Visual Basic .NET. Księga eksperta

Autor: Paul Kimmel
Tłumaczenie: Krzysztof Jurczyk, Marek Pałczyński
ISBN: 83-7197-771-9
Tytuł oryginału: Visual Basic .NET Unleashed
Format: B5, stron: 682
Przykłady na ftp: 3044 kB

Visual Basic przeszedł generalny remont. Istnieje wiele powodów, dla których programiści Visual Basica 6 powinni przesiąść się na nowy Visual Basic .NET. Należy do nich zaliczyć chociażby formularze Web, możliwość tworzenia aplikacji i usług WWW, strukturalną obsługę wyjątków, prawdziwe programowanie zorientowane obiektowo czy też wielowątkowość.

„Visual Basic .NET. Księga eksperta" zawiera dokładne omówienie nowego języka Visual Basic .NET, zunifikowanego środowiska programowania Visual Studio IDE, programowania formularzy WWW, ADO.NET, usługi WWW, GDI+ i wiele innych.

Visual Studio .NET jest środowiskiem bardzo rozbudowanym i potężnym. Aby w pełni je wykorzystać, poznasz także sposoby tworzenia makr oraz znajdziesz omówienie modelu automatyzacji służącego do indywidualizacji zadań i interfejsu IDE w Visual Studio. Książka zawiera wiele przykładów wziętych z praktyki programistycznej. Książka omawia:

- Środowisko programistyczne Visual Studio, korzystanie z SourceSafe
- Język Visual Basic .NET, programowanie zorientowane obiektowo w VB .NET
- Rozszerzanie środowiska programistycznego za pomocą makr
- Zaawansowane programowanie w VB .NET: refleksje, przeciążanie, programowane oparte na zdarzeniach, polimorfizm, definiowanie atrybutów
- Tworzenie interfejsu użytkownika (aplikacje konsolowe, aplikacje z interfejsem Windows)
- Pisanie aplikacji wielowątkowych
- Uruchamianie usług WWW (Web Services)

„Visual Basic .NET. Księga eksperta" jest doskonałym podręcznikiem dla wszystkich osób, dla których osiągnięcie wysokiej sprawności w posługiwaniu się językiem Visual Basic stanowi podstawę kariery programistycznej. Niezależnie, od tego, czy używałeś poprzedniej wersji tego języka, czy też nie: jeśli chcesz stać się ekspertem Visual Basica, trzymasz w ręku odpowiednią książkę.

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

Spis treści

Rozdział 7.
Tworzenie klas

Możliwość definiowania klas i tworzenia ich egzemplarzy (instancji) jest jedną z najważniejszych cech każdego języka zorientowanego obiektowo. Aż do chwili obecnej Visual Basic nie obsługiwał w pełni obiektowości. Mimo że moduły w VB6 nazywane były **modułami klas**, tak naprawdę były **interfejsami** w sensie technologii COM (ang. *Component Object Model*). Oznacza to, że VB6 nie obsługiwał idiomu klasy; nie umożliwiał korzystania z innej bardzo pożytecznej cechy programowania obiektowego — dziedziczenia.

Zasadniczą różnicą pomiędzy interfejsami, a klasami jest dziedziczenie. Każda implementacja interfejsu VB6 (modułu klasy) wymagała wprowadzenia wszystkich publicznych metod dla tego interfejsu. Klasy, w przeciwieństwie do interfejsu, obsługują dziedziczenie. **Dziedziczenie** oznacza, że mogą istnieć podklasy zawierające pola, właściwości, metody i zdarzenia klasy nadrzędnej. Tworząc nowe klasy, możesz je wykorzystywać do rozszerzania istniejących klas bazowych.

Zarówno zorientowane obiektowo klasy, jak i interfejsy COM udostępniają twórcom oprogramowania potężne marzędzia do tworzenia i zarządzania zaawansowanymi projektami. Obie technologie są bardzo przydatne i obie zostały zaimplementowane w Visual Basic .NET. Dodatkowo, Microsoft udoskonalił i poprawił interfejsy i opisy klas w celu zapobiegnięcia niejednoznaczności podczas ich stosowania. Obie technologie posiadają odmienne zasady składni.

W tym rozdziale zostanie przedstawiona zmieniona składnia interfejsów oraz właściwości nowej definicji klasy, włączając w to dziedziczenie, polimorfizm, przeciążanie i przesłanianie metod. Dodatkowo, rozdział ten zawiera kolejne przykłady wykorzystania obsługi wyjątków i wielowątkowości w Visual Basic .NET.

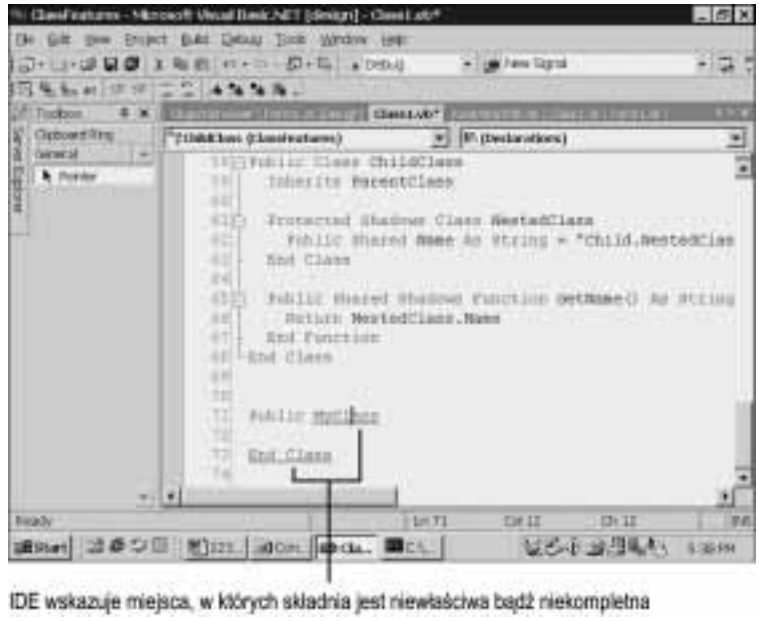
Definiowanie klas

Klasa w Visual Basic .NET nie jest klasą VB6. Klasy VB6 są w VB .NET deklarowane i definiowane przy użyciu słowa kluczowego `interface`. Klasy VB .NET są definiowanymi przez użytkownika typami agregacyjnymi, umożliwiającymi dziedziczenie. Różnią się one od interfejsów COM, które są dokładnie tym, czym w VB6 były moduły klasy.

Wszystkie klasy w Visual Basic .NET są definiowane w pliku .VB (w przeciwieństwie do pliku .CLS), a w pliku takim może być zawarta jedna lub kilka klas. (Klasy, struktury i moduły mogą być zgromadzone w jednym pliku). Podstawowa składnia klasy jest następująca:

```
Class classname
End Class
```

Jeśli przez przypadek popełnisz błąd podczas deklarowania klasy, Visual Studio .NET IDE podkreśli linią falistą miejsce, w którym ten błąd występuje (zobacz rysunek 7.1)



Rysunek 7.1. *Visual Studio .NET IDE zaznacza linią falistą miejsca zawierające błąd*

Najczęściej klasy poprzedzone są specyfikatorem dostępu `Public`, możesz jednak wykorzystać inne specyfikatory. W podrozdziale „Używanie specyfikatorów dostępu do klas" znajdziesz więcej informacji na ten temat.

Instrukcje `Class` i `End Class` tworzą jednostkę hermetyzacji dla klasy. Wszystkie składowe klasy umieszczane są pomiędzy tymi instrukcjami.

Możesz dodawać dowolną liczbę pól, właściwości, metod i zdarzeń w celu zdefiniowania klasy, która ma spełniać określone wymagania w dziedzinie rozwiązania jakiegoś problemu. Jakkolwiek liczba składowych w klasie może być dowolna, jednak przestrzeganie następujących zasad w większości przypadków pozwoli na zachowanie przejrzystości w kodzie i uproszczenie tworzonych konstrukcji.

- Klasy powinny składać się z maksymalnie sześciu składowych publicznych, włączając w to właściwości i metody. Liczba składowych niepublicznych może być większa, gdyż nie wpływają one na prostotę korzystania z klasy przez użytkownika.
- W ogólnym przypadku jakiekolwiek składowe niepubliczne powinny być chronione i wirtualne — posiadając modyfikator `overridable`.

Pola, właściwości, zdarzenia wspólnie określane są mianem **składowych**. Termin **składowa** odnosi się po prostu do czegoś, co zostało zdefiniowane jako część klasy.

Jak we wszystkich regułach, także i tu są wyjątki. Istnieje zasada „wystarczająco dobry", wprowadzona przez Grady Boocha, mówiąca, że jeśli różnorodność oraz liczba składowych jest rozsądna i wystarczająca, to liczba ta jest wystarczająco dobra. W książce Jamesa Coplien „Zaawansowany C++" zostały przedstawione zaawansowane zagadnienia, które łamią ogólne zasady. Kolejnym przykładem wyjątku od opisanych powyżej reguł są klasy ze wszystkimi składowymi typu `Shared`, jak np. klasa `Math` w CLR.

