.NET. Najpilniej strzeżone tajemnice

Autor: Deborah Kurata
Tłumaczenie: Grzegorz Werner
ISBN: 83-246-0173-2
Tytuł oryginału: Best Kept Secrets In .Net
Format: B5, stron: 192

Platforma .NET staje się coraz popularniejsza. Tysiące programistów na całym świecie doceniają jej doskonałe narzędzia programistyczne i tysiące stron wyczerpującej dokumentacji. Każdego dnia adepci sztuki programowania odkrywają nowe możliwości oferowane im przez .NET. Wiele z tych możliwości pozostaje jednak wciąż nie odkrytych. Sztuczki i furtki zaszyte przez twórców platformy .NET pozwalające na szybszą i sprawniejszą realizację projektów czekają na to, aby ktoś je zastosował w pracy. Jak dotrzeć do tych ukrytych cech? Skąd wziąć informacje o tym, co jeszcze można osiągnąć wykorzystują znane już narzędzia, na temat których napisano setki książek?

Dzięki książce „.NET. Najpilniej strzeżone tajemnice" poznasz te możliwości platformy .NET, których nie opisuje oficjalna dokumentacja. Znajdziesz w niej omówienie niezwykle cennych, ale mniej znanych funkcji Visual Studio i .NET Framework. Każda z tych funkcji może pomóc Ci podnieść wydajność procesu tworzenia oprogramowania oraz poprawić jakość swoich aplikacji. Dowiesz się, jak wykorzystać maksimum możliwości środowiska programistycznego, wykorzystasz nieznane funkcje GDI+ i ADO.NET i nauczysz się zasad programowania defensywnego, dzięki któremu znacznie ograniczysz liczbę błędów w swoich programach.

- Zarządzanie wyrywkami kodu w Visual Studio
- Dostosowywanie skrótów klawiaturowych
- Praca z kontrolkami Windows Forms
- Udoskonalone techniki rzutowania
- Stosowanie komentarzy XML
- Korzystanie z komponentu Microsoft Data Access Application Block
- Wyświetlanie zestawów danych w formacie XML
- Zapobieganie nieautoryzowanemu dostępowi do aplikacji
- Przeprowadzanie testów jednostkowych

Wykorzystaj sekrety i skarby ukryte w platformie .NET

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

Spis treści

Rozdział 3.

Tajniki kodowania

Jeśli programujesz już od jakiegoś czasu, zapewne wielokrotnie wykorzystujesz te same wzorce kodowania. Na przykład kod, który przetwarza zdarzenia kontrolki `Toolbar`, zwykle zawiera instrukcję `Case`; procedury zdarzeniowe zawierają standardowy blok `Try Catch` itd. Łatwo popaść w rutynę i koncentrować się na wykonywaniu bieżących zadań do tego stopnia, że brakuje czasu na wypróbowanie innych technik kodowania.

Celem tego rozdziału jest przedstawienie kilku trików, dzięki którym kodowanie staje się łatwiejsze, produktywniejsze — po prostu lepsze. Do rozdziału dodano opis kilku technik debugowania — bo po co komu choćby najefektowniejszy, ale błędnie działający kod?

Co zostanie opisane w tym rozdziale?

W rozdziale tym przedstawione zostaną następujące tajniki kodowania:

- warunkowe operatory logiczne,
- skrócone operatory przypisania,
- zarządzanie łańcuchami przy użyciu klasy `StringBuilder`,
- deklarowanie zmiennych w pętli,
- ścisła konwersja typów danych,
- ulepszone rzutowanie,
- aliasy typów danych,
- wyrażenia regularne,
- przeciążanie procedur,
- przeciążanie operatorów,
- eksploracja nieodkrytych regionów,

- komentarze XML,
- oznaczanie przestarzałego kodu,
- rozwijanie technik debugowania.

Po przeczytaniu niniejszego rozdziału będziesz umiał zadziwić kolegów nowymi technikami kodowania. Przekonasz się, że techniki te zwiększą Twoją produktywność i ułatwią pisanie oraz konserwację kodu.

Warunkowe operatory logiczne

Operatory logiczne porównują dwa wyrażenia (typu prawda — fałsz) i zwracają wynik logiczny. Najczęściej używanymi operatorami logicznymi są iloczyn (`And`) i suma (`Or`). Aby wykonać operację w sytuacji, w której spełnione są dwa warunki, użyj operatora iloczynu logicznego (jeśli **A** jest prawdziwe **oraz B** jest prawdziwe, to…). Aby wykonać operację, gdy spełniony jest przynajmniej jeden spośród dwóch warunków, posłuż się operatorem sumy logicznej (jeśli **A** jest prawdziwe **lub B** jest prawdziwe, to…).

Ściśle mówiąc, operator sumy logicznej bada dwa wyrażenia logiczne. Jeśli którekolwiek z nich jest prawdziwe (ma wartość `True`), operator zwraca `True`. Jeśli żadne z wyrażeń nie ma wartości `True`, operator zwraca `False`. Oto prosty przykład użycia operatora sumy logicznej: jeżeli miejsce przeznaczone na nazwę użytkownika **lub** hasło jest puste, kod powinien wyświetlić odpowiedni komunikat.

W Visual Basicu operatorem sumy logicznej jest słowo `Or`:

```
If sUserName = "" Or sPassword = "" Then
    MessageBox.Show("Musisz wprowadzić nazwę użytkownika i hasło")
End If
```

W C# operatorem sumy logicznej jest znak `|`:

```
if (sUserName == "" | sPassword == "")
{
    MessageBox.Show("Musisz wprowadzić nazwę użytkownika i hasło");
}
```

Operator iloczynu logicznego również bada dwa wyrażenia logiczne. W przypadku gdy oba są prawdziwe (mają wartość `True`), operator zwraca wartość `True`. Jeśli jednak jedno wyrażenie jest fałszywe albo też fałszywe są obydwa wyrażenia (mają wartość `False`), operator zwraca wartość `False`. Przykład użycia operatora iloczynu logicznego? — jeżeli kolekcja uporządkowanych elementów nie jest pusta **oraz** liczba elementów jest większa niż zero, można przypisać liczbę właściwości `Text` pola tekstowego.

W Visual Basicu operatorem iloczynu logicznego jest słowo `And`:

```
With  m_dsOrder.Tables(Order.TN_OrderHeader).Rows(0)
    If Not .IsNull(Order.FN_ITEM_COUNT) And _
        CType(.Item(Order.FN_ITEM_COUNT), Int32) > 0 Then
```

```
            txtItemCount.Text = .Item(Order.FN_ITEM_COUNT).ToString
        End If
    End With
```

W C# operatorem iloczynu logicznego jest znak &:

```
DataRow dr = m_dsOrder.Tables[Order.TN_OrderHeader].Rows[0];
If (!dr.IsNull(Order.FN_ITEM_COUNT) &
    System.Convert.ToInt32(dr[Order.FN_ITEM_COUNT]) > 0)
{
    txtItemCount.Text = dr[Order.FN_ITEM_COUNT].ToString();
}
```

Ups! Powyższy kod wygeneruje wyjątek, jeśli pole liczby elementów będzie puste. Zarówno w C#, jak i w VB operatory sumy oraz iloczynu logicznego wyznaczają wartość obu wyrażeń. Jeżeli nawet pole liczby elementów jest puste i pierwsze wyrażenie ma wartość `False`, zostanie zbadane również drugie wyrażenie, co spowoduje zgłoszenie wyjątku błędnego rzutowania (nie można rzutować wartości pustej na liczbę całkowitą).

Aby zapobiec temu problemowi, należy skorzystać z warunkowych operatorów logicznych, nazywanych również operatorami „zwierającymi". Operatory te najpierw badają wartość pierwszego wyrażenia, a następnie warunkowo wyznaczają wartość drugiego.

W Visual Basicu operatorami „zwierającymi" są słowa `AndAlso` oraz `OrElse`. W tym przykładzie użyłbyś operatora `AndAlso` zamiast `And`, żeby zapobiec badaniu drugiego wyrażenia, jeśli pierwsze jest prawdziwe:

```
With  m_dsOrder.Tables(Order.TN_OrderHeader).Rows(0)
    If Not .IsNull(Order.FN_ITEM_COUNT) AndAlso _
        CType(.Item(Order.FN_ITEM_COUNT), Int32) > 0 Then
        txtItemCount.Text = .Item(Order.FN_ITEM_COUNT).ToString
    End If
End With
```

W C# operatorami „zwierającymi" są znaki && oraz ||. W zaprezentowanym przykładzie używa się operatora && zamiast & i zapobiega badaniu drugiego wyrażenia, jeśli pierwsze jest prawdziwe:

```
DataRow dr = m_dsOrder.Tables[Order.TN_OrderHeader].Rows[0];
If (!dr.IsNull(Order.FN_ITEM_COUNT) &&
    System.Convert.ToInt32(dr[Order.FN_ITEM_COUNT]) > 0)
{
    txtItemCount.Text = dr[Order.FN_ITEM_COUNT].ToString();
}
```

Jest to pozornie niewielka zmiana (`And` na `AndAlso` lub & na &&), ale oferuje wiele korzyści. Nie tylko zwiększa wydajność, lecz także może zapobiegać błędom, takim jak nieprawidłowe rzutowanie.

Operatory „zwierające" są warunkowymi wersjami operatorów logicznych. Wyznaczają wartość drugiego wyrażenia w następujących przypadkach:

- `x AndAlso y` zawsze wyznacza wartość wyrażenia `x`, a wartość `y` bada jedynie wtedy, gdy `x` jest prawdziwe (Visual Basic);

- `x OrElse y` zawsze wyznacza wartość wyrażenia `x`, a wartość `y` bada jedynie wtedy, gdy `x` jest fałszywe (Visual Basic);
- `x && y` zawsze wyznacza wartość wyrażenia `x`, a wartość `y` bada jedynie wtedy, gdy `x` jest prawdziwe (C#);
- `x || y` zawsze wyznacza wartość wyrażenia `x`, a wartość `y` bada jedynie wtedy, gdy `x` jest fałszywe (C#).

Chcąc zwiększyć wydajność swoich aplikacji, powinieneś używać operatorów „zwierających" wszędzie, gdzie to możliwe. Operatory te skracają czas wykonywania programu, ponieważ mogą pomijać badanie drugiego wyrażenia w zależności od wartości pierwszego.

Przejrzyj swoje programy i zastąp operatory logiczne „zwierającymi" wszędzie, gdzie to możliwe.

Ponieważ operatory „zwierające" pomijają badanie drugiego wyrażenia, nie używaj ich w przypadkach, gdy drugie wyrażenie wywołuje funkcję, która musi zostać wykonana bez względu na wartość pierwszego wyrażenia.

Operatory „zwierające" są szczególnie przydatne podczas pracy z polami bazy danych, gdyż w jednym wyrażeniu warunkowym możesz sprawdzić, czy pole nie jest puste, i zbadać, czy ma konkretną wartość. Pamiętaj o tym triku za każdym razem, gdy będziesz pisał `And`, `Or`, `&` lub `|`.

Skrócone operatory przypisania

Zarówno w VB, jak i w C# operatorem przypisania jest znak równości (=). Znaku tego używasz wtedy, gdy chcesz przypisać zmiennej wartość pewnego wyrażenia. Aby na przykład obliczyć łączną liczbę produktów w zamówieniu, dodawałbyś kolejno zamawiane produkty do ogólnej sumy.

Visual Basic:

```
iOrderItemTotal = iOrderItemTotal + iQuantity
```

C#:

```
iOrderItemTotal = iOrderItemTotal + iQuantity;
```

Możesz jednak oszczędzić sobie pisania, używając skróconej wersji operatora przypisania: +=. Operator ten dodaje wartość wyrażenia do wartości zmiennej i przypisuje wynik tej samej zmiennej.

Visual Basic:

```
iOrderItemTotal += iQuantity
```

C#:

```
iOrderItemTotal += iQuantity;
```

Najczęściej używane skrócone operatory przypisania zgromadzono w tabeli 3.1.

Tabela 3.1. *Skrócone operatory przypisania*

Operator	Opis i przykład	Uwagi
`+=`	Dodaje wartość wyrażenia do wartości zmiennej i przypisuje wynik tej samej zmiennej. `x += 4` Zwiększa wartość zmiennej `x` o 4 i przypisuje wynik zmiennej `x`.	W C# operator `+=` łączy również wyrażenia łańcuchowe.
`-=`	Odejmuje wartość wyrażenia od wartości zmiennej i przypisuje wynik tej samej zmiennej. `x -= 3` Zmniejsza wartość zmiennej `x` o 3 i przypisuje wynik zmiennej `x`.	
`*=`	Mnoży wartość wyrażenia przez wartość zmiennej i przypisuje wynik tej samej zmiennej. `x *= 2` Mnoży zmienną `x` przez 2 i przypisuje wynik zmiennej `x`.	
`/=`	Dzieli wartość zmiennej przez wartość wyrażenia i przypisuje wynik tej samej zmiennej. `x /= 4` Dzieli zmienną `x` przez 4 i przypisuje wynik zmiennej `x`.	W VB, jeśli opcja `Option Strict` jest ustawiona na `On`, zmienna po lewej stronie (w tym przypadku `x`) musi być zadeklarowana jako `Double`. Jeśli opcja `Option Strict` jest ustawiona na `Off`, zmienna po lewej stronie zostanie niejawnie przekształcona w typ `Double`.
`\=`	Dzieli wartość zmiennej przez wartość wyrażenia i przypisuje zmiennej **część całkowitą** wyniku. `x \= 4` Dzieli zmienną `x` przez 4 i przypisuje wynik zmiennej `x`, odrzucając resztę z dzielenia.	Tylko VB. Jeśli opcja `Option Strict` jest ustawiona na `On`, zmienna musi być typu `Byte`, `Short`, `Integer` lub `Long`. Jeśli opcja `Option Strict` jest ustawiona na `Off`, zmienna zostanie przekształcona w typ `Long`.
`&=`	Łączy wyrażenie łańcuchowe ze zmienną łańcuchową i przypisuje wynik tej samej zmiennej. `x &= "Świecie"` Dołącza słowo „świecie” do łańcucha przechowywanego w zmiennej `x` i przypisuje wynik zmiennej `x`.	Tylko VB. W C# do łączenia łańcuchów używa się operatora `+=`. W C# operator `&=` przypisuje zmiennej wynik iloczynu logicznego. Więcej informacji można znaleźć w pomocy Visual Studio.

