydzielanie podklas — *przyp. tłum.*

Rysunek 8.29. *Klasa Dog implementuje interfejs IAnimal. Klasa GreatDane jest pochodną klasy Dog, a zatem przejmuje implementację interfejsu IAnimal. Jednak co zrobić, aby zaimplementować metodę Speak interfejsu IAnimal inaczej niż w implementacji Dog, a implementację metody Eat pozostawić bez zmian? W takiej sytuacji można ponownie zaimplementować interfejs IAnimal w klasie GreatDane i zmienić implementację wybranych metod*

Kod

```
interface IAnimal
{
   string Speak();
   string Eat();
}

class Dog : IAnimal
{
   string IAnimal.Speak()
   {
      return "woof! woof!";
   }

   string IAnimal.Eat()
   {
      return "Yum!";
   }
}

class GreatDane : Dog , IAnimal
{
   string IAnimal.Speak()
   {
      return "Wielkie woof! Wielkie woof!";
   }
}
```

Ponowna implementacja interfejsów w klasie pochodnej

Czytelnicy, którzy przeczytali wcześniejszy podrozdział o refaktoryzacji, wiedzą o tym, że można utworzyć klasę bazową obsługującą interfejs, a następnie napisać klasę pochodną dla tej klasy bazowej. Klasa pochodna również obsługuje interfejs. Czasami jednak trzeba przesłonić jedną metodę lub kilka metod implementacyjnych w klasie bazowej. W tym celu można ponownie zaimplementować interfejs w klasie pochodnej.

Aby ponownie zaimplementować interfejs w klasie pochodnej:

1. Za nazwą klasy pochodnej wpisz dwukropek, a po nim nazwę interfejsu, który chcesz zaimplementować ponownie.
2. Wprowadź implementację tylko dla tych metod, które mają być różne od metod klasy bazowej (rysunek 8.29).

Wskazówka

- Innym sposobem ponownej implementacji metody interfejsu jest zaznaczenie oryginalnej metody implementacyjnej jako wirtualnej, a następnie przesłonięcie jej w podklasie (rysunek 8.30).

Rysunek 8.30. *Ta metoda wymaga od autora klasy bazowej zaznaczenia metody jako wirtualnej, co nie zawsze jest możliwe*

Kod

```
interface IAnimal
{
   string Speak();
   string Eat();
}

class Dog : IAnimal
{
   public virtual string Speak()
   {
      return "woof! woof!";
   }

   public string Eat()
   {
      return "Yum!";
   }
}

class GreatDane : Dog
{
   public override string Speak()
   {
      return "WOOF! WOOF!";
   }
}
```

Rysunek 8.31. *Środowisko ASP .NET przed rozpoczęciem aplikacji odczytuje ustawienia w pliku web.config. Ustawienia w tym pliku można wykorzystać w celu zarządzania sposobem uruchamiania aplikacji przez system ASP .NET*

Kod

```
<httpModules>
   <add name="Dimensions"
   type="interfacesproject.custommodule,
   interfacesproject"/>
</httpModules>
```

Kończymy przykładową aplikację

Pozostało jedynie kilka czynności, które trzeba wykonać, aby nasza aplikacja stała się w pełni funkcjonalna. W tym podrozdziale uzupełnimy pozostały kod.

Najpierw upewnimy się, że środowisko ASP .NET wie o module użytkownika.

Aby uaktywnić moduł użytkownika w aplikacji:

1. W module *Solution Explorer* kliknij dwukrotnie plik *web.config*.
2. Przewiń plik do końca i wpisz kod pokazany na rysunku 8.31 zaraz za wierszem `</system.web>`.

To wszystko, co trzeba zrobić, aby uaktywnić moduł użytkownika dla naszej aplikacji. Następny krok polega na wprowadzeniu kodu na stronie *dimensions.aspx*, aby upewnić się, czy moduł działa.

Aby zakończyć stronę dimensions.aspx:

1. Dwukrotnie kliknij pozycję *dimension.aspx* w oknie *Solution Explorer*.
2. Dwukrotnie kliknij pusty obszar na formularzu, co spowoduje wywołanie edytora kodu. Kreator doda zdarzenie `Page_Load`.
3. Wewnątrz metody `Page_Load` wprowadź kod zaprezentowany na rysunku 8.32. Kod ten spowoduje wyświetlenie wartości szerokości i wysokości w dwóch polach tekstowych (rysunek 8.33).

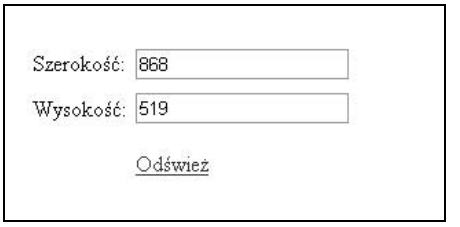
Wskazówki

- Aby przekonać się, że moduł działa, wystarczy zmienić rozmiar okna przeglądarki, a następnie kliknąć przycisk *Odśwież* znajdujący się pod polami tekstowymi. W polach tekstowych powinny pojawić się nowe rozmiary okna.
- Dzisiejszej nocy warto się dobrze wyspać.

Rysunek 8.32. *W tym kodzie sprawdzana jest kolekcja Items i pobierane wartości __ClientWidth oraz __ClientHeight zapisane przez moduł użytkownika*

Kod

```
private void Page_Load(object sender,
System.EventArgs e)
{
   txtWidth.Text =
   ((int)Context.Items["__ClientWidth"]).
   ➥ToString();
   txtHeight.Text =
   ((int)Context.Items["__ClientHeight"]).
   ➥ToString();
}
```



Rysunek 8.33. *Aby przetestować moduł użytkownika, wystarczy zmienić rozmiary okna przeglądarki, a następnie kliknąć łącze Odśwież. Wartości szerokości i wysokości okna powinny ulec zmianie*