Jak zostało wspomniane wcześniej, podstawowe wskazówki do definiowania klas są dobrym punktem startowym, istnieją jednak inne zasady i motywacje, którymi można się kierować podczas tworzenia klas. Na przykład technika refaktoringu „Zastosuj obiekt parametryczny", której zastosowanie ulepsza przekazywanie parametrów, wymaga utworzenia dodatkowych klas.

Używanie specyfikatorów dostępu do klas

W Visual Basic .NET występuje pięć specyfikatorów dostępu, opisanych w kolejnych podrozdziałach.

Public

Specyfikator dostępu `Public`, zastosowany na poziomie klasy, jest najczęściej używanym specyfikatorem. Klasy publiczne są to klasy, do których przewidziany jest dostęp przez każdego użytkownika. Specyfikator dostępu `Public` występuje w kodzie po atrybutach i bezpośrednio przed słowem kluczowym `Class`. (W rozdziale 12., „Definiowanie atrybutów", znajdziesz więcej informacji o atrybutach).

```
Public Class MyClass
End Class
```

Protected

Specyfikator `Protected` ma zastosowanie wyłącznie w klasach zagnieżdżonych. Klasy zagnieżdżone są są dostępne wyłącznie w danej klasie i w klasach potomnych. Nie możesz zdefiniować składników egzemplarza klasy zagnieżdżonej z widocznością większą niż definicja klasy.

Definicja zagnieżdżonej klasy `ProtectedClass` przedstawia się następująco:

```
Public Class HasProtected
  Protected Class ProtectedClass
  End Class
End Class
```

Klasa `HasProtected` zawiera zagnieżdżoną klasę `ProtectedClass`. `HasProtected` może implementować egzemplarze `ProtectedClass`, a klasy dziedziczące z `HasProtected` mogą deklarować i tworzyć egzemplarze zagnieżdżonej klasy `ProtectedClass`.

Friend

Klasy zaprzyjaźnione są dostępne wyłącznie w programie, w którym zostały zdefiniowane. Jeśli dodasz specyfikator dostępu `Friend` do definicji klasy, egzemplarze tej klasy mogą być tworzone wyłącznie w tym samym programie.

Załóżmy, że mamy bibliotekę klas z klasą o trybie dostępu `Friend` o nazwie `FriendClass`. Moglibyśmy utworzyć egzemplarze tej klasy w bibliotece klas, lecz nie możemy uczynić tego z poziomu użytkownika tej biblioteki. Można zatem powiedzieć, że `FriendClass` jest wyłącznie do użytku wewnętrznego. Oto przykład:

```
Friend Class FriendClass
  Public Shared Sub PublicMethod()
    MsgBox("FriendClass.PublicMethod()")
  End Sub
End Class
```

Korzystaj ze specyfikatora dostępu `Friend`, gdy chcesz utworzyć klasę zawierającą detale implementacyjne biblioteki klas lub jakiejś innej aplikacji, uniemożliwiając tym samym dostęp do takiej klasy użytkownikom danej aplikacji.

Protected Friend

Klasy `Protected Friend` reprezentują połączenie specyfikatorów `Protected` i `Friend`. Klasy chronione muszą być zagnieżdżone; z tego względu klasy `Protected Friend` również muszą być zagnieżdżone. Przykład pokazuje zagnieżdżoną klasę `Protected Friend`:

```
Public Class HasProtectedFriend

  Protected Friend Class ProtectedFriend
    Public Shared Sub Test()
      MsgBox("Text")
    End Sub
  End Class

End Class
```

Klasy `Protected Friend` są najczęściej wykorzystywane jako klasy realizacyjne dla klasy, która je zawiera. Metody definiowane w klasach zagnieżdżonych mogą być wywoływane pośrednio przez metody proxy zawierającej je klasy. Nie możesz zwrócić egzemplarza klasy `Protected Friend`. Poniższy przykład, bazując na kodzie powyżej, jest więc niepoprawny:

```
Public Class HasProtectedFriend

  Protected Friend Class ProtectedFriend
    Public Shared Sub Test()
      MsgBox("Text")
    End Sub
  End Class

  Public Shared Function Factory() As ProtectedFriend

  End Function

End Class
```

Taka definicja funkcji spowoduje wystąpienie błędu, wyraz `ProtectedFriend` zostanie podkreślony linią falistą, a w liście zadań pojawi się uwaga, że `'Factory'` niewłaściwie udostępnia typ `ProtectedFriend` poza klasą `publiczną 'HasProtectedFriend'`.

Private

Specyfikator dostępu `Private` ma zastosowanie wyłącznie w klasach zagnieżdżonych. Klasy zagnieżdżone `Private` reprezentują szczegóły implementacyjne danej klasy. Gdy pojawi się wewnętrzny problem, którego rozwiązanie jest bardziej złożone i wymaga mocy, jakiej nie są w stanie zapewnić proste metody, definiuje się wówczas zagnieżdżoną klasę prywatną, która rozwiązuje ten problem. Przykład składni takiej klasy wygląda następująco:

```
Public Class HasPrivate
  Private Class PrivateClass
  End Class
End Class
```

Egzemplarze `PrivateClass` mogą być tworzone wyłącznie w egzemplarzach `HasPrivate`. Cel korzystania z klasy prywatnej pojawia się wówczas, gdy mamy grupę metod i właściwości i musimy przechować wiele egzemplarzy stanu każdego z tych obiektów. Różnica polega na tym, że nie chcemy, aby prywatne klasy zagnieżdżone były widoczne dla użytkownika z zewnątrz.

Stosunkowo rzadko w swych aplikacjach będziesz wykorzystywał klasy chronione i prywatne. Miej jednak na uwadze, że takie konstrukcje istnieją i w razie potrzeby nie bój się ich użyć

Hermetyzacja i ukrywanie informacji

Hermetyzacja i ukrywanie informacji to zagadnienia bezpośrednio związane z programowaniem zorientowanym obiektowo; mogłeś się z nimi spotkać już wcześniej. Terminy te dotyczą strategii programowania. **Hermetyzacja** literalnie oznacza dodawanie składowych — pól, właściwości, metod i zdarzeń — do klas lub struktur. Ogólną zasadą jest to, że składowe dodaje się w tych miejscach, w których ich pojawienie się będzie najbardziej korzystne; to znaczy, gdzie będą mogły być najczęściej wykorzystywane lub zapewnią najlepsze udoskonalenie Twojej implementacji.

Konsument klasy jest to programista, który tworzy egzemplarze klasy. Twórca klasy (producent) może być również jednocześnie konsumentem.

Generalizer jest konsumentem, który będzie dziedziczył Twoje klasy.

Ukrywanie informacji jest to przypisywanie egzemplarzom określonych możliwości dostępu — mogą one być prywatne, chronione, publiczne i zaprzyjaźnione — w celu uwolnienia konsumentów lub generalizatorów od konieczności poznawania wszystkich składowych danej klasy lub struktury. Ograniczając liczbę składowych klasy, których muszą się oni nauczyć — poprzez właśnie ukrycie informacji — tworzone klasy są łatwiejsze do wykorzystania.

Specyfikatory dostępu

Do ukrywania informacji wykorzystuje się następujące specyfikatory dostępu: `Private`, `Protected`, `Public`, `Friend` oraz `Shadows` (tymi ostatnimi zajmiemy się w dalszej części rozdziału).

Kod klasy, który chcesz, aby był dostępny dla wszystkich, jest określany jako `Public`. Mówimy, że interfejs publiczny jest utworzony ze składowych publicznych. Składowe `Public` są implementowane dla potrzeb konsumentów klas. Gdy chcesz natomiast ukryć część informacji przed konsumentami publicznymi, jednak generalizatorzy mają mieć możliwość dostępu do tej informacji, zastosuj specyfikator dostępu `Protected`.

Składowe `Private` służą wyłącznie do użytku wewnętrznego i mówimy, że zawierają szczegóły implementacyjne klasy. Ze szczegółami tymi jedynie producent klasy musi być zaznajomiony.

Ostatnie ze specyfikatorów — `Friend` oraz `Protected Friend` — używane są w tych przypadkach, w których klasy wykorzystywane są przez jednostki w tej samej aplikacji. Dostęp zaprzyjaźniony oznacza dostęp wyłącznie wewnątrzaplikacyjny. Na przykład, jeśli definiujesz bibliotekę klas, a użytkownicy z zewnątrz nie muszą lub nie powinni mieć dostępu do niektórych klas z tej biblioteki, oznaczasz je jako klasy `Friend`.

Ogólna zasada jest taka, że należy wszystkie metody definiować jako `Protected`, chyba że muszą być publiczne. Jeśli musisz ujawniać składowe, rób to bardzo rozsądnie i rozmyślnie.