Więcej informacji o niejawnych konwersjach oraz opcji `Option Strict` znajdziesz w dalszym podrozdziale „Ścisła konwersja typów danych”.

Skrócone operatory przypisania oszczędzają pisania i są efektywniejsze, ponieważ zmienna (`x` w przykładach z tabeli 3.1) jest szacowana tylko raz.

Zarządzanie łańcuchami przy użyciu klasy StringBuilder

Łańcuchy są **niezmienne**. Innymi słowy, raz przypisanego łańcucha nie można zmienić. Kiedy dołączasz jeden łańcuch do drugiego, Visual Studio w rzeczywistości tworzy nowy łańcuch złożony z pierwotnego i dołączonego. Jeśli przypuszczasz, że opóźnia to wykonywanie programu, to masz rację.

Łączenie łańcuchów jest zoptymalizowane, więc ograniczone operacje łączenia nie mają zauważalnego wpływu na wydajność aplikacji. Wpływ ten staje się odczuwalny, gdy łączysz wiele łańcuchów, na przykład podczas budowania strony ASP.NET albo treści komunikatu.

Visual Basic:

```
Dim sMessage As String = "To jest komunikat"
sMessage &= " dla użytkownika "
sMessage &= sUserName
```

C#:

```
string Message =  "To jest komunikat";
sMessage += " dla użytkownika ";
sMessage += sUserName;
```

Wydawać by się mogło, że w powyższych przykładach tworzony jest łańcuch `sMessage`, do którego dołączane są kolejne łańcuchy, przez co zmienia się pierwotna zawartość łańcucha `sMessage`. W rzeczywistości dzieje się coś innego. W pierwszym wierszu przykładu tworzony jest łańcuch o zdefiniowanej treści („To jest komunikat”). Zarówno w drugim, jak i w trzecim wierszu łańcuch `sMessage` jest niszczony (oznaczany przez „odśmiecacz” jako przeznaczony do usunięcia), po czym tworzony jest nowy łańcuch `sMessage` złożony z pierwotnego i dołączonego.

Jeśli łączenie łańcuchów odbywa się w jednej instrukcji, łańcuch nie jest niszczony między kolejnymi dołączeniami, więc wydajność programu nie zmniejsza się.

Visual Basic:

```
Dim sMessage As String = "To jest komunikat" & " dla użytkownika " & sUserName
```

C#:

```
string sMessage = "To jest komunikat" + " dla użytkownika " + sUserName;
```

Jednakże podczas łączenia wielu długich łańcuchów albo budowania całej zawartości strony ASP.NET użycie pojedynczego przypisania może być niepraktyczne (albo nieczytelne).

Aby zminimalizować wpływ łączenia łańcuchów na wydajność aplikacji, możesz użyć klasy `StringBuilder`. Klasa ta tworzy obiekt przypominający łańcuch, który jednak może być modyfikowany. Kiedy dokonasz wszystkich żądanych zmian, możesz przekształcić zawartość obiektu `StringBuilder` w rzeczywisty łańcuch.

Visual Basic:

```
Imports System.Text
Dim sbMessage As StringBuilder
sbMessage = New StringBuilder("To jest komunikat")
sbMessage.Append(" dla użytkownika ")
sbMessage.Append(sUserName)
Dim sMessage As String = sbMessage.ToString
```

C#:

```
using System.Text;
StringBuilder sbMessage = new StringBuilder("To jest komunikat");
sbMessage.Append(" dla użytkownika ");
sbMessage.Append(sUserName);
string sMessage = sbMessage.ToString();
```

Ten przykładowy kod tworzy instancję klasy `StringBuilder`. Potem używa metody `Append` klasy `StringBuilder`, aby dołączyć kolejne łańcuchy tekstu. Po połączeniu wszystkich łańcuchów metoda `ToString` przekształca obiekt `StringBuilder` w typ `String`. Klasa `StringBuilder` zawiera inne metody, takie jak `Insert` i `Replace`, które przeprowadzają różne operacje na łańcuchu.

Klasa `StringBuilder` zapewnia efektywniejsze łączenie łańcuchów, ponieważ tworzy bufor na tyle duży, aby mógł pomieścić pierwotny łańcuch i zostawić miejsce na jego rozszerzanie. Jeśli jednak dołączane łańcuchy zajmą całe miejsce w buforze, trzeba będzie go powiększyć, a to spowoduje spadek wydajności.

Żeby jeszcze bardziej zwiększyć wydajność, użyj metody `EnsureCapacity` klasy `StringBuilder`. Metoda ta umożliwia ustawienie rozmiaru bufora `StringBuilder` na przewidywaną długość końcowego łańcucha. Obiekt `StringBuilder` nie będzie musiał powiększać bufora, co pozwoli uniknąć spadku wydajności.

Inicjalizowanie obiektu `StringBuilder` wymaga więcej czasu niż tworzenie łańcucha, więc nie ma sensu używać go w przypadku łączenia zaledwie kilku łańcuchów. Ogólnie mówiąc, punktem granicznym jest liczba pięć. Jeśli łączysz więcej niż pięć łańcuchów, `StringBuilder` zwykle bywa wydajniejszy.

W sytuacji, w której obsługa łańcuchów w programie musi być jak najwydajniesza, jedyny sposób sprawdzenia, czy szybsze jest łączenie łańcuchów, czy też użycie obiektu `StringBuilder`, to przeprowadzenie testów wydajności.

Jeśli kiedykolwiek zauważysz wiele łączeń łańcuchów w swoim kodzie, rozważ użycie klasy `StringBuilder` w celu uzyskania większej wydajności.

Deklarowanie zmiennych w pętli

Niegdyś większość podręczników kodowania zalecała deklarowanie wszystkich zmiennych metody na samym jej początku. Takie działanie może jednak powodować problemy w przypadku zmiennych, które są definiowane tylko na użytek pętli i używane w innym miejscu metody. Obecnie zaleca się deklarować zmienne pętli jak najbliżej miejsca, w którym zostaną użyte.

Najprostszym sposobem zadeklarowania zmiennej pętli jest zdefiniowanie jej w samym kodzie pętli. Możesz to zrobić w przypadku dowolnej pętli `For` lub `For Each`.

Oto przykład pętli `For Each` w Visual Basicu:

```
For Each ctrl As Control In pnl.Controls
    ...
Next
```

A oto pętla `for` w języku C#:

```
for (int i = 1; i <= 5; i++)
{
    sResponse = i + sCustomerName;
    ...
}
```

Deklarując te zmienne w konstrukcji pętli, zyskujesz pewność, że nie zostaną one przypadkowo użyte w innym miejscu metody.

Ścisła konwersja typów danych

Nie zawsze dane należą do typu, który jest potrzebny w programie. Przypuśćmy, że Twoja aplikacja pobiera liczbę całkowitą z bazy danych i ma wyświetlić ją w kontrolce `TextBox`. Właściwość `Text` kontrolki `TextBox` wymaga łańcucha, więc liczbę całkowitą trzeba najpierw przekształcić w łańcuch. Bywa też, że masz krótką liczbę całkowitą, a jakaś metoda wymaga długiej, więc konieczne jest jej przekształcenie.

Niektóre konwersje w .NET są wykonywane automatycznie. Nazywamy je konwersjami **niejawnymi**, ponieważ nie musimy dokonywać ich jawnie w kodzie.

Zarówno w VB, jak i w C# niejawnie wykonywane są konwersje **rozszerzające**. Zachodzą one podczas przekształcania typu mniejszego w większy, zwykle bez utraty informacji. Na przykład, przekształcanie krótkiej liczby całkowitej w długą to konwersja rozszerzająca, obsługiwana automatycznie przez VB i C#.

Visual Basic:

```
Dim i As Int16
Dim j As Int32
i = 7
j = i
```

C#:

```
Int16 i;
Int32 j;
i = 7;
j = i;
```

Zwróć uwagę, że w obu tych przykładach niepotrzebne jest rzutowanie czy przekształcanie typów. .NET obsługuje takie konwersje automatycznie.

Do konwersji **zawężających** dochodzi podczas przekształcania typu większego w mniejszy, na przykład długiej liczby całkowitej w krótką. Konwersje zawężające mogą prowadzić do utraty informacji, więc zwykle wymagają przekształcenia **jawnego**, co znaczy, że aplikacja musi definiować sposób konwersji.

Visual Basic:

```
Dim i As Int16
Dim j As Int32
j = 7
i = CType(j, Int16)
```

C#:

```
Int16 i;
Int32 j;
j = 7;
i = (Int16)j;
```

W powyższych przykładach kod VB używa funkcji `CType`, a kod C# — operatora konwersji, żeby jawnie przekształcić wartość na węższy typ danych.

Jawna konwersja wymagana jest ponadto w trakcie przekształcania niespokrewnionych typów danych, na przykład liczb całkowitych w łańcuchy.

Visual Basic:

```
Dim x As Int32
txtUserName.Text = x.ToString
```

C#:

```
Int32 x;
txtUserName.Text = x.ToString();
```

W tych przykładach wywołano metodę `ToString`, aby jawnie wykonać konwersję.

Domyślnie VB próbuje niejawnie przekształcać większość typów danych, wykonując konwersje zawężające a także konwersje między niespokrewnionymi typami. W zwykłych okolicznościach pokazane wcześniej przykłady VB nie wymagały jawnej konwersji. Ponieważ VB wykonuje je niejawnie, nie masz kontroli nad przekształcaniem typów. Może to prowadzić do błędów w programie.

C# nie wykonuje niejawnie konwersji zawężających ani konwersji między niespokrewnionymi typami danych.

Dużo lepiej jest samodzielnie przekształcać typy danych. Możesz to wymusić, używając instrukcji `Option Strict`. Dodaj ją na początku każdego pliku kodu VB:

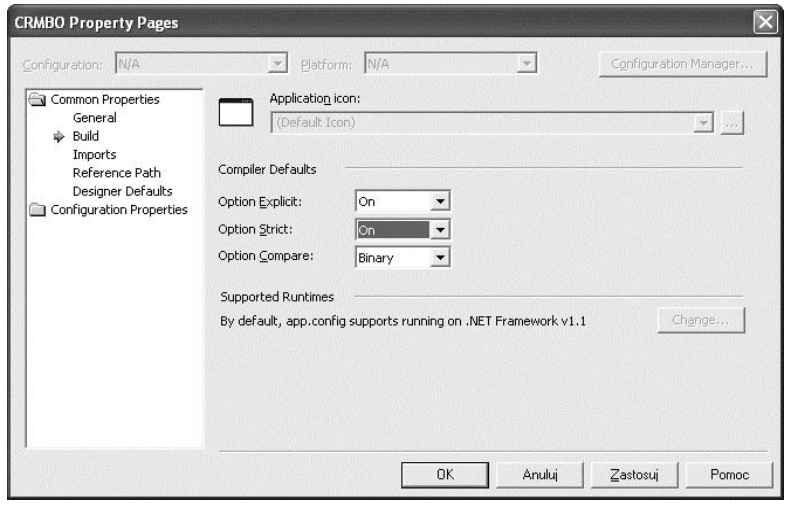
```
Option Strict On
```

Instrukcja `Option Strict` musi pojawiać się w kodzie przed wszystkimi pozostałymi.

Kiedy opcja `Strict` jest ustawiona na `On`, będziesz musiał jawnie przekształcać lub rzutować typy danych w przypadku konwersji zawężających i konwersji między niespokrewnionymi typami. Otrzymasz komunikat o błędzie składniowym, jeśli w kodzie znajdzie się którakolwiek z poniższych konstrukcji:

- konwersje zawężające bez operatora rzutowania;
- późne wiązanie (kiedy wynikiem wywołania metody jest typ `Object`, a nie konkretny typ danych, wywołanie nazywamy późno wiązanym);
- operacje na typie `Object` inne niż =, <>, `TypeOf...Is` oraz `Is`;
- pominięcie klauzuli `As` w deklaracji.

Możesz ustawić opcję `Strict` na `On` dla całego projektu na zakładce *Build* okna dialogowego *Property Pages* (rysunek 3.1).



Rysunek 3.1. *Opcja Strict jest domyślnie wyłączona. Włącz ją w oknie dialogowym Property Pages, aby wymusić ścisłą konwersję typów (tylko Visual Basic)*

Jeśli używasz Visual Basica, możesz wymusić ścisłą konwersję typów w całym kodzie, dodając instrukcję `Option Strict` do każdego pliku kodu albo ustawiając opcję *Option Strict* we właściwościach każdego projektu.

Ulepszone rzutowanie

Konieczność rzutowania zachodzi wówczas, gdy masz zmienną, którą trzeba przekształcić w inny typ danych, jak to opisano w poprzednim podrozdziale. Ponieważ .NET Framework używa wielu wartości typu `Object`, często potrzebne jest rzutowanie typu `Object` na konkretny typ.

W Visual Basicu najlepszym sposobem rzutowania typu `Object` na konkretny typ jest użycie instrukcji `DirectCast`:

```
Private Sub txt_Enter(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles txtUserName.Enter
     DirectCast(sender, TextBox).BackColor = Color.BlanchedAlmond
End Sub
```

W tym przykładzie instrukcja `DirectCast` rzutuje zmienną `sender` (typu `Object`) na typ `TextBox`, co pozwala uzyskać dostęp do właściwości kontrolki `TextBox`.

Instrukcja `DirectCast` działa wydajniej niż `CType` podczas rzutowania obiektu na bardziej specyficzny typ, ponieważ nie używa funkcji pomocniczych wywoływanych w czasie działania programu.

Instrukcji `DirectCast` możesz używać tylko w przypadku rzutowania typu na pokrewny, ale bardziej specyficzny typ, nazywany **typem pochodnym**. Możesz, na przykład, rzutować typ `Control` na `TextBox`, ponieważ `TextBox` wywodzi się z `Control`. Nie możesz użyć instrukcji `DirectCast` do przekształcenia liczby całkowitej w łańcuch, ponieważ typ `String` nie wywodzi się z typu `Integer`. Zawsze możesz używać jej do przekształcania typu `Object` w każdy inny typ, ponieważ wszystkie typy wywodzą się z typu `Object`.

Operator `DirectCast` zgłasza wyjątek `InvalidCastException`, jeśli docelowy typ nie wywodzi się z przekształcanego.

Choć w C# nie ma bezpośredniego odpowiednika instrukcji `DirectCast`, istnieje operator, który przekształca kompatybilne typy danych:

```
private void txt_Enter(object sender, System.EventArgs e)
{
    TextBox tb = sender as TextBox;
    tb.BackColor = Color.BlanchedAlmond;
}
```

Operator `as` przypomina `DirectCast`, ale różni się od niego pod jednym ważnym względem: nie zgłasza wyjątku, kiedy nie może przeprowadzić rzutowania, lecz zwraca wartość `null`. Operator `as` nie może rzutować na typy wartościowe, takie jak `Int`, lecz tylko na typy referencyjne.

W nadchodzącej wersji Visual Studio 2005 Visual Basic prawdopodobnie będzie zawierać nowy operator równoważny operatorowi `as` języka C#. Obecnie nosi on nazwę `TryCast`. Operator ten próbuje rzutowania, a jeśli operacja się powiedzie, zwraca wartość przekształconą w żądany typ. Jeśli operacja się nie powiedzie, zwraca wartość `Nothing`.

Aliasy typów danych

Kolejnym tajnikiem kodowania jest używanie aliasów typów danych. Jeśli zamierzasz użyć jakiegoś typu, ale spodziewasz się, że w przyszłości będziesz musiał zastąpić go innym, zdefiniuj alias typu danych i posługuj się nim w pozostałej części kodu.

Przypuśćmy, że chcesz użyć typu `Int16`, ale nie wykluczasz, że później zmienisz go na `Int32`. Zdefiniuj alias typu `Int16` w opisany niżej sposób.

Visual Basic:

```
Imports ChangeableType = System.Int16
```

C#

```
using ChangeableType = System.Int16;
```

Następnie we wszystkich miejscach, w których użyłbyś typu `Int16`, zastąp go nowo zdefiniowanym aliasem.

Visual Basic:

```
Private m_iSomething As ChangeableType
Debug.WriteLine(m_iSomething.GetTypeCode)
```

C#:

```
ChangeableType m_iSomething;
Debug.WriteLine(m_iSomething.GetTypeCode);
```

Jeśli później zdecydujesz się wykorzystać typ `Int32`, po prostu zmień alias, a w pozostałej części kodu będzie obowiązywał nowy typ.

Visual Basic:

```
Imports ChangeableType = System.Int32
```

C#

```
using ChangeableType = System.Int32;
```

Technikę tę najczęściej stosuje się w przypadkach, gdy nazwy pewnych klas są nieznane, ponieważ pisze je inny programista. Tworzysz więc namiastkę klasy, definiujesz alias nazwy klasy i możesz uruchomić swoją aplikację. Kiedy otrzymasz nową klasę, wystarczy, że wstawisz ją w miejsce namiastki i zmienisz nazwę klasy w aliasie.

Wyrażenia regularne

Wyrażenie regularne to wzorzec, który pozwala sprawdzić, czy pewna wartość (zwykle łańcuch) odpowiada specyficznym kryteriom. Przy użyciu wyrażenia regularnego możesz na przykład sprawdzić, czy numer telefonu zawiera wyłącznie cyfry albo prawidłową kombinację cyfr, nawiasów i myślników. Jest to szczególnie przydatne podczas sprawdzania poprawności danych wpisanych przez użytkownika. Wyrażenia regularne przydają się też do zastępowania lub usuwania części łańcucha, które pasują do pewnego wzorca, na przykład do usuwania wszystkich znaków specjalnych z łańcucha.

Wyrażeń regularnych najlepiej uczyć się, wykorzystując przykłady. Dzięki temu poznajesz ogólne zasady tworzenia wyrażeń regularnych i używania ich w kodzie. Warto też jednak zrozumieć, jak budować wyrażenia regularne przy użyciu odpowiedniej składni.

W kolejnych punktach najpierw zostaną przedstawione przykłady zastosowania wyrażeń regularnych w rzeczywistych aplikacjach, a następnie szczegółowe informacje o samodzielnym budowaniu takich wyrażeń.

Używanie wyrażeń regularnych

Jednym z najczęstszych zastosowań wyrażeń regularnych jest weryfikowanie danych wprowadzonych przez użytkownika. W poniższych podpunktach podano przykłady sprawdzania poprawności numerów PESEL, kodów pocztowych oraz numerów identyfikacji podatkowej (NIP).

Numery PESEL

Zacznijmy od prostego przykładu: możesz użyć wyrażeń regularnych do sprawdzenia, czy łańcuch zawiera wyłącznie wartości liczbowe. Za pomocą tej techniki upewnisz się, czy numer PESEL wprowadzony przez użytkownika zawiera wyłącznie cyfry.

Zamiast korzystać z wyrażeń regularnych, mógłbyś wykonać ten test za pomocą funkcji `IsNumeric` Visual Basica, która sprawdza, czy wyrażenie można przekształcić w liczbę. Aby wykorzystać tę funkcję z poziomu C#, dodaj referencję do biblioteki *Microsoft.VisualBasic.dll*.

Chcąc sprawdzić, czy dany łańcuch zawiera tylko cyfry, użyj następującego wzorca:

```
^\d+$
```

Daszek (^) wskazuje, że dopasowywanie wzorca powinno rozpocząć się od pierwszego znaku łańcucha wpisanego przez użytkownika. Symbol `\d` definiuje cyfrę (0–9). Znak plusa (+) wymaga, aby poprzedzający go znak pojawiał się jeden raz lub więcej razy; w tym przypadku oznacza to jedną cyfrę lub więcej cyfr. Znak dolara ($) wskazuje, że dopasowywanie wzorca powinno zakończyć się na ostatnim znaku łańcucha wpisanego przez użytkownika. Dodatkowe informacje o znakach używanych w wyrażeniach regularnych znajdziesz w tabeli 3.2 w dalszej części rozdziału.

Przestrzeń nazw `System.Text.RegularExpressions` w .NET Framework oferuje zbiór funkcji, które pomagają w pracy z wyrażeniami regularnymi. Możesz wykorzystać te funkcje do dopasowywania wzorców i zastępowania znaków przy użyciu wyrażeń regularnych. Dzięki poniższym przykładom dowiesz się, jak sprawdzić, czy numer PESEL wpisany przez użytkownika zawiera wyłącznie cyfry.

Visual Basic:

```
Imports System.Text.RegularExpressions
Dim sRegExPattern As String = "^\d+$"
Dim sTextToValidate As String = txtPesel.Text
If Not Regex.IsMatch(sTextToValidate, sRegExPattern) Then
    epValidation.SetError(ctrl, "Numer PESEL musi składać się wyłącznie z cyfr.")
End if
```

C#:

```
using System.Text.RegularExpressions;
string sRegExPattern = @"^\d+$";
string sTextToValidate = txtPesel.Text;
if(!Regex.IsMatch(sTextToValidate, sRegExPattern))
{
    epValidation.SetError(ctrl, "Numer PESEL musi składać się wyłącznie z cyfr.");
}
```

Metoda `IsMatch` klasy `Regex` zwraca `True`, jeśli tekst pasuje do zdefiniowanego wzorca. Jeśli metoda `IsMatch` zwróci `False`, kontrolka `ErrorProvider` powiadamia użytkownika o błędzie (jeśli nie miałeś do czynienia z kontrolką `ErrorProvider`, zajrzyj do odpowiedniego podrozdziału w rozdziale 2.). Mógłbyś uogólnić kod w podanych przykładach tak, aby porównywać dowolny wzorzec wyrażenia regularnego (zdefiniowany w zmiennej `sRegExPattern`) z dowolnym łańcuchem (zdefiniowanym w zmiennej `sTextToValidate`).

Zwróć uwagę na znak at (@) we wzorcu wyrażenia regularnego w kodzie C#. Zapobiega on przetwarzaniu znaków specjalnych przez kompilator C#.

Kody pocztowe

Kolejnym, często spotykanym wzorcem, jest kod pocztowy w postaci `xx-xxx`. Opisana poniżej technika może przydać się nie tylko do weryfikowania kodów pocztowych, ale również innych wzorców liczbowych, takich jak numery klienta albo dostawcy.

Aby sprawdzić, czy wpisany kod pocztowy składa się z dwóch cyfr, myślnika i trzech cyfr, użyj poniższego wzorca:

```
^\d{2}-\d{3}$
```

Wzorzec ten rozpoczyna dopasowywanie od początku łańcucha (^), dopasowuje dwie cyfry (\d{2}), myślnik (-) i trzy cyfry (\d{3}), i kończy dopasowywanie na końcu łańcucha ($).

Wykorzystując ten wzorzec w swojej aplikacji, zastosuj kod podobny do tego, który weryfikuje numery PESEL (zamieszczony w poprzednim punkcie) i podaj kod pocztowy jako wyrażenie regularne.

Numery telefoniczne

Aplikacja przetwarzająca informacje kontaktowe albo dane klientów lub pracowników powinna sprawdzać poprawność numerów telefonicznych. Kłopot z numerami telefonicznymi polega na tym, że użytkownicy nigdy nie są pewni, czy i gdzie mają wpisywać nawiasy, spacje, myślniki albo inne znaki specjalne. Zanim więc sprawdzisz poprawność numeru telefonicznego, musisz usunąć z niego wszystkie nadmiarowe znaki.

Oto wyrażenie regularne dopasowujące standardowy numer telefoniczny (dwie cyfry numeru kierunkowego i siedem cyfr numeru lokalnego):

```
^\d{9}$
```

Daszek (^) i znak dolara ($) określają, czy do wzorca pasuje cały łańcuch, a symbol \d{9} wskazuje ciąg 9 cyfr.

Przed sprawdzeniem poprawności numeru musisz usunąć z niego wszystkie znaki specjalne wpisane przez użytkownika. Służy do tego metoda `Replace` klasy `Regex`. Kod, który usuwa znaki specjalne, a następnie sprawdza poprawność numeru, przypomina kod z poprzedniego przykładu, ale zawiera dodatkowy wiersz przeznaczony do zastąpienia znaków.

Visual Basic:

```
Imports System.Text.RegularExpressions
Dim sRegExPattern As String = "^\d{9}$"
Dim sTextToValidate As String = txtPhone.Text
sTextToValidate = Regex.Replace(sTextToValidate, "[^0-9]", "")
If Not Regex.IsMatch(sTextToValidate, sRegExPattern) Then
    epValidation.SetError(ctrl, "Numer telefoniczny musi zawierać 9 cyfr.")
End if
```

C#:

```
using System.Text.RegularExpressions;
string sRegExPattern = @"^\d{9}";
string sTextToValidate = txtPhone.Text;
sTextToValidate = Regex.Replace(sTextToValidate, "[^0-9]", "");
```

```
if(!Regex.IsMatch(sTextToValidate, sRegExPattern))
{
    epValidation.SetError(ctrl, "Numer telefoniczny musi zawierać 9 cyfr.");
}
```

Powyższy kod używa metody `Replace` klasy `Regex`, aby każdy znak nienumeryczny, zdefiniowany jako nienależący do zakresu 0–9 (`[^0-9]`), zastąpić łańcuchem pustym. Powoduje to usunięcie wszystkich nadmiarowych znaków. Potem kod używa metod `IsMatch`, aby sprawdzić, czy wynikowy łańcuch pasuje do wzorca numeru telefonicznego.

Budowanie wyrażeń regularnych

Kiedy poznasz ogólne zasady użycia wyrażeń regularnych, łatwiej będzie Ci zrozumieć, jak budować własne.

Większość znaków w wyrażeniu regularnym to zwykłe znaki. Niektóre są szczególnie istotne dla mechanizmu przetwarzającego wyrażenia regularne, więc trzeba je poprzedzać znakiem unikowym — lewym ukośnikiem (\) — w sposób zaprezentowany w tabeli 3.2.