Praca z zasięgiem

Zagadnienie zasięgu zostało rozszerzone w Visual Basic .NET o zasięg blokowy. Nowe reguły rządzące zasięgami zostały przedstawione w rozdziale 3., „Podstawy programowania w Visual Basic .NET". Oprócz istniejących zasad i tych związanych z nowym zasięgiem blokowym, pojawił się nowy aspekt pracy — wsparcie dla dziedziczenia, wprowadzające dodatkowe względy, na które należy zwracać uwagę.

Gdy dziedziczysz z istniejącej klasy, Twoja nowa klasa posiada w swoim zasięgu wszystkie z odziedziczonych składników — `Protected`, `Friend` i `Public`. Na szczęście, Visual Basic .NET nie pozwala na pojawienie się problemów z nazwami tych składników; gdy wystąpi konflikt nazw, kompilator od razu o tym ostrzeże. Będziesz mógł wówczas zmienić nazwy konfliktowych składników, tworząc dwie oddzielne jednostki, lub wykorzystać słowo `Shadows` do rozwiązania problemu. (Więcej informacji znajdziesz w podrozdziale „Korzystanie z modyfikatora Shadows").

Dodawanie pól i właściwości

Pole jest daną składową klasy. Pola mogą być składowymi typu `ValueType`, jak np. `Integer` lub `Date`, lub też typu złożonego, czyli strukturą, wyliczeniem lub klasą. **Właściwości** są specjalnymi metodami, które ogólnie używane są do zapewnienia określonego dostępu do pól.

Generalnie, pola są prywatnymi elementami klasy. Jeśli zapewniony jest dostęp do pola, jest to zrealizowane za pomocą właściwości. Z tego powodu pola są zazwyczaj prywatne, a właściwości — publiczne. Jednakże czasami pola nie są w ogóle udostępnione poprzez właściwości. Właściwości również nie zawsze reprezentują pole. Czasami reprezentują one dane istniejące w bazie danych, rejestrze systemowym, pliku INI lub inną wartość, nie będącą bezpośrednio polem.

Motywacją tworzenia pól w postaci elementów prywatnych jest to, że nieograniczony dostęp do danych jest z natury ryzykowny. Rozważmy sposób przyrządzania sosu holenderskiego. Jeśli ugotujesz żółtka zbyt szybko, dostaniesz jajecznicę. Jednak jeśli będziesz powoli je gotował — kurek gazu w kuchence jest tu analogią do właściwości zmniejszającej lub zwiększającej ogień — otrzymasz prawidłową konsystencję sosu.

Kurek gazu jest metaforą na określenie metody właściwości. Pole reprezentuje ilość ciepła, a kurek — właściwość umożliwiającą jej zmianę. Osoba obsługująca kuchenkę nie jest uprawniona do zwiększenia ilości gazu poza określoną wartość. Wydruki 7.1 oraz 7.2 przedstawiają dwie częściowe klasy reprezentujące kuchenkę oraz ustawianie ilości wypływającego gazu.

Wydruk 7.1. *Klasa reprezentująca pole i właściwość dla ustawień temperatury kuchenki gazowej*

```
 1: Public Class Stove
 2:
 3:   Private FTemperature As Double
 4:
 5:   Public Property Temperature() As Double
 6:
 7:     Get
 8:       Return FTemperature
 9:     End Get
10:
11:     Set(ByVal Value As Double)
12:       Debug.Assert(Value < 400 And Value >= 0)
13:       If (FTemperature >= 400) Then Exit Property
14:       FTemperature = Value
15:       Debug.WriteLine(FTemperature)
16:     End Set
17:
18:   End Property
19:
20: End Class
```

Na wydruku 7.1 widzimy zdefiniowane pole `FTemperature` jako `Double`, dostępne poprzez właściwość `Temperature`. Metoda właściwości `Get` zwraca wartość pola, a `Set` przypisuje nową wartość do wartości pola. W wierszu 13. sprawdzany jest warunek, czy nie zostały ustawione niedopuszczalne wartości temperatury podczas tworzenia klasy. Przy jej wdrażaniu metoda `Debug.Assert` zostanie wyłączona przez kompilator. Jeśli `Stove` byłaby używana do kontroli rzeczywistej kuchenki, na pewno nie chciałbyś, aby temperatura mogła przekroczyć możliwości fizyczne kuchenki. W celu upewnienia się, że ustawienia temperatury są prawidłowe podczas normalnej pracy urządzenia, w wierszu 13. skorzystano z lustrzanego warunku `If` w celu utrzymania temperatury w dopuszczalnych granicach.

Powróćmy teraz do dyskusji poświęconej gramatyce języka. Zauważyłeś pewnie, że korzysta się z prefiksu `F` przy nazwach pól, a opuszcza się je w nazwach właściwości. Zwróć uwagę również na zmianę sposobu deklaracji właściwości. W Visual Basic .NET deklaracja właściwości jest pisana w pojedynczej strukturze bloku. Metody `Get` oraz `Set` są blokami zagnieżdżonymi pomiędzy słowami `Property` i `End Property`, które definiują granice właściwości. (Przypomnij sobie, że w VB6 metody `Get` i `Set` były definiowane jako dwa oddzielne i niezależne bloki).

Wydruk 7.2 definiuje nową klasę o nazwie `FuelSystem`, która wykorzystuje różne techniki do zapewnienia, że ustawienia przepustnicy mieszczą się w dopuszczalnych granicach.

Wydruk 7.2. *Wykorzystanie wyliczenia w połączeniu z metodami właściwości w celu ograniczenia wartości pola*

```
 1: Public Class FuelSystem
 2:
 3:   Public Enum ThrottleSetting
 4:     Idle
 5:     Cruise
 6:     Accelerate
 7:   End Enum
 8:
 9:   Private FThrottle As ThrottleSetting
10:
11:   Public Property Throtte() As ThrottleSetting
12:
13:     Get
14:       Return FThrottle
15:     End Get
16:
17:     Set(ByVal Value As ThrottleSetting)
18:       FThrottle = Value
19:       Debug.WriteLine(Value)
20:     End Set
21:
22:   End Property
23:
24: End Class
```

Wyliczenie `ThrottleSetting` definiuje trzy dopuszczalne stany przepustnicy: `Idle` (bieg jałowy), `Cruise` (prędkość podróżna) i `Accelerate` (przyspieszanie). Jeśli wykorzystasz dyrektywę `Option Strict` z tym wyliczeniem, klienci nie będą mogli ustawić niewłaściwej wartości `Throttle`. `Option Strict` zapewni, że wszystkie wartości przekazywane do metody `Set` właściwości `Property` będą z zakresu `ThrottleSetting`, czyli `Idle`, `Cruise` lub `Accelerate`.

Hermetyzacja i właściwości

Specyfikatory dostępu mogą być dodawane do każdego składnika w klasie. W największej liczbie przypadków, właściwości będą deklarowane jako składowe `Public` klasy lub struktury. Oczywiście możesz zdefiniować właściwość z dowolnym innym specyfikatorem.

Jedną ze strategii, gdzie stosuje się właściwości `Protected`, są klasy, które konsumenci mogą czynić bardziej ogólnymi. Przez zadeklarowanie klasy z właściwościami chronionymi umożliwiasz generalizatorom podjęcie decyzji, czy awansować te właściwości do dostępu publicznego.

Specyfikator dostępu jest umieszczany przed słowem kluczowym `Property`. Jeśli nie dołączysz żadnego specyfikatora, składowe będą miały dostęp publiczny. Przykłady umieszczania specyfikatorów są zawarte w wydrukach 7.2 oraz 7.3.

Podawaj specyfikatory dostępu za każdym razem, gdy definiujesz składową. W rezultacie otrzymasz kod czytelny i jasny.

Definiowanie właściwości indeksowanych

Właściwości indeksowane są po prostu metodami właściwości posiadającymi obowiązkowy parametr. Parametr ten jest semantycznie traktowany jak indeks, ale po umieszczeniu kodu w metodzie właściwości możesz robić z nim, co zechcesz. Poniżej zostaną omówione cechy właściwości i różnice pomiędzy właściwościami a właściwościami indeksowanymi.

Podstawowa, zwykła właściwość posiada metody `Get` oraz `Set`. Pierwsza z nich zachowuje się jak funkcja i jest niejawnie wywoływana, gdy właściwość zostaje użyta jako wartość prawostronna. Metoda `Set` działa jak dane wykorzystywane jako wartości lewostronne i ustawia wartość odpowiadającą tej właściwości. W najprostszym wydaniu wartość właściwości jest przechowywana w odpowiadającym jej polu. Zarówno właściwość, jak i pole są tego samego typu. Wydruk 7.2 demonstruje wykorzystanie prostej właściwości, posiadającej metody `Get`, `Set` oraz korzystającej z wartości pola. Zauważ, że definicja właściwości w wierszu 11. określa zwracany typ, lecz nie pobiera żadnych argumentów.

W przeciwieństwie do zwykłej właściwości, właściwość indeksowana posiada zdefiniowany w nawiasach argument. Nie reprezentuje on wartości pola; jest raczej indeksem do tej wartości. Konsekwencją obecności indeksu jest to, że odpowiadające pole musi być tablicą albo kolekcją, co zazwyczaj ma miejsce. Jednakże indeks może być używany do czegokolwiek, a parametr indeksowy nie musi być wartością liczbową.

Podstawową ideą stojącą za korzystaniem z właściwości indeksowanych jest to, że możesz bezpiecznie opakować tablice i inne rodzaje kolekcji danych w metody właściwości. Cel tego działania jest taki sam, jak w przypadku opakowania pojedynczej danej w metodę właściwości: chcesz ochronić dane przed nadużywaniem. Na wydruku 7.3 zostały zaprezentowane dwie właściwości indeksowane. Obie w rzeczywistości odnoszą się do tej samej tablicy, jednak odwołują się do niej na dwa różne sposoby.

Wydruk 7.3. *Właściwości indeksowane*

```
1: Public Class Indexed
2:
3:   Private FStrings() As String = {"Jeden", "Dwa", "Trzy"}
4:
5:   Default Public Property Strings(ByVal Index As Integer) As String
6:     Get
7:       Return FStrings(Index)
```

```
 8:     End Get
 9:     Set(ByVal Value As String)
10:       FStrings(Index) = Value
11:     End Set
12:   End Property
13:
14:   Private Sub Swap(ByVal OldIndex As Integer, _
15:     ByVal NewIndex As Integer)
16:
17:     Dim Temp As String = FStrings(NewIndex)
18:     FStrings(NewIndex) = FStrings(OldIndex)
19:     FStrings(OldIndex) = Temp
20:   End Sub
21:
22:   Public Property Names(ByVal Name As String) As Integer
23:     Get
24:       Return Array.IndexOf(FStrings, Name)
25:     End Get
26:
27:     Set(ByVal Value As Integer)
28:       Swap(Names(Name), Value)
29:     End Set
30:   End Property
31:
32: End Class
```

W klasie `Indexed` są zdefiniowane dwie właściwości indeksowane. Pierwsza z nich, o nazwie `Strings`, rozpoczyna się w wierszu 5. i pobiera argument typu `Integer`. Parametr ten został nazwany `Index` i literalnie funkcjonuje jako indeks do pola będącego tablicą łańcuchów, `FStrings`. Zwróć uwagę, w jaki sposób indeks ten jest wykorzystywany w metodzie `Get` w wierszach 6. – 8. i `Set` w wierszach 9. – 11. Wiersze 7. i 11. demonstrują intuicyjne użycie indeksu i tablicy. Gdy egzemplarze właściwości `Indexed` oraz `Strings` są użyte jako wartości prawostronne, wówczas wywoływana jest metoda `Get`. (Przykładem wykorzystania wartości prawostronnej jest instrukcja `MsgBox(Indexed.Strings(1))`). Metoda `Get` ma zastosowanie, gdy `Indexed.Property` jest używana jako wartość lewostronna, jak w instrukcji `Indexed.Strings(0) = "Jeden"`.

Porównaj właściwości `Strings` i `Names`. W obu z nich argument pełni rolę indeksu. Właściwości te zwracają wartości o określonych typach — `String` w przypadku `Strings` (wiersz 5.) i `Integer` w przypadku `Names` (wiersz 22.). Z tego względu `Strings` jest właściwością indeksowaną typu `String`, a `Names` — właściwością indeksowaną typu `Integer`. `Names` pobiera jako argument łańcuch znaków, `Name`, i zwraca pozycję tego łańcucha w postaci indeksu odpowiadającej tablicy.

W wierszu 24. użyta jest metoda `Shared` o nazwie `System.Array.IndexOf`, która pobiera tablicę i poszukiwany obiekt, a zwraca indeks tego obiektu w pobranej tablicy. W wierszu 28. widzimy wywołanie `Swap` z wykorzystaniem bieżącego indeksu istniejącego elementu (znalezionego za pomocą metody `Get`) oraz nowego indeksu reprezentowanego przez `Value`. Element tablicy jest przenoszony ze starej pozycji do nowej. Poniższy fragment pokazuje, w jaki sposób możesz znaleźć właściwość `Names` użytą w kodzie:

```
Dim MyIndexed As New Indexed
MyIndexed.Names("Jeden") = 1
```

Gdy kod z wiersza 2. powyższego fragmentu zostanie wykonany, wartość pola — tablicy `FStrings` — jest równa `{"Dwa", "Jeden", "Trzy"}`. Wartość na pozycji odpowiadającej nazwie „Jeden" jest przenoszona do indeksu 1 poprzez zamianę miejscami z wartością elementu o indeksie 1.

Korzyści ze stosowania właściwości indeksowanych

Korzyści ze stosowania właściwości indeksowanych są wielorakie. Oczywistą korzyścią jest to, że dane powinny być chronione przez metody bez wpływu na łatwość korzystania z nich. To, że dane są tablicami czy kolekcją, nie oznacza, że nie powinny być chronione przed niewłaściwym użyciem. Być może nie tak oczywistą zaletą jest również to, że bardziej złożone kolekcje danych mogą być łatwiejsze do użycia poprzez wskazanie, że właściwość indeksowana ma być właściwością domyślną. (Więcej na ten temat za chwilę).

Wydruk 7.3 nie pokazuje ochrony danych przed nadużyciem. W naszej aplikacji musielibyśmy zapewnić sprawdzanie, czy nie są używane niewłaściwe indeksy lub nazwy. Spójrzmy zatem, w których miejscach kodu możemy wprowadzić usprawnienia. Zacznijmy od metody `Indexed.Names`.

Co stałoby się, gdybyśmy zażądali `Indexed.Names("Cztery")`? Patrząc na istniejący kod — metoda `Array.IndexOf` zwróciłaby wartość `-1`. Odpowiada nam takie rozwiązanie, pozostawiamy więc `Names Get` bez zmian. Alternatywnie, moglibyśmy wygenerować wyjątek, jeśli nazwa nie zostałaby znaleziona, jednak już teraz przy podaniu niewłaściwego indeksu jest on podnoszony. W rezultacie ten fragment kodu jest w porządku.

Przyjrzyjmy się teraz metodzie `Set` właściwości `Names`. Nie ma w niej żadnego sprawdzania poprawności danych. Możemy sprawdzić, jak zachowuje się fragment kodu w przypadku podania błędnych danych — jeśli użyty zostanie zły indeks, powinien wystąpić wyjątek. Faktycznie, `Indexed.Names("Szesc") = 5` podnosi `System.IndexOutOfRangeException`. Wywołujący musi prawdopodobnie wiedzieć, że popełnił błąd, więc wystąpienie wyjątku jest rozsądne. Z drugiej strony, informacja pokazująca się w momencie wystąpienia wyjątku jest zbyt ogólna i możemy uznać, że nas nie zadowala (zobacz rysunek 7.2).

Rysunek 7.2. *Informacja pojawiająca się w chwili wystąpienia wyjątku przy niewłaściwym użyciu indeksu w obiekcie System.Array*

Dla celów praktycznych, jeśli domyślna informacja o wyjątku zapewnia wystarczającą informację, nie ma potrzeby pisania dodatkowego kodu do jego obsługi. Pamiętaj, że zawsze możesz obsłużyć wyjątek dodatkową procedurą.

Dla celów demonstracyjnych załóżmy, że domyślne zachowanie się wyjątku nie jest wystarczające. Zaimplementujemy rozszerzoną jego obsługę, aby zapewnić konsumentom klasy `Indexed` dodatkowe informacje.

Modyfikacja metody Set właściwości Names

Praktyczna modyfikacja tej metody ma na celu poinformowanie konsumenta, że użył niewłaściwego indeksu. Będzie ona polegać na dodaniu we właściwości `Names` nowych funkcji i modyfikacji metody `Set`. Na wydruku 7.4 przedstawiono zmiany dokonane w stosunku do kodu z wydruku 7.3.