Wszystkie zwykłe znaki (bez znaku unikowego) pasują do samych siebie, z wyjątkiem znaków specjalnych zdefiniowanych w tabeli 3.3. Aby użyć znaków specjalnych, należy poprzedzić je znakiem unikowym. Na przykład, w celu dopasowania łańcucha `(nnn)` należy użyć wzorca `^\(\d{3}\)$`. Rozpoczyna on dopasowywanie od początku łańcucha, dopasowuje lewy nawias okrągły, trzy cyfry oraz prawy nawias okrągły i kończy dopasowywanie na końcu łańcucha.

Przeciążanie procedur

To właściwie nie jest żaden sekret; jeśli programujesz, prawdopodobnie przeciążasz procedury. Jeśli nie, oto szybkie wprowadzenie.

Przeciążanie polega na definiowaniu wielu procedur, konstruktorów albo właściwości, które mają takie same nazwy, ale inne sygnatury. Podstawowym celem przeciążania jest zaoferowanie wielu zbiorów parametrów, których można użyć do wykonania jakiejś operacji.

Przypuśćmy, że klasa `Customer` ma metodę `Retrieve`. Załóżmy, że chcemy pobierać wszystkie kluczowe dane klientów w celu umieszczenia ich na liście rozwijanej, wszystkie kluczowe dane klientów z określonego regionu w celu umieszczenia ich na filtrowanej liście rozwijanej oraz wszystkie dane określonego klienta według jego identyfikatora.

Moglibyśmy osiągnąć ten cel na kilka różnych sposobów: utworzyć trzy różne metody z różnymi nazwami, utworzyć jedną metodę z parametrami opcjonalnymi albo utworzyć trzy metody o takiej samej nazwie, ale różnych sygnaturach. Ta ostatnia technika wykorzystuje przeciążanie procedur (lub metod).

Tabela 3.2. *Znaki używane w wyrażeniach regularnych*

Znak	Dopasowuje	Przykład
`\b`	granicę słowa	`\bna` dopasowuje ciąg `na` znajdujący się na początku słowa, na przykład „Chodź tu **na**tychmiast”. `na\b` dopasowuje ciąg `na` znajdujący się na końcu słowa, na przykład „Pierwsza zmia**na**”. `\bna\b` dopasowuje tylko całe słowo `na`, na przykład „Czas **na** zmianę”.
`\B`	środek słowa	`\Bna` dopasowuje słowo `na` znajdujące się gdziekolwiek z wyjątkiem początku słowa, na przykład „To z**na**jduje się w tym”.
`\b`	znak cofania	`\b` reprezentuje znak cofania w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\d`	cyfry od 0 do 9	`\d` dopasowuje pierwszą cyfrę w łańcuchu, na przykład „**4**44B Privet Drive”.
`\D`	znak inny niż cyfra	`\D` dopasowuje pierwszy znak inny niż cyfra w dowolnym miejscu łańcucha, na przykład „444**B** Privet Drive”.
`\e`	znak escape	`\e` reprezentuje znak escape w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\f`	znak wysuwu strony	`\f` reprezentuje znak wysuwu strony w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\n`	znak nowego wiersza	`\n` reprezentuje znak nowego wiersza w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\r`	znak powrotu karetki	`\r` reprezentuje znak powrotu karetki w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\s`	spację	`\s` dopasowuje pierwszą spację, na przykład „444B Privet Drive”.
`\S`	znak inny niż spacja	`\S` dopasowuje pierwszy znak inny niż spacja, na przykład „**4**44B Privet Drive”.
`\t`	znak tabulacji	`\t` reprezentuje znak tabulacji w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\v`	znak tabulacji pionowej	`\v` reprezentuje znak tabulacji pionowej w zbiorze zdefiniowanym w nawiasie kwadratowym `[]` (zobacz tabela 3.3). Używa się go przede wszystkim we wzorcach zastępowania.
`\w`	dowolny znak alfanumeryczny łącznie ze znakiem podkreślenia	`\w` dopasowuje pierwszy znak alfanumeryczny lub znak podkreślenia, na przykład „**4**44B Privet Drive”.
`\W`	dowolny znak niealfanumeryczny	`\W` dopasowuje pierwszy znak niealfanumeryczny albo znak podkreślenia, na przykład „444B Privet Drive”.

Tabela 3.3. *Znaki specjalne używane w wyrażeniach regularnych*

Znak	Dopasowuje	Przykład
`.`	dowolny znak z wyjątkiem znaku nowego wiersza (`\n`)	`.` dopasowuje dowolny znak, na przykład „**4**44B Privet Drive". Często używa się go, aby sprawdzić, czy użytkownik wpisał jakieś dane w polu.
`[...]`	zbiór określonych znaków	`[ABC]` dopasowuje pierwsze wystąpienie litery `a`, `b` lub `c`, na przykład „444**B** Privet Drive".
`[^...]`	zbiór znaków innych niż określone	`[^ABC]` dopasowuje pierwsze wystąpienie litery innej niż `a`, `b` lub `c`, na przykład „**4**44B Privet Drive".
`[a-bm-n]`	zbiór określonych zakresów znaków	`[0-9a-kA-K]` dopasowuje pierwsze wystąpienie jakiekolwiek cyfry z zakresu 0 – 9, małej litery z zakresu a – k lub wielkiej litery z zakresu A – K, na przykład „**4**44B Privet Drive".
`*`	poprzedni znak zero lub większą ilość razy	`\d*` dopasowuje cały ciąg cyfr, na przykład „**444**B Privet Drive".
`+`	poprzedni znak jeden lub większą ilość razy	`\d*` dopasowuje cały ciąg cyfr, na przykład „**444**B Privet Drive".
`?`	poprzedni znak zero lub jeden raz	`\d?` dopasowuje ciąg składający się z zera cyfr lub jednej cyfry, na przykład „**444**B Privet Drive".
`{n}`	poprzedni znak dokładnie `n` razy	`\d{2}` dopasowuje ciąg składający się z dokładnie 2 cyfr, na przykład „**44**4B Privet Drive".
`{n,}`	poprzedni znak `n` lub większą ilość razy	`\d{2,}` dopasowuje ciąg składający się z 2 lub większej ilości cyfr, na przykład „**444**B Privet Drive".
`{n,m}`	poprzedni znak co najmniej `n` i najwyżej `m` razy	`\d{1,3}` dopasowuje ciąg składający się z co najmniej 1 i najwyżej 3 cyfr, na przykład „**444**B Privet Drive".
`^`	wzorzec występujący na początku łańcucha	`^4` dopasowuje cyfrę `4`, jeśli występuje ona na początku łańcucha, na przykład „**4**44B Privet Drive".
`$`	wzorzec występujący na końcu łańcucha	`$Drive` dopasowuje słowo `Drive`, jeśli występuje ono na końcu łańcucha, na przykład „444B Privet **Drive**".
`a|b`	`a` lub `b`	`P | Q` dopasowuje `P` lub `Q`, na przykład „444B **P**rivet Drive".

Nadanie wszystkim trzem metodom tej samej nazwy ułatwia ich wykorzystanie. Przeciążanie jest również lepsze od parametrów opcjonalnych, ponieważ sygnatury metod są jednoznaczne (poza tym parametry opcjonalne są niedostępne w C#).

Korzystanie z przeciążania nie wymaga żadnych specjalnych technik ani słów kluczowych; wystarczy, że utworzysz dwie procedury lub większą ich ilość o tej samej nazwie i różnych zbiorach parametrów.

Visual Basic:

```
Public Function Retrieve() As DataSet
    ' Pobiera kluczowe dane wszystkich klientów
End Function

Public Function Retrieve(ByVal sRegion As String) As DataSet
    ' Pobiera kluczowe dane wszystkich klientów z określonego regionu
End Function
```

```
Public Function Retrieve(ByVal iCustomerID As Int32) As DataSet
    ' Pobiera wszystkie dane określonego klienta
End Function
```

C#:

```
public DataSet Retrieve()
{
    // Pobiera kluczowe dane wszystkich klientów
}

public DataSet Retrieve(String sRegion)
{
    // Pobiera kluczowe dane wszystkich klientów z określonego regionu
}

public Retrieve(Int32 iCustomerID)
{
    // Pobiera wszystkie dane określonego klienta
}
```

Każda z tych procedur ma inną sygnaturę. Pierwsza metoda nie przyjmuje żadnych parametrów, druga przyjmuje pojedynczy łańcuch, a trzecia liczbę całkowitą. Możesz zdefiniować dowolnie wiele przeciążonych procedur z różnymi kombinacjami typów parametrów.

W niektórych przypadkach, jak w powyższym przykładzie, kod przeciążonych metod jest różny. Bywa jednak i tak, że metody są bardzo podobne — możesz wtedy umieścić wspólny kod w jednej metodzie i wywoływać ją z pozostałych metod z odpowiednimi parametrami.

Kiedy wywołujesz przeciążoną metodę, środowisko uruchomieniowe wybiera odpowiednią wersję na podstawie parametrów określonych w wywołaniu. Jeśli na przykład wywołasz metodę `Retrieve` i przekażesz parametr w postaci liczby całkowitej, zostanie wykonana ta wersja metody `Retrieve`, która przyjmuje parametr całkowity.

Przeciążanie rządzi się pewnymi regułami. Najbardziej oczywista jest ta, że procedury muszą mieć taką samą nazwę. Sygnatury metod (typy danych parametrów) muszą się różnić, ponieważ w przeciwnym razie środowisko uruchomieniowe nie mogłoby ustalić, którą wersję metody należy wykonać. Nie możesz na przykład napisać dwóch metod o nazwie `Retrieve`, z których każda przyjmuje pojedynczy parametr w postaci łańcucha. Do odróżniania metod nie możesz też wykorzystać:

- typu wartości zwracanej przez metodę,
- sposobu przekazywania parametrów — przez wartość (`ByVal`) albo przez referencję (`ByRef`).

Przeciążanie procedur ułatwia pisanie i konserwację programów. Poza tym upraszcza sposób korzystania z metod.

Przyjrzyj się swoim programom i zastanów się, w których miejscach możesz wykorzystać przeciążanie. Dobrymi „kandydatami" są tu metody z parametrami opcjonalnymi a także procedury wykonujące podobne zadania, ale o innych zbiorach parametrów i różnych nazwach.

Przeciążanie operatorów

Przeciążanie operatorów to jedna z tych funkcji, które rzadko są potrzebne, ale jeśli już, to **naprawdę**. Obecnie przeciążanie operatorów jest dostępne tylko w C# (choć w nowym Visual Studio 2005 Visual Basic również ma je obsługiwać).

Przeciążanie operatorów polega na definiowaniu operatorów, takich jak +, - i *, dla własnych typów danych. Pozwala to manipulować obiektami przy użyciu standardowych operatorów i ułatwia pracę z samodzielnie zdefiniowanymi typami danych.

Rozważmy aplikację, która przetwarza ceny podane w walutach obcych. Typ danych `Price` pozwala przechowywać kwoty wyrażone w dowolnej walucie. Jeśli przeciążymy operatory matematyczne dla typu danych `Price`, będziemy mogli manipulować cenami w różnych walutach. Zważywszy na globalizację współczesnej gospodarki, taka funkcja byłaby bardzo przydatna.

Typ danych `Price` to struktura złożona z kilku pól:

```
public struct Price
{
    public float Amount;
    public string CurrencyCode;
    public float ExchangeRateWRTUSD;
}
```

Pole `Amount` definiuje kwotę w zdefiniowanej walucie. Pole `CurrencyCode` to trzyliterowy kod waluty, na przykład USD (dolar amerykański), PLN (złoty) lub EUR (euro).

W rzeczywistej aplikacji wystarczyłyby pola `Amount` i `CurrencyCode`. Program używałby pola `CurrencyCode` do wyszukania odpowiednich kursów wymiany w tabeli (jeśli aktualność informacji nie jest wymagana) albo pobrania ich z usługi (jeśli informacje muszą być aktualne).

Jednakże pisanie kodu przeszukującego tabele albo łączącego się z usługą sieciową odciągnęłoby nas od tematu niniejszego podrozdziału: przeciążania operatorów. Dlatego do typu danych `Price` dodano trzecie pole, które określa kurs wymiany.

Jak wiadomo, każda waluta ma pewien kurs wymiany w odniesieniu do każdej innej. Standardowy sposób manipulowania cenami wyrażonymi w różnych walutach polega na znalezieniu ich kursu wymiany, przekształceniu jednej waluty w drugą i wykonaniu operacji.

Niestety, w przedstawionym niżej przykładzie nie można z góry określić, w jaką walutę trzeba będzie przekształcać poszczególne obiekty `Price`. Nie wystarczy więc zdefiniowanie pojedynczego kursu wymiany dla typu danych `Price`; trzeba by przechowywać w nim kursy wymiany dla wszystkich możliwych walut.

W takiej sytuacji lepiej jest zdefiniować walutę bazową. Typ danych `Price` ma pojedyncze pole, które przechowuje kurs wymiany w odniesieniu do waluty bazowej. Aplikacja przekształca każdą kwotę w walutę bazową, wykonuje operację, a następnie przekształca wynik z powrotem w żądaną walutę. W tym przykładzie wybrano dolara amerykańskiego, ale mogłaby to być każda inna waluta.