Wydruk 7.4. *Dodanie kodu umożliwiającego obsługę błędu wynikającego z niewłaściwego użycia indeksu*

```
 1: Private Function ValidIndex(ByVal Index As Integer) As Boolean
 2:   Return (Index >= FStrings.GetLowerBound(0)) And _
 3:     (Index <= FStrings.GetUpperBound(0))
 4: End Function
 5:
 6: Private Overloads Sub Validate(ByVal Name As String)
 7:   Validate(Names(Name))
 8: End Sub
 9:
10: Private Overloads Sub Validate(ByVal Index As Integer)
11:   If (Not ValidIndex(Index)) Then
12:     Throw New ApplicationException(Index & " jest niepoprawnym indeksem")
13:   End If
14: End Sub
15:
16: Public Property Names(ByVal Name As String) As Integer
17:   Get
18:     Return Array.IndexOf(FStrings, Name)
19:   End Get
20:
21:   Set(ByVal Value As Integer)
22:     Validate(Name)
23:     Swap(Names(Name), Value)
24:   End Set
25: End Property
```

Zmiana w metodzie `Set` właściwości `Names` polega na dodaniu wywołania do funkcji `Validate`. Wiemy, że moglibyśmy sprawdzać poprawność indeksu bezpośrednio w metodzie `Set`, lecz we wcześniejszych rozdziałach omawialiśmy zalety i cel stosowania techniki refaktoringu „Wydzielanie metod". W kodzie powyżej utworzyliśmy osobną funkcję `Validate` głównie po to, aby `Set` była jak najkrótsza i najprostsza oraz aby mieć możliwość wielokrotnego wykorzystania kodu `Validate`. Zastosowaliśmy przy tym dwie przeciążone wersje `Validate`. Pierwsza z nich pobiera nazwę jako argument i wywołuje drugą wersję, pobierającą indeks. (Kolejną właściwością, gdzie moglibyśmy wprowadzić kod sprawdzający poprawność, jest `Strings`, która wykorzystuje indeks całkowity) Aby uczynić kod bardziej czytelnym, zaimplementowaliśmy sprawdzanie wartości indeksu w postaci metody `ValidIndex` (wiersze 1. – 4.). Jeśli indeks jest niepoprawny, generowany jest wyjątek w wierszu 12.

Zauważ, że w kodzie nie ma komentarzy. Zastosowanie krótkich, zwięzłych i jasnych metod czyni komentarze zbędnymi.

Tworzenie podklasy z klasy wyjątku

Przypuśćmy, że chcemy wykorzystać kod wyrzucający wyjątek w wierszu 12. na wydruku 7.4 w kilku innych miejscach programu i być może nawet w innych klasach. Załóżmy również, że klasa `Indexed` reprezentuje bardzo istotną i ważną część tworzonego systemu.

Więcej informacji na temat obsługi wyjątków znajdziesz w rozdziale 5., „Procedury, funkcje i struktury", a na temat dziedziczenia — w rozdziale 10., „Dziedziczenie i polimorfizm".

Moglibyśmy więc wprowadzić osobną klasę wyjątku, która hermetyzowałaby obsługę błędnych indeksów. Jest to możliwe poprzez utworzenie podklasy z istniejącej klasy obsługi wyjątku i rozszerzenie jej działania. Podejście to jest jak najbardziej prawidłowe i wielu programistów innych języków od lat wykorzystuje rozszerzone klasy wyjątków w swoich aplikacjach.

Założyliśmy, że nasza klasa `Indexed` jest istotna oraz że kod obsługi wyjątku wyrzucanego ze względu na niepoprawny indeks będzie wykorzystywany wielokrotnie. W pomocy Visual Studio .NET możemy przeczytać, że jeśli chcemy utworzyć własną obsługę wyjątków, powinniśmy skorzystać z klasy `System.ApplicationException`.

Warunkiem wystąpienia błędu w naszej aplikacji jest przekazanie nieprawidłowego indeksu do metody `Set` właściwości `Names` klasy `Indexed`. Wyjątek jest generowany w wierszu 12. wydruku 7.4, gdy sprawdzanie poprawności zakończy się niepowodzeniem. Takie zachowanie się kodu jest funkcjonalne, nie będziemy więc zmieniać organizacji w tym miejscu. Zmienimy jednak zaangażowane w to obiekty, co prezentuje wydruk 7.5.

Wydruk 7.5. ***Kompletny wydruk klasy Indexed, zawierający nową klasę wyjątku***

```
 1: Public Class IndexedException
 2:   Inherits ApplicationException
 3:
 4:   Public Sub New(ByVal Str As String)
 5:     MyBase.New(Str)
 6:   End Sub
 7:
 8:   Public Shared Sub ThrowException(ByVal Index As Integer)
 9:     Throw New IndexedException(Index & " jest niepoprawnym indeksem")
10:   End Sub
11:
12: End Class
13:
14: Public Class Indexed
15:
16:   Private FStrings() As String = {"Jeden", "Dwa", "Trzy"}
17:
18:   Default Public Property Strings(ByVal Index As Integer) As String
19:     Get
20:       Return FStrings(Index)
21:     End Get
22:
23:     Set(ByVal Value As String)
24:       FStrings(Index) = Value
25:     End Set
26:   End Property
```

```
27:
28:   Private Sub Swap(ByVal OldIndex As Integer, _
29:     ByVal NewIndex As Integer)
30:
31:     Dim Temp As String = FStrings(NewIndex)
32:     FStrings(NewIndex) = FStrings(OldIndex)
33:     FStrings(OldIndex) = Temp
34:   End Sub
35:
36:   Private Function ValidIndex(ByVal Index As Integer) As Boolean
37:     Return (Index >= FStrings.GetLowerBound(0)) And _
38:       (Index <= FStrings.GetUpperBound(0))
39:   End Function
40:
41:   Private Overloads Sub Validate(ByVal Name As String)
42:     Validate(Names(Name))
43:   End Sub
44:
45:   Private Overloads Sub Validate(ByVal Index As Integer)
46:     If (Not ValidIndex(Index)) Then
47:       'Throw New ApplicationException(Index & " jest niepoprawnym indeksem")
48:       IndexedException.ThrowException(Index)
49:     End If
50:   End Sub
51:
52:   Public Property Names(ByVal Name As String) As Integer
53:     Get
54:       Return Array.IndexOf(FStrings, Name)
55:     End Get
56:
57:     Set(ByVal Value As Integer)
58:       Validate(Name)
59:       Swap(Names(Name), Value)
60:     End Set
61:   End Property
62:
63: End Class
```

Nowa klasa wyjątku jest zdefiniowana w pierwszych dwunastu wierszach. W wierszu 2. następuje wskazanie, że `IndexedException` dziedziczy z `System.ApplicationException`. W żargonie programowania obiektowego mówi się, że jest to relacja `IsA`. Zunifikowany język modelowania (ang. *Unified Modeling Language — UML*) określa dziedziczenie generalizacją. Mówimy, że `IndexedException` jest `ApplicationException`.

`IndexedException` wprowadza przeciążonego konstruktora, który pobiera argument w postaci łańcucha. Argument ten — wiersz 4. — stanie się częścią komunikatu naszego wyjątku. W wierszu 5. używana jest zmienna `MyBase`, która odnosi się do klasy bazowej. `MyBase` jest dostępna podobnie jak zmienna `Me`, która jest referencją do samej siebie. W wierszu 8. definiowana jest metoda `Shared ThrowException`. Metoda `Shared`, która tworzy egzemplarz obiektu, jest nazywana metodą fabryczną. Metoda taka umożliwia zlokalizowanie konstrukcji i zainicjowanie obiektu, zmniejszając potrzebę duplikowania kodu tworzącego obiekt. I w końcu, procedura `Validate` zastępuje instrukcję z wiersza 12. wydruku 7.4 na wywołanie do nowej metody fabrycznej, która tworzy i wyrzuca wyjątek. (Oryginalna instrukcja i jej zastąpienie są odpowiednio w wierszach 47. i 48. wydruku 7.3.).

W chwili obecnej znasz już sposoby definiowania właściwości indeksowanych, dodawania kodu sprawdzającego do metod właściwości oraz wiesz, jak wprowadzać nowe klasy obsługi wyjątków. Jeśli dziedziczenie jest dla Ciebie pojęciem niejasnym, więcej informacji na jego temat znajdziesz w rozdziale 10., „Dziedziczenie i polimorfizm".

Korzystanie z właściwości domyślnych

Visual Basic .NET umożliwia korzystanie z właściwości domyślnych `Default`, jednak w przeciwieństwie do VB6, w VB .NET jedynie właściwości indeksowane mogą być domyślne. Powód tego jest przedstawiony poniżej.

W VB6 korzystaliśmy z metody `Set` podczas przypisywania referencji obiektu do zmiennej obiektowej. Ponadto, jeśli element przypisywany był obiektem i nie było obecnej instrukcji `Set`, VB6 wnioskował, że programista zamierza wykorzystać właściwość `Default`. Z tego powodu właściwości domyślne w VB6 nie musiały być właściwościami indeksowanymi. Istnienie słowa kluczowego `Set` stanowiło wystarczającą wskazówkę dla kompilatora.

Visual Basic .NET nie wykorzystuje `Set` do przypisywania obiektów. Z tego względu obecność obiektu, a nieobecność `Set` nie są wystarczające dla kompilatora do określenia Twoich zamiarów. W Visual Basic .NET podpowiedzią dla kompilatora, że chcemy skorzystać z właściwości `Default`, jest obecność nawiasów przy nazwie obiektu. Wypływa stąd wniosek, że w VB .NET wszystkie właściwości `Default` muszą być właściwościami indeksowanymi oraz możliwe jest posiadanie tylko jednej właściwości domyślnej.

Również w wersji obiektowej Pascala — Object Pascal — jedynie właściwości indeksowane mogą być domyślne. Obecność `object[]` w tym języku jest wskazówką dla kompilatora, że programiście chodzi raczej o właściwość indeksowaną niż o przypisanie obiektu.