Konstruktor typu danych `Price` ustawia wartości pól i umożliwia utworzenie obiektu `Price` z odpowiednimi danymi.

```
public Price(float fAmount,
             string sCurrencyCode,
             float fExchangeRateWRTUSD)
{
    this.Amount = fAmount;
    this.CurrencyCode = sCurrencyCode;
    this.ExchangeRateWRTUSD = fExchangeRateWRTUSD;
}
```

Przeciążanie operatorów matematycznych

Aby umożliwić dodawanie cen, moglibyśmy zdefiniować metodę `Add`, która przyjmowałaby dwa obiekty `Price`. Bardziej intuicyjne byłoby jednak dodawanie dwóch cen przy użyciu operatora +. Właśnie do tego służy przeciążanie operatorów.

Aby przeciążyć operator, należy zdefiniować statyczną metodę ze słowem kluczowym `operator`:

```
public static Price operator +(Price p1, Price p2)
{
    Price p3;
    if (p1.ExchangeRateWRTUSD == p2.ExchangeRateWRTUSD)
    {
        p3 = new Price(p1.Amount+p2.Amount,
                       p1.CurrencyCode,p1.ExchangeRateWRTUSD);
    }
    else
    {
        float p1InUSD = p1.Amount * p1.ExchangeRateWRTUSD;
        float p2InUSD = p2.Amount * p2.ExchangeRateWRTUSD;
        float p3Converted = (p1InUSD + p2InUSD) / p1.ExchangeRateWRTUSD;
        p3 = new Price(p3Converted,p1.CurrencyCode,p1.ExchangeRateWRTUSD);
    }
    return p3;
}
```

Metoda najpierw porównuje dwa kursy wymiany. Jeśli są takie same, dodaje ceny. Jeśli się różnią, obie ceny są przekształcane w walutę bazową (USD) i dodawane, po czym wynik przekształcany jest z powrotem w walutę pierwszego obiektu `Price`. Na koniec metoda zwraca nowy obiekt `Price`, który zawiera sumę dwóch pierwotnych cen.

Przeciążonego operatora możesz używać w połączeniu ze swoim typem danych tak samo jak w połączeniu z typami podstawowymi. Po prostu dodaj dwa obiekty `Price`:

```
Price keyboardPrice = new Price(40, "PLN", (float).25);
Price mousePrice = new Price(20, "EUR", (float)1.25);
Price Total = keyboardPrice + mousePrice;
MessageBox.Show("Łączna cena zamówienia to: " + Total.Amount.ToString());
```

Powyższy kod najpierw tworzy dwa obiekty `Price`, następnie dodaje je przy użyciu operatora + i wyświetla otrzymany wynik.

Możesz napisać podobny kod w celu przeciążenia operatorów -, * lub + dla obiektów `Price`.

Przeciążanie operatorów przypisania

Operatorów przypisania, takich jak znak równości (=), nie można przeciążać. W razie przeciążenia operatorów matematycznych można jednak korzystać ze skróconych operatorów przypisania, takich jak +=, -=, *= i /=.

```
Price speakerPrice = new Price(30, "PLN", (float).25);
Price surCharge = new Price(5, "PLN", (float).25);
speakerPrice += surCharge;
MessageBox.Show("Końcowa cena głośnika to: " +
   speakerPrice.Amount.ToString());
```

Powyższy kod definiuje kwotę dopłaty i zwiększa cenę towaru o tę kwotę.

Jeśli przeciążysz inne operatory matematyczne, będziesz mógł korzystać z tej techniki także w przypadku pozostałych skróconych operatorów przypisania.

Przeciążanie operatorów porównania

Możesz przeciążać również operatory porównania: równy (= w VB, == w C#), nierówny (<> w VB, != w C#), mniejszy (<), większy (>), mniejszy lub równy (<=) oraz większy lub równy (>=).

Kod, który przeciąża operator większości, zwraca wartości logiczne: `True` lub `False`.

```
public static bool operator >(Price p1, Price p2)
{
    // Jeśli oba kursy wymiany są takie same, po prostu porównujemy kwoty
    if (p1.ExchangeRateWRTUSD == p2.ExchangeRateWRTUSD)
    {
        return (p1.Amount > p2.Amount);
    }
    else
    {
        float p1InUSD = p1.Amount * p1.ExchangeRateWRTUSD;
        float p2InUSD = p2.Amount * p2.ExchangeRateWRTUSD;
        return (p1InUSD > p2InUSD);
    }
}
```

Ten kod porównuje kursy wymiany. Jeśli są takie same, kod zwraca `True` w przypadku, gdy pierwsza kwota jest większa od drugiej. Jeśli kursy są różne, obie kwoty są przekształcane na walutę bazową i dopiero wtedy porównywane.

Operatory porównania trzeba przeciążać parami. Jeśli przeciążysz operator ==, musisz również przeciążyć operator `!=`. Ponieważ w tym przykładzie przeciążyliśmy operator większości (>), musimy przeciążyć operator mniejszości (<).

```
public static bool operator <(Price p1, Price p2)
{
    // Jeśli oba kursy wymiany są takie same, po prostu porównujemy kwoty
    if (p1.ExchangeRateWRTUSD == p2.ExchangeRateWRTUSD)
    {
        return (p1.Amount < p2.Amount);
    }
    else
    {
        float p1InUSD = p1.Amount * p1.ExchangeRateWRTUSD;
        float p2InUSD = p2.Amount * p2.ExchangeRateWRTUSD;
        return (p1InUSD < p2InUSD);
    }
}
```

Powyższy kod jest bardzo podobny do tego, który przeciąża operator większości.

Nie można przeciążać warunkowych operatorów logicznych (`&&` oraz `||`).

Przeciążanie operatorów pozwala na intuicyjne wykonywanie działań matematycznych oraz porównywanie obiektów. W Visual Studio 2005 funkcja ta ma zostać rozbudowana.

Eksploracja nieodkrytych regionów

Prawdopodobnie zauważyłeś zwinięte sekcje *Windows Forms Designer generated code* w kodzie Windows Forms. Takie sekcje noszą nazwę **regionów**. Regiony są pomocne przy organizacji kodu i ułatwiają pisanie oraz konserwację programów.

Aby utworzyć region wokół fragmentu kodu, użyj słowa kluczowego `Region`.

Visual Basic:

```
#Region "Metody prywatne"
    ' Tutaj umieść dowolny kod
#End Region
```

C#:

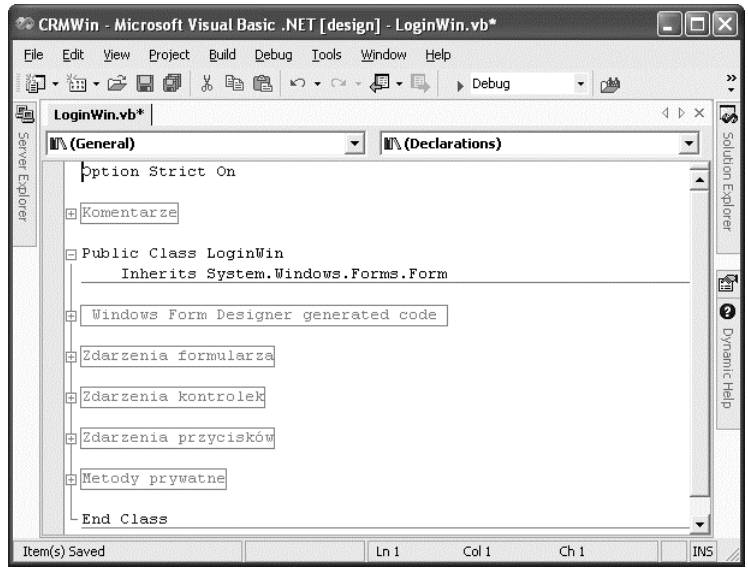
```
#region Metody prywatne
    // Tutaj umieść dowolny kod
#endregion
```

W swoich plikach kodu możesz zdefiniować dowolną liczbę regionów. Możesz je również zagnieżdżać, tworząc regiony regionów. Przydaje się to, na przykład, do zdefiniowania regionu *Metody prywatne*, a następnie oddzielnego regionu dla każdej metody.

Chcąc, by przeszukiwane były zamknięte regiony, musisz zaznaczyć pole *Search hidden text* w oknie dialogowym *Find*. W przeciwnym razie zamknięte regiony zostaną pominięte.

Jeśli podzielisz cały kod na regiony, będziesz mógł obejrzeć ogólną strukturę pliku kodu (rysunek 3.2). Rozwijanie i zwijanie odpowiednich regionów pomoże Ci skupić się na bieżącym zadaniu.

Rysunek 3.2. *Ogólny widok całego kodu zawartego w pliku. Rozwiń odpowiedni region, aby wyświetlić interesujący Cię kod*



Zdefiniuj regiony, aby ułatwić zarządzanie kodem zarówno sobie, jak i przyszłym pokoleniom konserwatorów programu.

Komentarze XML

C# oferuje programistom mechanizm dokumentowania kodu przy użyciu standardowych elementów XML. Dzięki temu mechanizmowi komentarze zyskują jednolity styl, a środowisko programistyczne Visual Studio może wykorzystać je w funkcji IntelliSense. Programiści VB również mogą dokumentować swój kod przy użyciu XML, ale muszą wprowadzać komentarze ręcznie, a Visual Studio ich nie wykorzystuje (funkcja komentowania XML w Visual Basicu ma być dostępna w Visual Studio 2005).

Tworzenie komentarzy XML

Komentarzy XML można używać w języku C# do dokumentowania klas, delegacji, interfejsów, pól, zdarzeń, właściwości lub metod. Aby utworzyć komentarz XML do pewnego elementu, przejdź do wiersza znajdującego się tuż nad definicją elementu w pliku kodu C# i wpisz trzy ukośniki (///). Kiedy wpiszesz trzeci ukośnik, automatycznie pojawią się komentarze XML:

```
/// <summary>
///
/// </summary>
/// <param name="pnl"></param>
private void AddEventHandlers(Control pnl)
```

W elemencie <summary> możesz podać krótki opis klasy lub metody. Zwróć uwagę, że element <param> zawiera w atrybucie name parametr pobrany z sygnatury metody. Możesz dodać definicję parametru. Minimalny komentarz XML do metody mógłby wyglądać tak:

```
/// <summary>
/// Przetwarza wszystkie kontrolki na formularzu i ustawia procedury
/// obsługa zdarzeń Enter i Leave dla każdej kontrolki TextBox.
/// </summary>
/// <param name="pnl">Formularz albo dowolna kontrolka na formularzu.</param>
private void AddEventHandlers(Control pnl)
```

Narzędzie do tworzenia komentarzy XML oferuje kilka dodatkowych elementów:

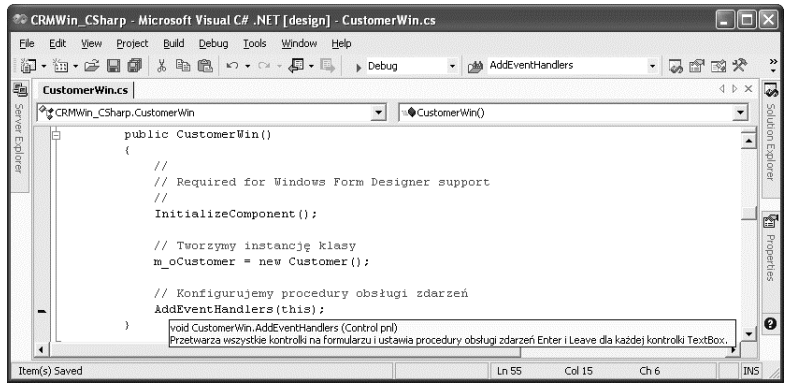
- <example> — pozwala podać przykład użycia metody albo innej składowej klasy;
- <exception> — pozwala określić wyjątki, które może zgłosić metoda;
- <returns> — pozwala określić wartość zwrotną metody.

Pełne omówienie komentarzy XML w języku C# znajdziesz pod hasłem „XML Documentation tutorial” w pomocy MSDN.

Oglądanie komentarzy XML w IntelliSense

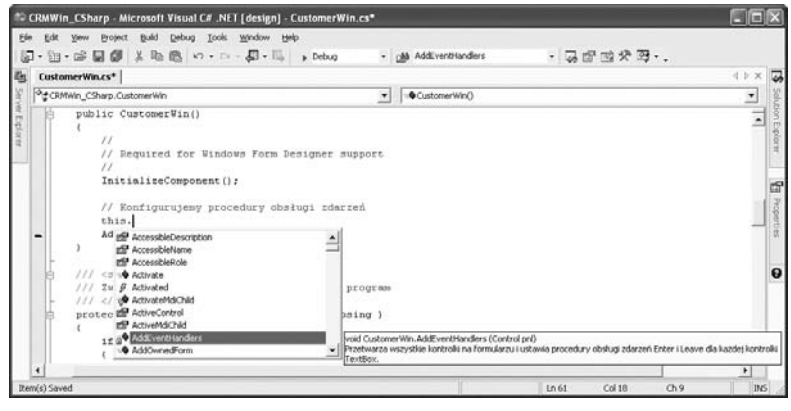
Funkcja IntelliSense w Visual Studio używa komentarzy XML w szybkim podglądzie (Quick Info), na listach składowych (List Members) oraz w informacjach o parametrach (Parametr Info). Jeśli nawet nie znasz tych nazw, prawdopodobnie widziałeś już powyższe funkcje w działaniu.

- **Quick Info** — wyświetla deklarację oraz skrótowy opis każdego identyfikatora w kodzie. Przesuń wskaźnik myszy nad dowolny identyfikator albo kliknij przycisk *Quick Info* umieszczony na pasku narzędzi *Text Editor*, aby zobaczyć szybki podgląd identyfikatora (rysunek 3.3). Zauważ, że pojawia się w nim tekst komentarza XML zdefiniowany w poprzednim przykładzie.