Jest możliwe — i bardzo prawdopodobne — że zmiana wprowadzona we właściwościach domyślnych była dokonana częściowo pod wpływem Andersa Hejlsberga. Hejlsberg, w chwili obecnej Wyróżniony Inżynier (Distinguished Engineer) w Microsoft, był kierownikiem sekcji architektów w firmie Borland i odegrał zasadniczą rolę w procesie implementacji Delphi, stworzonego w Object Pascal. To nie jedyny wpływ Object Pascala na elementy Visual Basic .NET. Nie można jednak powiedzieć, że VB .NET jest podobny do Pascala; należałoby raczej stwierdzić, że Visual Basic .NET jest językiem rozwijającym się i Microsoft podjął bardzo mądrą decyzję, udoskonalając go z uwzględnieniem zalet innych języków.

Trzeba się przyzwyczaić do tego, że Visual Basic .NET jest produktem silnie zmienionym i mocno udoskonalonym, jednak cały czas jest to Visual Basic.

Aby wskazać, że właściwość ma być właściwością domyślną, umieść słowo kluczowe `Default` w wierszu, w którym definiujesz właściwość, bezpośrednio przed specyfikatorem dostępu. Fragment wydruku 7.5 pokazuje sposób umieszczania słowa `Default`, czyniąc `Strings` właściwością domyślną:

```
Default Public Property Strings(ByVal Index As Integer) As String
  Get
    Return FStrings(Index)
  End Get
```

```
    Set(ByVal Value As String)
      FStrings(Index) = Value
    End Set
  End Property
```

Zakładając, że zadeklarowaliśmy egzemplarz `Indexed`, używając instrukcji `Dim IndexedObject As New Indexed`, dostęp do właściwości `Strings` jest możliwy poprzez wywołanie:

```
MsgBox(IndexedObject.Strings(1))
```

lub, ze względu na to, że `Strings` jest właściwością domyślną:

```
MsgBox(IndexedObject(1))
```

Podsumowując, właściwości domyślne muszą być indeksowane, możesz mieć tylko jedną właściwość domyślną w klasie i możesz wywoływać metody `Get` i `Set` właściwości domyślnej, korzystając z zapisu pełnego lub skróconego.

Korzystanie z modyfikatorów właściwości

Oprócz specyfikatorów dostępu, istnieją również specjalne modyfikatory, które występują tylko przy właściwościach. Są to modyfikatory `ReadOnly` oraz `WriteOnly`.

Właściwość tylko do odczytu (*read-only*) może być wykorzystywana wyłącznie jako argument prawostronny. To znaczy, użytkownicy takiej metody mogą jedynie odczytywać właściwość, nie mogą jej modyfikować. Taka właściwość posiada więc jedynie metodę `Get`. Konsumenci klasy mogą traktować właściwości `ReadOnly` jak stałe lub niezmienne dane, lecz należy pamiętać, że może istnieć mechanizm wewnętrzny w stosunku do obiektu, który może zmienić odpowiadającą właściwości wartość, mimo że konsument nie ma takiej możliwości.

Właściwość tylko do zapisu (*write-only*) może być modyfikowana przez konsumenta, ale nie może zostać przez niego oglądana. W nich zaimplementowana jest wyłącznie metoda `Set`.

Zewnętrzny blok właściwości jest w obu przypadkach identyczny, z wyjątkiem obecności modyfikatora. Wewnętrznie właściwości `ReadOnly` posiadają jedynie metodę `Get`, a `WriteOnly` — metodę `Set`.

Implementowanie właściwości ReadOnly

Przykładem właściwości tylko do odczytu może być zwracanie znacznika czasowego podczas tworzenia obiektu. Ponieważ obiekt może być utworzony tylko raz — mimo że możemy mieć wiele egzemplarzy klasy, to dany egzemplarz jest tworzony tylko jeden raz — nie ma sensu, aby umożliwiać konsumentom modyfikację daty utworzenia obiektu.

Jeśli musisz śledzić, jak długo dany obiekt istnieje, i chcesz mieć pewność, że konsumenci nie zmienią znacznika czasu tego obiektu, możesz zdefiniować właściwość tylko do odczytu `CreateTime` i zainicjować ją w konstruktorze:

```
Public Sub New()
  MyBase.New()
  FCreateTime = Now
End Sub

Privae FCreateTime As Date

Public ReadOnly Property CreateTime() As Date
  Get
    Return FCreateTime
  End Get
End Property
```

Zwróć uwagę na położenie modyfikatora `ReadOnly`. Publiczna właściwość `CreateTime` jest zdefiniowana jako tylko do odczytu i z tego powodu posiada wyłącznie blok `Get`. Konstruktor — `Sub New` — inicjuje odpowiadające właściwości pole, `FCreateTime`, w momencie wywołania konstruktora.

Innym przykładem wykorzystania właściwości tylko do odczytu jest pole obliczone. Jeśli właściwość nie posiada odpowiadającej wartości, a jej wartość jest gdzieś obliczana, nie ma sensu implementacja metody `Set`. Weźmy pod uwagę symetryczną właściwość `ElapsedTime`. Jeśli chcielibyśmy określić czas, jaki upłynął od uruchomienia aplikacji, moglibyśmy zwrócić ten czas jako wynik odjęcia od bieżącego czasu czas `CreateTime`:

```
Public ReadOnly Property ElapsedTime() As TimeSpan
  Get
    Return Date.op_Subtraction(Now, FCreateTime)
  End Get
End Property
```

Właściwość `ElapsedTime` jest tylko do odczytu, gdyż odpowiadająca jej wartość jest obliczana dynamicznie za każdym razem, gdy właściwość jest wywoływana. Przy okazji widzimy wykorzystanie klasy `TimeSpan`, której implementacja w powyższym kodzie umożliwia bardzo dokładny pomiar czasu. Inną cechą kodu jest wywołanie współdzielonej metody `Date.op_Subtraction`. Visual Basic .NET nie zezwala jeszcze na przeciążanie operatorów, więc zaimplementowane zostały specjalne metody z prefiksem `op_` w miejscach, gdzie zachowanie się ich jest analogiczne do operatora przeciążonego, który mógłby wystąpić np. w przypadku pisania kodu w C#.

`ElapsedTime` zwraca różnicę pomiędzy `Now` i czasem utworzenia obiektu. Możesz przetestować obie właściwości za pomocą kilku wierszy kodu:

```
Dim O As New PropertyModifiers()
System.Threading.Thread.Sleep(1000)
MsgBox(O.ElapsedTime.ToString)
```

W wierszu 1. tworzony jest egzemplarz klasy `PropertyModifiers`, zawierający predefiniowane właściwości, o których dyskutowaliśmy. Konstruktor inicjuje pole `FCreateTime`. Druga instrukcja korzysta z metody `Sleep`, która usypia bieżący wątek. W naszym przypadku jest to 1000 milisekund, czyli 1 sekunda. W wierszu 3. wywoływana jest metoda właściwości `ElapsedTime`, która zwraca obiekt `TimeSpan`. Na końcu wywoływana jest metoda `ToString` przy użyciu niejawnej referencji do obiektu `TimeSpan`.

Te trzy proste wiersze kodu powodują w rezultacie wyświetlenie okna komunikatu pokazanego na rysunku 7.3. Zgodnie z jego zawartością wygląda na to, że Visual Basic .NET potrzebował około 1,4 milisekundy na wywołanie funkcji `Now` od momentu obudzenia wątku.

Rysunek 7.3. *Okno dialogowe prezentujące rozdzielczość klasy TimeSpan w Visual Basic .NET*

Implementowanie właściwości WriteOnly

Właściwości tylko do zapisu są wykorzystywane rzadziej niż wcześniej omawiane `ReadOnly`, jednak istnieje kilka dobrych powodów, dla których ich sporadyczne wykorzystanie jest pomocne. Jedną z takich okazji jest właściwość związana z ustalaniem hasła.

Przypuśćmy, że mamy klasę sprawdzania tożsamości użytkownika, która akceptuje nazwę użytkownika i zamaskowane hasło. O ile odpowiedź na zapytanie obiektu klienta „Kto jest użytkownikiem" byłaby wskazana, o tyle podobna odpowiedź na pytanie „Jakie jest jego hasło" byłaby już bardziej ryzykowna. W takim przypadku chcesz zapewne, aby użytkownik mógł wpisać hasło, ale jednocześnie żeby żaden z obiektów klienta nie mógł tej informacji uzyskać. Właściwość ta mogłaby więc być zaimplementowana w sposób następujący:

```
Private FPassword As String

Public WriteOnly Property Password() As String
  Set(ByVal Value As String)
    FPassword = Value
  End Set
End Property
```

Zauważ, że edytor kodu utworzył tylko blok `Set`, a wartość przypisywana do właściwości jest przechowywana w odpowiadającym polu.