Rysunek 3.3. *Funkcja Quick Info to część IntelliSense, wyświetlająca krótkie informacje o identyfikatorach zawartych w kodzie. Opis identyfikatora pobierany jest z elementu <summary> komentarza XML*

- **List Members** — wyświetla składowe konkretnego obiektu na liście rozwijanej. Kiedy wpiszesz nazwę obiektu i kropkę (.) albo klikniesz przycisk *Member List* umieszczony na pasku narzędzi *Text Editor*, pojawi się lista składowych. Zatrzymaj wskaźnik myszy nad wybraną pozycją tej listy, aby obejrzeć podgląd Quick Info (rysunek 3.4).

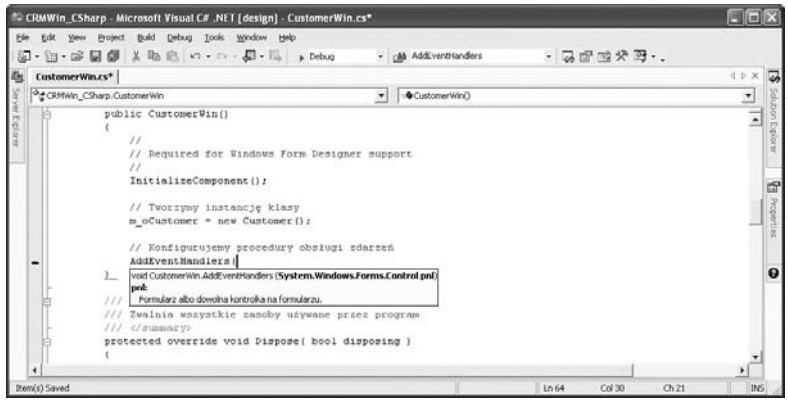


Rysunek 3.4. *Funkcja List Members również wykorzystuje komentarze XML*

- **Parameter Info** — wyświetla informacje o parametrach metody. Pojawiają się one po wpisaniu nazwy metody i nawiasu albo po kliknięciu przycisku *Parameter Info* umieszczonego na pasku narzędzi *Text Editor* (rysunek 3.5).

Możesz wygenerować zewnętrzny plik ze wszystkimi komentarzami XML zawartymi w projekcie. Wybierz z menu *Project* polecenie *Properties*, otwórz folder *Configuration Properties* i wybierz pozycję *Build*. Następnie wpisz nazwę pliku w polu *XML Documentation File* pod węzłem *Output*. Kiedy zbudujesz projekt, wygenerowany zostanie plik XML. Kolejną korzyścią będzie to, że wszystkie dostępne publicznie typy i metody bez komentarza XML zostaną wymienione jako ostrzeżenia w oknie *Task List* (więcej informacji o pracy z oknem *Task List* znajdziesz w rozdziale 1.).

Wygenerowany w ten sposób plik XML można przetwarzać przy użyciu zewnętrznych narzędzi, na przykład NDoc (*http://sourceforge.net/projects/ndoc*), w celu utworzenia plików pomocy na podstawie komentarzy XML zawartych w projekcie.



Rysunek 3.5. *Funkcja Parameter Info pobiera informacje z elementu <param> w komentarzu XML*

Jeśli będziesz używał komentarzy XML we wszystkich metodach i innych składowych klas, automatycznie wygenerujesz dokumentację projektu, a ponadto ułatwisz sobie korzystanie z tych metod, ponieważ dokumentacja będzie pojawiać się w IntelliSense.

Dodawaj komentarze XML do nowych i zmodyfikowanych metod. Jeśli programujesz w C#, Visual Studio ułatwi Ci to zadanie. Jeśli korzystasz z VB, dodawaj komentarze XML ręcznie, używając trzech apostrofów (' ' ') jako ogranicznika. Choć obecnie programiści VB nie mogą wykorzystać wszystkich zalet komentarzy XML, niedługo się to zmieni.

Oznaczanie przestarzałego kodu

Ostatnimi czasy rośnie zainteresowanie metodologią programowania „elastycznego" (ang. *agile development*). Jednym z kanonów tej metodologii jest kodowanie pod kątem zmian, to znaczy pisanie kodu, który można łatwo zmodyfikować i zaadaptować do zmian biznesowych. Pozwala to zrezygnować ze sztywno określonych wymagań i lepiej dostosować aplikację do potrzeb firmy i użytkowników (jeśli nigdy nie słyszałeś o programowaniu elastycznym, więcej informacji na ten temat znajdziesz w rozdziale 5.).

Kiedy kod stopniowo się zmienia, czasem konieczna okazuje się **refaktoryzacja** fragmentów programu. Refaktoryzacja polega na przepisywaniu procedur albo zbiorów procedur w celu nadania kodowi lepszej struktury. Stanowi ona jeden z aspektów programowania elastycznego, ponieważ adaptowalność kodu wymaga, by można było modyfikować go w miarę potrzeb.

Visual Studio 2005 będzie zawierać wiele funkcji ułatwiających proces refaktoryzacji. Będziesz na przykład mógł przekształcić dowolny fragment kodu w oddzielną procedurę przy użyciu menu kontekstowego. Funkcja ta ułatwi dzielenie dużych procedur oraz wyodrębnianie fragmentów programu w celu ich wielokrotnego wykorzystania.

Jednym z dostępnych już teraz elementów refaktoryzacji jest możliwość oznaczenia przestarzałych właściwości lub metod. Przypuśćmy, że mamy w aplikacji metodę (albo inną składową), która nie jest już potrzebna albo której sygnaturę trzeba zmienić. Mógłbyś

usunąć lub zmodyfikować metodę, a następnie wyszukać cały kod, który z niej korzysta. Lepiej będzie jednak oznaczyć metodę jako przestarzałą, tym samym deklarując, że nie należy jej już używać. Jeśli tak postąpisz, nie będziesz musiał modyfikować kodu korzystającego z metody.

Kod używający przestarzałej metody wygeneruje ostrzeżenie albo błąd składniowy (w zależności od sposobu oznaczenia metody) podczas następnej kompilacji. Dzięki temu możesz wycofywać kod z użycia w konsekwentny sposób. Jeśli na przykład budujesz komponenty na użytek innych programistów, firmowe standardy mogą wymagać, żeby przestarzała metoda przez sześć miesięcy generowała ostrzeżenie, a przez następne sześć — błąd składniowy, zanim zostanie ona ostatecznie usunięta z kodu.

Aby oznaczyć metodę jako przestarzałą, użyj atrybutu `ObsoleteAttribute`.

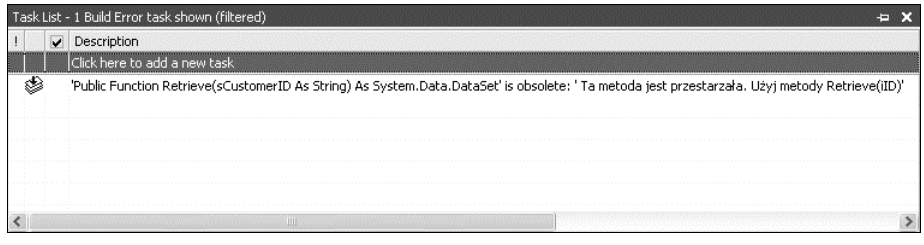
Visual Basic:

```
<ObsoleteAttribute("Ta metoda jest przestarzała. Użyj metody Retrieve(iID)", _
                   False)> _
Public Function Retrieve(ByVal sCustomerID As String) As DataSet
    ' Kod pobierający dane
End Function
```

C#:

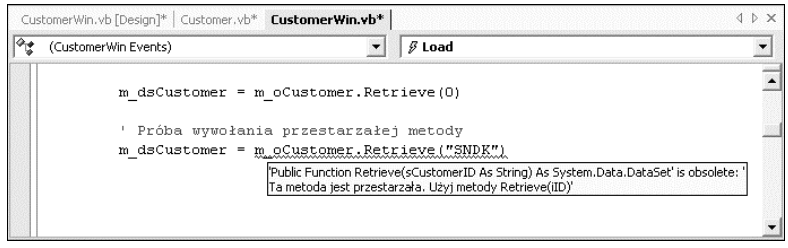
```
[ObsoleteAttribute("Ta metoda jest przestarzała. Użyj metody Retrieve(iID)", _
                   false)]
public DataSet Retrieve(string sCustomerID)
{
    // Kod pobierający dane
}
```

Pierwszym parametrem atrybutu `ObsoleteAttribute` jest komunikat, który zobaczy programista korzystający z przestarzałej metody. Tekst komunikatu będzie wyświetlony w oknie *Task List* (rysunek 3.6) oraz w podpowiedziach Quick Info (rysunek 3.7).



Rysunek 3.6. *Kiedy składowa klasy, na przykład metoda, jest oznaczona jako przestarzała, kod używający tej składowej zostanie wymieniony w oknie Task List*

Drugi parametr atrybutu `ObsoleteAttribute` określa, czy użycie przestarzałej składowej należy traktować jako błąd. Ustaw ten parametr na `False`, aby wyświetlać komunikat ostrzegawczy w oknie *Task List*, albo na `True`, aby użycie składowej powodowało błąd składniowy.



Rysunek 3.7. *Wszystkie odwołania do przestarzałej składowej są oznaczane w pliku kodu, a funkcja Quick Info wyświetla komunikat zdefiniowany w atrybucie ObsoleteAttribute*

Planuj pod kątem zmian. Zdefiniuj plan eliminacji przestarzałego kodu i używaj atrybutu `ObsoleteAttribute`, aby ułatwić sobie jego realizację.

Rozwijanie technik debugowania

Kiedy napiszesz kod, musisz sprawdzić, czy się kompiluje. Kiedy go skompilujesz, musisz sprawdzić, czy się uruchamia. Kiedy go uruchomisz, musisz sprawdzić, czy rzeczywiście wykonuje żądaną operację — to znaczy, czy realizuje założony cel. Jeśli kod działa, ale niezgodnie z oczekiwaniami, musisz skorzystać ze sprawdzonych technik wykrywania usterek. Ten proces określa się mianem **debugowania**.

Środowisko programistyczne Visual Studio oferuje wiele funkcji diagnostycznych, w tym takie, które z pewnością już znasz. Na początku niniejszego podrozdziału zostaną wymienione najpopularniejsze techniki debugowania, z których prawdopodobnie już korzystałeś. Dalej skupimy się na mniej znanych, ale przydatnych funkcjach diagnostycznych Visual Studio.

Przegląd podstawowych technik debugowania

Zanim przejdziesz do zagadnień, które być może nie są Ci znane, musisz wiedzieć, co już potrafisz. Dlatego niniejszy podrozdział zaczniemy od listy najpopularniejszych technik debugowania:

- przeglądanie listy błędów składniowych w oknie *Task List* (więcej informacji na ten temat znajdziesz w rozdziale 1.);
- ustawianie punktów wstrzymania;
- rozpoczynanie sesji diagnostycznej (uruchamianie programu w trybie *Debug*);
- krokowe wykonywanie kodu;
- zmiana punktu wykonywania;
- wyświetlanie wartości zmiennych w podpowiedziach ekranowych (dostępne tylko w trybie *Debug*);
- zatrzymywanie sesji diagnostycznej (powrót do trybu *Edit*).

Jeśli nie znasz którejkolwiek z tych technik, zajrzyj pod hasło „Debugging” w systemie pomocy Visual Studio.

Praca z oknem Watch

Być może wiesz już, jak używać okna *Watch* do obserwowania wartości zmiennych. Przydaje się ono jednak również do modyfikowania wartości zmiennych podczas debugowania kodu.

Aby wyświetlić okno *Watch*, wybierz z menu polecenie *Debug/Windows/Watch/Watch 1*, kiedy program uruchomiony w trybie *Debug* dotrze do punktu wstrzymania. Możesz jednocześnie korzystać z czterech oddzielnych okien *Watch*.

Okno *Watch* jest początkowo puste. Aby dodać do niego zmienną lub wyrażenie, wpisz je w kolumnie *Name*. Możesz również przeciągnąć nazwę zmiennej z pliku kodu do okna *Watch*. Rezultat pokazano na rysunku 3.8.

Rysunek 3.8. *Okno Watch wyświetla wartości zmiennych, sygnalizuje zmiany wartości i pozwala aktualizować zmienne*

Jeśli obserwowana zmienna lub wyrażenie znajdują się poza zasięgiem kodu zawierającego punkt wykonywania, ich wartości nie można ustalić. W takim przypadku kolumna *Value* w oknie *Watch* będzie zawierać odpowiedni komunikat, jak w przypadku zmiennej `sbMessage` na rysunku 3.8.

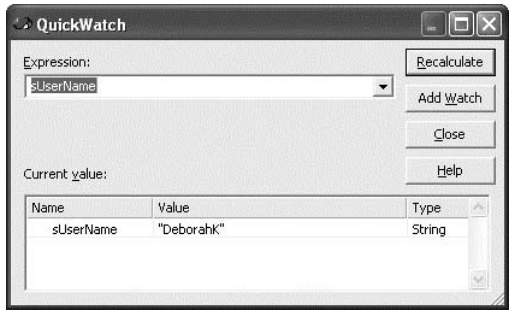
Modyfikując wartość dowolnej zmiennej będącej w zasięgu, kliknij kolumnę *Value* i wpisz nową wartość. Nie możesz zmieniać wartości wyrażeń wyświetlanych w oknie *Watch*.