Z przyczyn praktycznych możesz zmodyfikować ten kod, aby był jeszcze bardziej bezpieczny. Na przykład możesz zaimplementować właściwość `WriteOnly`, która będzie od razu sprawdzać hasło w metodzie `Set` tej właściwości, a potem wyczyści zmienną zawierającą hasło. A informacją przechowywaną będzie jedynie to, czy podane hasło było poprawne. Taka zmiana zapobiegnie możliwości podglądnięcia obszaru pamięci odpowiadającej zawartości klasy przez programu szpiegujące.

Inne modyfikatory właściwości, jak `Shadows`, mogą być używane zarówno z właściwościami, jak i z metodami. Aby uniknąć nadmiarowości informacji w Twoim kodzie, przyjmij, że modyfikatory mogą być stosowane w dowolnych składnikach klas, chyba że tekst wskazuje na coś innego. `WriteOnly` i `ReadOnly` można stosować wyłącznie we właściwościach.

Definiowanie właściwości współdzielonych

Obszerna i wyczerpująca dyskusja na temat składowych `Shared` została zawarta w rozdziale 11., „Składowe współdzielone". Tam również znajdziesz przykłady ich stosowania W tej chwili wystarczy powiedzieć, że właściwości mogą być współdzielone (`Shared`) lub występować jako składowe egzemplarzy. Właściwości `Shared` oznaczają, że możesz je wywołać w klasie — właściwość taka jest definiowana z użyciem słowa `Shared`. Składowe `Shared` mogą być także wywołane przez egzemplarze, za to składowe egzemplarzy mogą być wywołane jedynie za pomocą egzemplarza klasy. Oto przykład:

```
Public Shared ReadOnly Property Name() As String
  Get
    Return "PropertyModifiers"
  End Get
End Property
```

W przykładzie zdefiniowana jest właściwość `Public`, `Shared` i `ReadOnly` o nazwie `Name`. Ponieważ nazwa klasy nigdy się nie zmienia, chyba że zmiany dokona programista w kodzie programu, powyższa metoda może mieć charakter tylko do odczytu. Zrobienie metody publiczną oznacza, że konsumenci mogą z tej metody korzystać, a użycie modyfikatora `Shared` zwalnia nas z potrzeby posiadania obiektu (egzemplarza klasy) w celu zapytania klasy „Jak się nazywasz?" Jeśli klasa jest nazwana jako `PropertyModifiers`, możemy wywołać jej metodę `Get` pisząc `PropertyModifiers.Name`.

Właściwość `Name` w powyższym fragmencie pokazuje bardzo istotną różnicę między Visual Basic .NET a VB6. Właściwość ta jest bardzo sprecyzowana w swoich zamierzeniach — Visual Basic .NET umożliwia nam określenie swoich zamiarów co do właściwości i wprowadzenie ich w życie za pomocą bardzo precyzyjnej gramatyki języka. W VB6 mogliśmy zdefiniować właściwość tylko do odczytu poprzez instrukcję `Let Property`, lecz jej status tylko-do-odczytu był ukryty. Poza tym, nie mogliśmy definiować składowych `Shared`.

Oczywiście można dyskutować, czy tak niewielkie zmiany semantyczne w języku rzeczywiście są istotne. Odpowiedź brzmi: „Są jak najbardziej istotne". Programowanie jest niemal tak samo precyzyjne jak matematyka, a języki, w których programujemy, powinny być ekspresyjne w sposób umożliwiający uniknięcie tworzenia niejednoznacznego, głupiego i nadmiarowego kodu. Niuanse i subtelność w wyrażaniu się powinny być pozostawione kochankom i mężom stanu.

Dodawanie atrybutów właściwości

Atrybuty nie są całkowicie nowym zagadnieniem w Visual Basic .NET. Mogłeś się z nimi spotkać już w VB6. Na przykład otwórz w edytorze tekstu moduł klasy VB6 — plik .CLS — i zmień wartość atrybutu `Attribute VB_PredeclaredID = False` na `Attribute VB_PredeclaredID = True`, a w rezultacie otrzymasz klasę utworzoną automatycznie. Jeśli klasa ma nazwę `Class1`, a jej metoda `Foo`, to po powyższej modyfikacji napisanie `Class1.Foo` powinno uruchomić kod klasy. Dzieje się tak dlatego, gdyż atrybut `VB_PredeclaredID` jest mechanizmem, który sprawia, że formularze tworzą się automatycznie.

No dobrze, dlaczego więc ten temat jest poruszany w książce o Visual Basic .NET? Atrybuty są istotne, gdyż stanowią pełnoprawny aspekt tworzenia aplikacji w VB .NET. Zostały również znacząco zmodyfikowane i ulepszone w stosunku do atrybutów znanych z VB6. Wykorzystaj właściwość `Name` z poprzedniego podrozdziału do sprawdzenia działania jednego z atrybutów:

```
Public Shared ReadOnly Property Name() As String
    <System.Diagnostics.DebuggerHidden()> Get
      Return "PropertyModifiers"
    End Get
  End Property
```

Zwróć uwagę na wykorzystanie atrybutu w metodzie `Get`. (Atrybut `DebuggerHidden`, należący do klasy atrybutów z przestrzeni nazw `System.Diagnostics`, był omawiany w rozdziale 1., „Ujednolicone środowisko pracy Visual Studio"). Zapobiega on wchodzeniu przez debuger do wnętrza tej metody.

Atrybuty mogą mieć zastosowanie w bardzo wielu miejscach w Visual Basic .NET. Na przykład korzystasz ze znacznika atrybutu przy konwersji metody do metody `Web`. Być może atrybuty zakończą swój żywot tak, jak szybkie samochody kończą go w rękach nastoletnich kierowców, jednak są one całkowicie odmienione w Visual Basic .NET i ich omówienie wymaga osobnego rozdziału. W tej chwili musisz jedynie wiedzieć, że jeśli znajdziesz znacznik `<name>` w kodzie VB .NET, to jest to atrybut. W rozdziale 12., „Definiowanie atrybutów", znajdziesz szczegółowe omówienie atrybutów.

Dodawanie metod do klas

Metody są to funkcje i procedury, które należą do klasy lub struktury. Ponieważ struktury zostały szczegółowo omówione w rozdziale 5., „Procedury, funkcje i struktury", wszystkie nasze kolejne przykłady będą się opierać na klasach.

Cokolwiek jest możliwe do zrobienia z procedurą w module, to samo możesz uczynić z procedurą w klasie. Jedyną zasadą w przypadku metod jest **kierowanie się z wyczuciem własnymi upodobaniami** w trakcie pisania kodu. Osobiście lubię muzykę Beethovena i kapelę rockową Creed, więc prawdopodobnie moje upodobania różnią się od Twoich. Czy kwestia indywidualnych upodobań w przypadku programowania jest rzeczą istotną? I tak, i nie.

Jeśli jesteś programistą, możesz robić i tworzyć, co tylko Ci przyjdzie do głowy. Jeśli w dodatku nie dostajesz pieniędzy za kod, który piszesz, możesz puścić wodze fantazji i czynić prawdziwe cuda w swoich aplikacjach. Jednakże istnieje kilkanaście dobrych pozycji książkowych omawiających zasady dotyczące dobrego stylu w programowaniu. Jedną z ostatnio wydanych książek na ten temat jest pozycja Martina Fowlera traktująca o refaktoringu. Świetne książki o programowaniu obiektowym napisali także Grady Booch, James Coplien i Bjarne Stroustrup. Do niektórych z tych pozycji znajdziesz odnośniki w literaturze do tej książki.

Większość z przykładów prezentowanych w tej książce jest napisana z zastosowaniem w pewnym stopniu zasad refaktoringu. Ważniejszą rzeczą jest jednak to, że programuję

w ten sposób od ponad 12 lat. Swoje aplikacje piszę takim stylem z bardzo prostego powodu: po ponad 10 latach programowania w sześciu językach i napisania milionów wierszy kodu mój osobisty styl pozwala na tworzenie kodu o bardzo niewielkich rozmiarach, który może być wielokrotnie wykorzystywany, a przy tym jest łatwy do utrzymania i jest szybki. W tym miejscu chciałbym zaproponować kilka wskazówek, z których korzystam podczas implementowania metod.

- Staraj się tworzyć niewielki interfejs publiczny. Parafrazując Boocha (1995), implementuj garść metod i właściwości publicznych.
- Staraj się trafnie nazywać metody, unikając niestandardowych skrótów. W nazwie metody stosuj czasownik, rzeczownik pozostawiając opisowi właściwości.
- Niech Twoje metody wykonują działanie ściśle odpowiadające ich nazwie; przyjmij zasadę: jedna metoda, jedno działanie. Nie twórz metod, które wykonują na raz zbyt wiele czynności.
- Ograniczaj metody do około pięciu wierszy kodu, a na pewno nigdy ponad to, co możesz za jednym razem zobaczyć na ekranie.
- Korzystaj z techniki refaktoringu „Wydzielanie metod", dając nowym metodom trafne nazwy zamiast pisania długich komentarzy.
- Rozważ zaimplementowanie wszystkich niepublicznych metod jako chronionych i wirtualnych — nigdy nie wiesz, kto będzie chciał do nich w przyszłości zaglądać.