Aby szybko przyjrzeć się wyrażeniu albo zaktualizować zmienną, której nie musisz obserwować, wyświetl okno dialogowe *QuickWatch*, wybierając z menu *Debug* polecenie *QuickWatch* (rysunek 3.9). Okno to wyświetlisz także poprzez kliknięcie zmiennej lub wyrażenia prawym przyciskiem myszy i wybranie polecenia *QuickWatch* z menu kontekstowego.

Okno dialogowe *QuickWatch* przypomina okno *Watch* pod tym względem, że pozwala obejrzeć wartość wyrażenia lub zmiennej oraz zaktualizować zmienną. Jest jednak modalne, więc nie możesz w nim śledzić wartości zmiennej lub wyrażenia podczas krokowego wykonywania kodu. Jeśli zdecydujesz, że chcesz obserwować zmienną lub wyrażenie, kliknij przycisk *Add Watch* w oknie *QuickWatch*, aby automatycznie dodać wpis do okna *Watch*.

Używaj okna *Watch* lub okna dialogowego *QuickWatch*, aby oglądać wartości zmiennych lub wyrażeń albo aktualizować zmienne podczas debugowania.

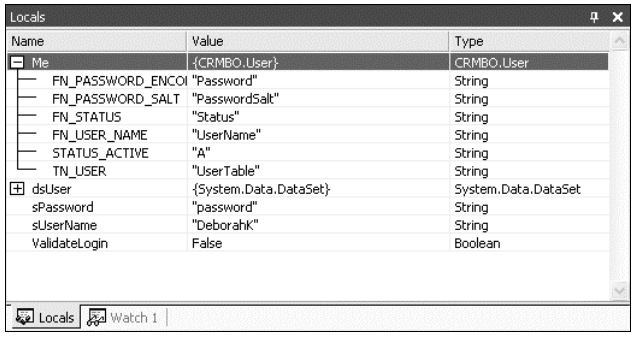
Rysunek 3.9. *Okno dialogowe QuickWatch wyświetla wartość zmiennej lub wyrażenia i pozwala zaktualizować wartość zmiennej*



Praca z oknem Locals

Okno *Locals* (rysunek 3.10) umożliwia oglądanie wszystkich zmiennych lokalnych procedury, w której znajduje się punkt wykonywania. Przypomina ono okno *Watch*, ale jego zawartość jest wstępnie zdefiniowana. Nie możesz do tego okna dodawać wpisów ani usuwać ich z niego.

Rysunek 3.10. *Okno Locals wyświetla wartości wszystkich zmiennych lokalnych procedury, w której znajduje się punkt wykonywania*



Użyj okna *Locals*, aby obejrzeć lub zaktualizować wartości zmiennych bez dodawania ich do okna *Watch*.

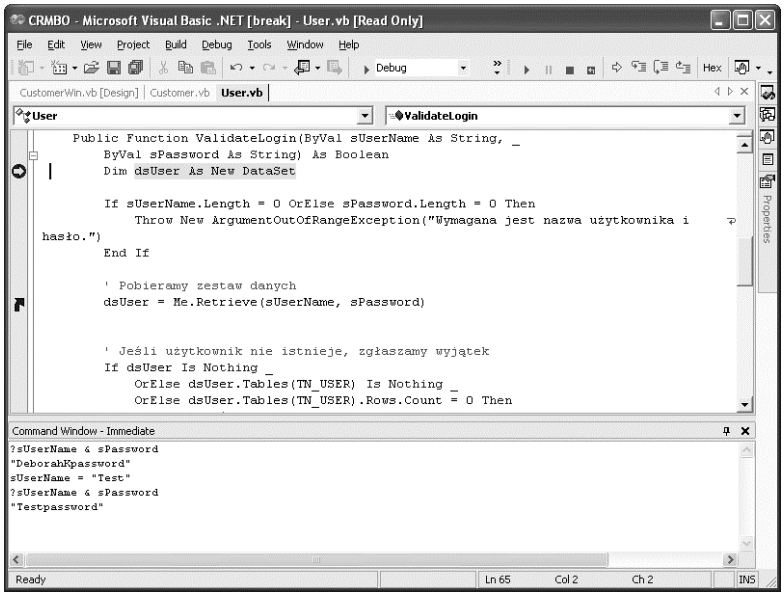
Praca z oknem Command

Okno *Command* jest jednym z najbardziej przydatnych okien diagnostycznych dostępnych w Visual Studio. Może ono działać w dwóch trybach: *Immediate* i *Command*. W trybie *Immediate* pozwala ono oglądać wartości zmiennych, przypisywać wartości zmiennym oraz obliczać wyrażenia. W trybie *Command* umożliwia wykonywanie poleceń Visual Studio.

Aby otworzyć okno *Command* w trybie *Command*, wybierz z menu *View* pozycję *Other Windows*, a następnie polecenie *Command Window*. Znak większości (>) zasygnalizuje, że okno znajduje się w trybie *Command*. Chcąc przełączyć się z trybu *Command* do *Immediate*, powinieneś wpisać polecenie `>immed`.

Kiedy okno *Command* znajduje się w trybie *Command*, możesz wydawać polecenia Visual Studio, wpisując polecenie i naciskając klawisz *Enter*. Więcej informacji o poleceniach Visual Studio znajdziesz w rozdziale 1.

Aby otworzyć okno *Command* w trybie *Immediate*, wybierz z menu *Debug* pozycję *Windows*, a potem polecenie *Immediate*. Na pasku tytułu okna *Command* pojawi się informacja, że działa ono w trybie *Immediate* (rysunek 3.11). Aby wydać polecenie Visual Studio w trybie *Immediate*, poprzedź je znakiem większości (>). Aby przełączyć się z trybu *Immediate* do *Command*, wpisz polecenie `>cmd`.



Rysunek 3.11. *Kiedy okno Command znajduje się w trybie Immediate, masz możliwość wyświetlania albo modyfikowania wartości zmiennych i obliczania wyrażeń*

W oknie *Command* pozostającym w trybie *Immediate*, możesz wpisać znak zapytania (?) oraz nazwę zmiennej lub wyrażenie, a następnie nacisnąć klawisz *Enter*, aby obejrzeć wartość zmiennej lub wyrażenia bez potrzeby otwierania okna *Watch* czy *Locals*. Żeby zmodyfikować zmienną, wpisz jej nazwę oraz żądany argument i naciśnij klawisz *Enter*. Wynik tych operacji pokazano na rysunku 3.11.

Możesz w dowolnym momencie opróżnić okno *Command*, klikając je prawym przyciskiem myszy i wybierając polecenie *Clear All* z menu kontekstowego.

Praca z punktami wstrzymania

Każdy programista .NET wie, że punkt wstrzymania można ustawić poprzez kliknięcie lewego marginesu w oknie kodu. Wielu nie odkryło jednak sekretów ukrytych w oknie dialogowym *New Breakpoint*. To niezwykle przydatne okno oferuje liczne opcje, które znacznie zwiększają efektywność debugowania.

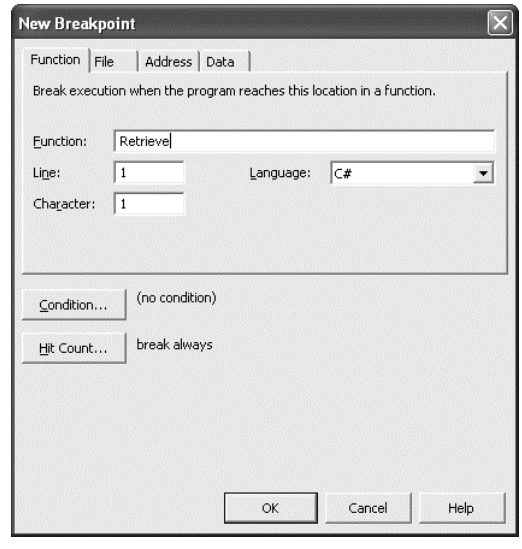
Na pierwszy rzut oka mogłoby się wydawać, że polecenie *New Breakpoint* w menu *Debug* służy po prostu do ustawiania punktu wstrzymania w kodzie. Otwiera ono jednak okno dialogowe *New Breakpoint*.

Jeśli używasz C#, możesz kliknąć okno kodu prawym przyciskiem myszy i wybrać z menu kontekstowego polecenie *New Breakpoint*, a otworzysz okno dialogowe *New Breakpoint*.

Okno dialogowe *New Breakpoint* pozwala ustawić cztery typy punktów wstrzymania: *Function*, *File*, *Address* i *Data*. Zostaną one omówione w kolejnych punktach wraz z kilkoma innymi cechami punktów wstrzymania.

Ustawianie punktów wstrzymania w funkcjach

Karta *Function* w oknie dialogowym *New Breakpoint* (rysunek 3.12) pozwala wstrzymać wykonywanie aplikacji na pierwszym wierszu wybranej funkcji.



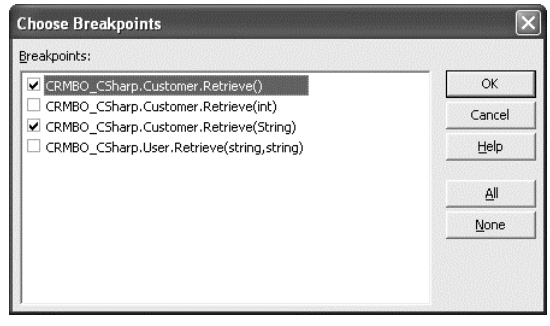
Rysunek 3.12. *Karta Function w oknie dialogowym New Breakpoint, dzięki której wstrzymasz wykonywanie programu na początku wybranej funkcji*

Choć wydaje się, że możesz wybrać konkretne miejsce funkcji przy użyciu pól tekstowych *Line* i *Character* w oknie dialogowym *New Breakpoint*, to nie da się ustawić wiersza (*Line*) innego niż pierwszy, a ustawienie pola *Character* jest ignorowane.

Ustawiając punkt wstrzymania w wybranej funkcji, wpisz jej nazwę w oknie dialogowym. Możesz na przykład wpisać `Retrieve` w celu ustawienia punktu wstrzymania w funkcji `Retrieve`. Opcjonalnie możesz poprzedzić nazwę funkcji nazwą klasy w formacie *`klasa.funkcja`*. Wpisz `User.Retrieve`, aby ustawić punkt wstrzymania w funkcji `Retrieve` w klasie `User`.

W przypadku C# nazwa funkcji wpisana w oknie dialogowym *New Breakpoint* musi pasować pod względem wielkości liter do nazwy funkcji w pliku kodu.

Jeśli w rozwiązaniu istnieje więcej niż jedna funkcja o takiej samej nazwie, pojawi się okno dialogowe *Choose Breakpoints* (rysunek 3.13). Wybierz jedną lub kilka wersji funkcji, aby ustawić w nich punkty wstrzymania.



Rysunek 3.13. *Przy użyciu okna dialogowego Choose Breakpoints możesz ustawić punkty wstrzymania w kilku funkcjach jednocześnie*

Przyciski *Condition* i *Hit Count* w oknie dialogowym *New Breakpoint* pozwalają określić dodatkowe ograniczenia punktów wstrzymania. Ponieważ są one dostępne w przypadku wszystkich typów punktów wstrzymania, zostaną omówione oddzielnie w dalszej części rozdziału.

Kiedy klikniesz przycisk *OK* w oknie dialogowym *New Breakpoint*, punkt wstrzymania zostanie wstawiony na początku każdej z określonych funkcji, o czym będą świadczyć czerwone kółka na lewym marginesie w odpowiednich plikach kodu.

Karta *Function* okna dialogowego *New Breakpoint* przydaje się szczególnie do ustawienia punktów wstrzymania we wszystkich przeciążonych wersjach wybranej funkcji, bez potrzeby lokalizowania każdej z nich.

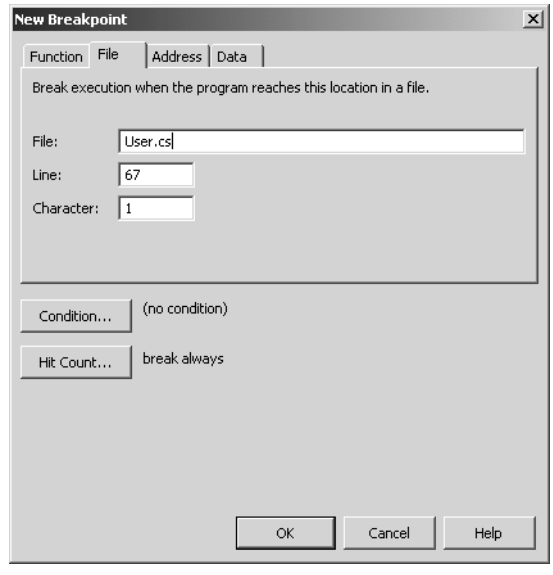
Ustawianie punktów wstrzymania w plikach

Plikowe punkty wstrzymania to te same, które ustawiasz poprzez kliknięcie lewego marginesu pliku kodu. Na marginesie pojawia się wówczas czerwone kółko. Punkty te możesz również ustawić przy użyciu karty *File* okna dialogowego *New Breakpoint* (rysunek 3.14).

Punkt wstrzymania w wybranym pliku ustawisz, wpisując nazwę pliku w oknie dialogowym. Możesz na przykład wpisać `user.cs`, aby ustawić punkt wstrzymania w pliku *User.cs*. Możesz również określić numer wiersza (*Line*) i położenie znaku (*Character*).

Opcja *Character* na karcie *File* przydaje się tylko wtedy, gdy masz wiele instrukcji w tym samym wierszu pliku kodu.

Kiedy klikniesz przycisk *OK* w oknie dialogowym *New Breakpoint*, punkt wstrzymania zostanie ustawiony przy określonym wierszu i znaku w pliku kodu.



Jeśli wiersz o podanym numerze nie jest dozwoloną lokalizacją punktu wstrzymania, zostaniesz o tym powiadomiony po kliknięciu przycisku *OK* w oknie dialogowym *New Breakpoint*. Jeśli podasz nieprawidłowe położenie znaku, punkt wstrzymania zostanie ustawiony przy najbliższym dozwolonym znaku.

Plikowe punkty wstrzymania są najczęściej używaną funkcją diagnostyczną. Stają się jeszcze użyteczniejsze, gdy wykorzystuje się je w połączeniu z funkcjami *Conditions* i *Hit Count*, omówionymi w dalszej części rozdziału.

Ustawianie punktów wstrzymania przy adresie

Adresowe punkty wstrzymania są bardziej zaawansowaną funkcją. Pozwalają one wstrzymać wykonywanie programu przy określonym adresie w pamięci (rysunek 3.15).

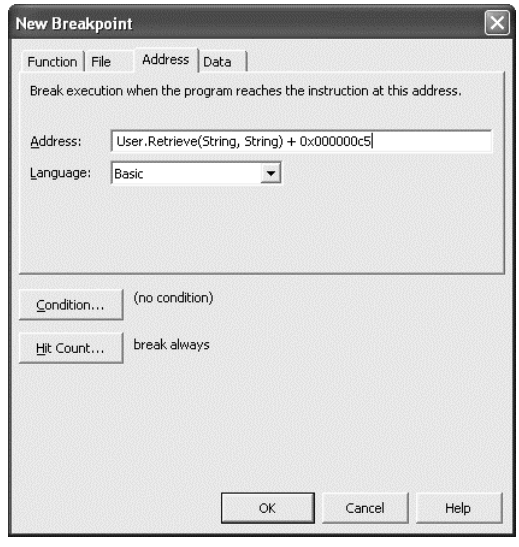
Większość programistów nie potrzebuje tej funkcji, ale jeśli jesteś nią zainteresowany, zajrzyj do pomocy Visual Studio.

Ustawianie punktów wstrzymania zależnych od danych

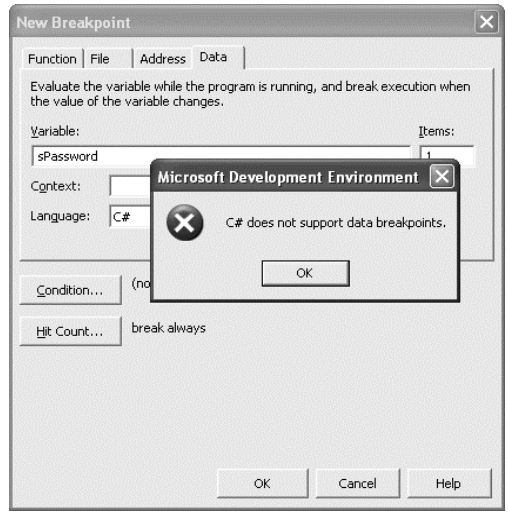
Punkty wstrzymania zależne od danych mogłyby być niezwykle przydatne. Powodują one wstrzymanie wykonywania programu, kiedy modyfikowana jest wartość zmiennej, bez względu na położenie w pliku kodu. Zła wiadomość jest taka, że ani Visual Basic, ani C# nie obsługują tego typu punktów wstrzymania (rysunek 3.16).

Warunkowe punkty wstrzymania

Do każdego punktu wstrzymania (niezależnie od jego typu) możesz dodać warunek logiczny, klikając przycisk *Condition* w oknie dialogowym *New Breakpoint*. Pojawi się okno dialogowe *Breakpoint Condition* (rysunek 3.17), w którym możesz ustalić, że program powinien zostać wstrzymany tylko w razie spełnienia pewnego warunku.



Rysunek 3.15. *Karta Address w oknie dialogowym New Breakpoint pozwala ustawić punkt wstrzymania przy wybranym adresie w pamięci*



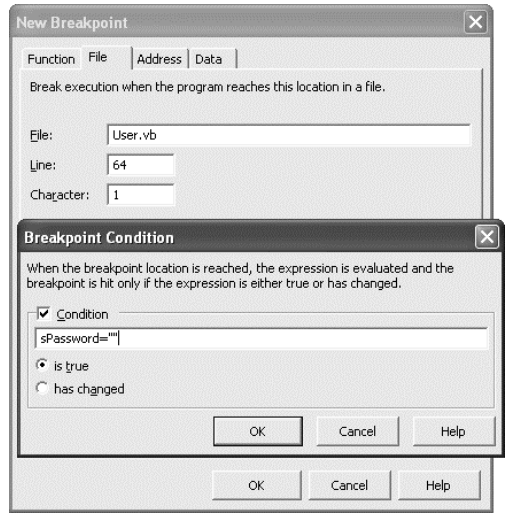
Rysunek 3.16. *Punkty wstrzymania zależne od danych nie są dostępne w Visual Basicu ani w C#*

W oknie dialogowym *Breakpoint Condition* wpisz żądane wyrażenie, a następnie zdefiniuj, czy program ma być wstrzymywany, gdy wyrażenie ma wartość `True`, czy też gdy jego wartość się zmienia.

Warunkowe punkty wstrzymania są szczególnie pomocne, gdy chcesz, aby program był wstrzymywany w zależności od określonej wartości zmiennej lub wyrażenia. Mógłbyś, na przykład, wstrzymać program po dotarciu do określonego elementu w pętli `For Each`.

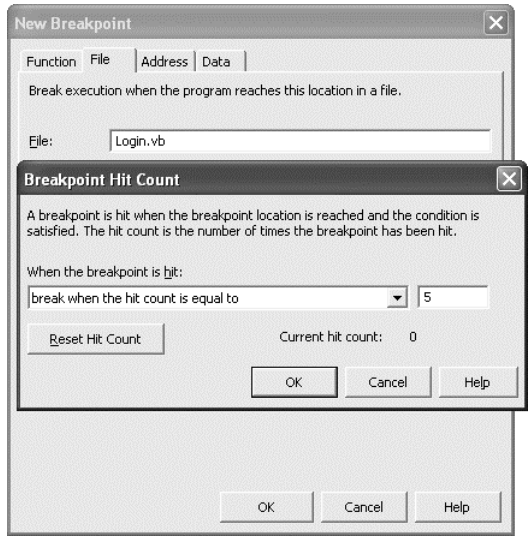
Punkty wstrzymania z liczbą trafień

Liczba trafień (ang. *hit count*) określa, ile razy punkt wstrzymania został osiągnięty podczas sesji debugowania. Aby dodać liczbę trafień do punktu wstrzymania, kliknij



Rysunek 3.17. *Okno dialogowe Breakpoint Condition umożliwia zdefiniowanie warunku wstrzymania. W tym przykładzie program zostanie wstrzymany tylko wtedy, gdy miejsce na hasło będzie puste*

przycisk *Hit Count* w oknie dialogowym *New Breakpoint*. Pojawi się okno dialogowe *Breakpoint Hit Count* (rysunek 3.18). Możesz w nim zdefiniować, że program ma zostać wstrzymany dopiero wtedy, gdy punkt wstrzymania zostanie osiągnięty określoną ilość razy.



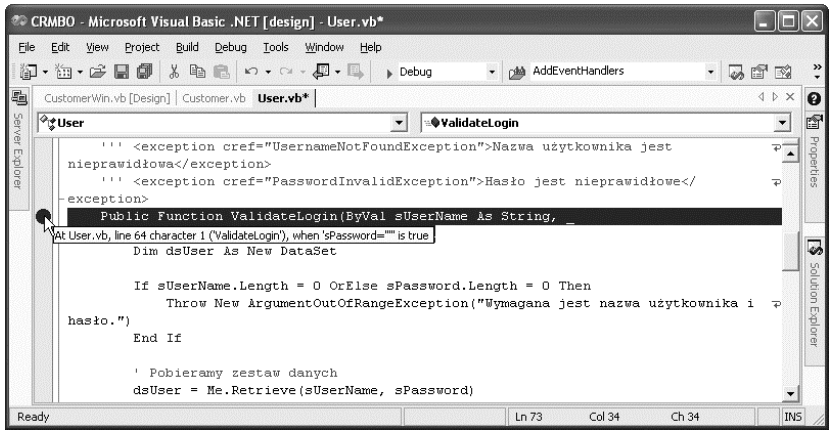
Rysunek 3.18. *Okno dialogowe Breakpoint Hit Count pozwala zdefiniować, ile razy trzeba osiągnąć punkt wstrzymania przed przerwaniem wykonywania programu*

W oknie dialogowym *Breakpoint Hit Count* wskaż, czy program ma być wstrzymywany zawsze (*break always*), przy określonej liczbie trafień (*break when the hit count is equal to*), przy określonej lub wyższej liczbie trafień (*break when the hit count is greater than or equal to*), czy określonej wielokrotności trafień (*break when the hit count is a multiple of*). Okno to pozwala również wyzerować bieżącą liczbę trafień za pomocą przycisku *Reset Hit Count*.

Określanie liczby trafień okazuje się pomocne wtedy, gdy masz pętlę i nie chcesz, aby była przerywana przy każdym przebiegu.

Edycja właściwości punktów wstrzymania

Ustawiony punkt wstrzymania pojawia się jako czerwone kółko na lewym marginesie obok instrukcji, na której program zostanie wstrzymany. Przytrzymaj wskaźnik myszy nad tym kółkiem, aby obejrzeć właściwości punktu wstrzymania (rysunek 3.19).



Rysunek 3.19. *Przytrzymaj wskaźnik myszy nad ikoną punktu wstrzymania, aby obejrzeć właściwości punktu*

Jeżeli masz zamiar zmienić właściwości punktu wstrzymania, kliknij prawym przyciskiem myszy wiersz zawierający punkt wstrzymania i wybierz polecenie *Breakpoint Properties*. Pojawi się okno dialogowe *New Breakpoint* z tytułem zmienionym na *Breakpoint Properties*. W oknie tym możesz zmodyfikować wszystkie właściwości punktu wstrzymania.

Stan punktu wstrzymania

Kiedy ustawiasz punkt wstrzymania, jest on włączony, co oznacza, że wykonywanie kodu zostanie wstrzymane po osiągnięciu wiersza zawierającego punkt. Oto inne stany punktów wstrzymania:

- **Wyłączony** — możesz tymczasowo wyłączyć punkty wstrzymania, wybierając z menu *Debug* polecenie *Disable All Breakpoints*. Punkty przerwania będą wówczas ignorowane. Wyłączone punkty wstrzymania są wyświetlane jako czerwone, puste kółka.
- **Ponownie włączony** — aby ponownie włączyć punkty wstrzymania, wybierz z menu *Debug* polecenie *Enable All Breakpoints*. Włączone punkty wstrzymania są wyświetlane jako czerwone, pełne kółka.
- **Błąd** — ikona wykrzyknika pojawia się, kiedy nie da się ustawić punktu w określonym wierszu. Zwykle dzieje się tak wtedy, gdy spróbujesz ustawić punkt wstrzymania w niedozwolonym miejscu albo zdefiniujesz nieprawidłowy warunek.

- **Ostrzeżenie** — ikona znaku zapytania pojawia się, kiedy nie można ustawić punktu wstrzymania, ponieważ kod nie jest wczytany. Jeśli kod zostanie wczytany podczas wykonywania programu, punkt wstrzymania będzie włączony.
- **Usunięty** — aby usunąć wszystkie punkty wstrzymania z aplikacji, wybierz z menu *Debug* polecenie *Clear All Breakpoints*. Aby usunąć pojedynczy punkt wstrzymania, kliknij jego ikonę na lewym marginesie.

Biegłe posługiwanie się punktami wstrzymania sprawi, że debugowanie programów będzie znacznie szybsze i efektywniejsze.

Co opisano w tym rozdziale?

Możesz zwiększyć czytelność, adaptowalność i wydajność kodu dzięki znajomości kilku prostych trików. Niektóre z nich zostały opisane w tym rozdziale.

Żeby zwiększyć wydajność programów, używaj warunkowych operatorów logicznych i skróconych operatorów przypisania oraz manipuluj łańcuchami przy użyciu klasy `StringBuilder`. Deklaruj zmienne pętli przy instrukcjach `For` albo `For Each`, aby miały one lokalny zasięg. Posługuj się ścisłą konwersją typów, korzystaj z ulepszonego rzutowania i twórz aliasy typów danych, by później łatwo je zmodyfikować.

Użyj wyrażeń regularnych do sprawdzania poprawności danych wprowadzanych przez użytkownika. Przeciążaj metody, aby przekazywać do nich różne zbiory parametrów, oraz operatory, aby przeprowadzać operacje matematyczne i logiczne na własnych typach danych. Usprawnij zarządzanie kodem dzięki regionom, komentarzom XML i oznaczaniu przestarzałego kodu.

Wykorzystaj też do maksimum funkcje diagnostyczne Visual Studio, przez co rozwiniesz swoje techniki debugowania i będziesz mógł w najefektywniejszy sposób usuwać usterki z programów.

W następnym rozdziale zajmiemy się tajnikami ADO.NET oraz pracy z danymi.