Zasady te nie powstały w ciągu jednego dnia. Są one efektem naśladowania mistrzów programowania przez dłuższy czas. Stosowanie niektórych z nich — jak na przykład tworzenia krótkich, pojedynczych metod — zawsze wydawało się logiczne. Na koniec opowiem krótką historyjkę, ilustrującą sens stosowania się do wskazówek.

Swój pierwszy samochód kupiłem za 325 dolarów w wieku 15 lat. Było to w dniu, w którym uzyskałem na to pozwolenie mojego nauczyciela. Samochód ten można było nazwać „toczącym się wrakiem". Nic w tym samochodzie nie było pięknego poza faktem, że potrafił jechać do przodu. Każde z kół miało inny rozmiar, dziura w chłodnicy była wielkości piłki do footballu, skrzynia biegów nie do końca chciała działać poprawnie, a regulacja położenia siedzenia kierowcy nie miała sprawnej blokady. Gdy samochód się zatrzymywał, siedzenie jechało do przodu; gdy ruszałem, siedzenie ślizgało się do tyłu.

Krótko po kupieniu auta podjąłem decyzję o rozpoczęciu prac renowacyjnych. Remont rozpocząłem od naprawy paska napędzającego wentylator chłodnicy. Od ojca pożyczyłem narzędzia i zacząłem naprawę od usunięcia wentylatora, a następnie pompy wodnej. Prawdopodobnie wyjąłbym też blok silnika, gdybym tylko wiedział, jak to zrobić — wszystko po to, aby wymienić pasek. Po kilku godzinach walki poddałem się. Śrubki, które znalazłem, wkręciłem z powrotem do silnika i zaprowadziłem samochód do zakładu. Mechanik poluzował alternator, luzując tym samym naciąg starego paska, zdjął ten pasek, założył nowy, po czym naciągnął go odpowiednio dokręcając śruby alternatora. Za 5 minut pracy skasował 35 dolców. Całkiem nieźle, jak na 5 minut. Zarobił 35 dolarów, jednak nie za to, że jego praca była ciężka, ale za to, że wiedział, jak ją zrobić. (No dobra, część tej historyjki zapożyczyłem ze starej bajki).

Morał z tej historii jest taki, że dokładna wiedza na temat, jak i dlaczego należy coś zrobić, jest jedynym usprawiedliwieniem na odchodzenie od ogólnie przyjętych zasad. Kod jest rzeczą tak osobistą, jak każda inna działalność twórcza. Przestrzeganie w przypadkach ogólnych dobrych zasad zwiększy Twoją produktywność; nieprzestrzeganie ich — udoskonali Twój talent.

Implementowanie konstruktorów i destruktorów

Istnieją dwie specjalne metody, które będziesz musiał implementować. Określa się je mianem konstruktora i destruktora. Konstruktor jest wywoływany do inicjowania klasy, destruktor — do deinicjowania lub finalizacji klasy. W Visual Basic .NET konstruktor implementowany jest jako `Sub New`, a destruktor występuje jako chroniona metoda `Sub Finalize`.

Każda klasa otrzymuje przynajmniej jednego konstruktora, odziedziczonego z klasy bazowej `Object`. `Object` definiuje procedurę bez parametrów `Sub New`, która jest wywoływana, gdy piszesz kod jak poniżej:

```
Dim zmiennaobiektowa As New nazwaklasy
```

zmiennaobiektowa jest dowolną poprawną nazwą zmiennej, a *nazwaklasy* reprezentuje jakikolwiek poprawny typ referencyjny. Z powyższej instrukcji można wywnioskować, że `New` wygląda jak operator, jednak po prześledzeniu kilku kolejnych przykładów zobaczysz, że taka instrukcja prowadzi bezpośrednio do metody `Sub New()`. Metoda New — czyli konstruktor — jest wywoływana, gdy tworzysz nowe egzemplarze klas.

Gdy obiekt jest usuwany z pamięci, systemowy odzyskiwacz pamięci (*garbage collector*) wywołuje metodę `Sub Finalize`, czyli destruktora. Możesz zaimplementować destruktora w celu deinicjalizacji obiektów poprzez dodanie metody `Finalize` do swoich klas:

```
Protected Overrides Sub Finalize()
End Sub
```

Mechanizm odzyskiwania pamięci często występuje jako skrót GC. GC jest również przestrzenią nazw zawierającą systemowy odzyskiwacz pamięci.

Tradycyjnie, celem destruktora jest zwolnienie pamięci przypisanej obiektom zawartym w klasie. Ponieważ Visual Basic .NET zatrudnia GC, nie możesz być pewien, kiedy dokładnie GC wywoła Twojego destruktora. Ciągle możesz korzystać z destruktora, aby usunąć obiekty ze swojej klasy, jeśli jednak posiadasz zasoby wrażliwe na czas, które muszą być zwolnione natychmiast, dodaj publiczną metodę `Dispose`. W kolejnych przykładach będziemy korzystali z `Public Sub Dispose()` w celu wykonania porządków typu zamykanie plików czy zestawów rekordów.

Definiowanie konstruktorów

Agregacja jest pojęciem określającym dodawanie składowych do klas. Gdy dana klasa posiada składowe, które również są klasami, przy czym bierze ona jednocześnie odpowiedzialność za tworzenie i niszczenie egzemplarzy tych składowych, to o takim powiązaniu mówimy, że jest to **agregacja**. Natomiast gdy klasa posiada składowe będące klasami, ale ich tworzeniem i usuwaniem zajmuje się jakaś jednostka z zewnątrz, to mówimy wówczas o **związku**. Gdy definiujesz agregacje w swojej klasie, musisz utworzyć dla niej konstruktora.

Powód tworzenia konstruktora może być dowolny, jednak jeśli będziesz chciał tworzyć egzemplarze składowych agregacyjnych, to konstruktor jest niezbędny. Domyślny

konstruktor jest reprezentowany przez `Sub New` i nie posiada parametrów. W celu jego zaprezentowania zdefiniowałem klasę `LoaderClass`, która wykorzystuje klasę `System.IO.TextReader` do załadowania pliku tekstowego do `ArrayList`.

```
Public Sub New()
  MyBase.New()
  FArrayList = New ArrayList()
End Sub
```

Kod będzie poprawny również wówczas, gdy usuniemy instrukcję `MyBase.New()`. Semantycznie, wszystkie klasy pochodne muszą wywoływać konstruktora macierzystego, jednak wygląda na to, że Visual Basic .NET w większości przypadków robi to za Ciebie. Zamiast zgadywania, kiedy i dlaczego VB .NET wywołuje konstruktora klasy bazowej, zawsze umieszczaj `MyBase.New()` jako pierwszą instrukcję w swoim konstruktorze. Jeśli zechcesz wywołać konstruktora parametrycznego w klasie potomnej, zamień po prostu wywołanie zwykłego konstruktora na wywołanie konstruktora parametrycznego.

Nie posiadające parametrów `Sub New()` wywołuje konstruktora klasy bazowej za pomocą instrukcji `MyBase.New()`. `MyBase` jest słowem zarezerwowanym, które umożliwia odwoływanie się do składowych w **klasie bazowej** danej klasy, zwanej także **klasą macierzystą**. Pamięcią wewnętrzną dla `LoaderClass` jest `ArrayList`. Ponieważ `LoaderClass` — którego konstruktor jest pokazany — jest w posiadaniu `ArrayList`, z tego względu konstruktor tworzy egzemplarz `ArrayList`, zanim cokolwiek innego zostanie wykonane.

Definiowanie przeciążonych konstruktorów parametrycznych

Jeśli w celu poprawnej inicjalizacji musisz przekazać do swojej klasy dane z zewnątrz, możesz zdefiniować konstruktora parametrycznego.

`LoaderClass` służy do załadowana pliku tekstowego do `ArrayList`. Wydaje się oczywiste, że należałoby zainicjować obiekty `LoaderClass` nazwą ładowanego pliku. Mając dwa konstruktory, możemy tworzyć egzemplarze bez znajomości nazwy pliku oraz gdy tę nazwę znamy:

```
Public Sub New(ByVal AFileName As String)
  Me.New()
  FileName = AFileName
End Sub
```

Aby uniknąć powielania kodu, wydelegujemy część odpowiedzialną za konstrukcję klasy do bezparametrycznego konstruktora z `Me.New()` i przechowamy parametr w pamięci podręcznej.

Posiadanie w tej samej klasie dwóch metod `Sub New` oznacza, że konstruktor został przeciążony. Konstruktory nie zezwalają na użycie słowa kluczowego `Overloads`. Jest to specjalna zasada wprowadzona dla konstruktorów przez inżynierów Visual Basic .NET. (Więcej na temat metod przeciążonych znajdziesz w podrozdziale „Korzystanie z modyfikatorów").

Implementowanie destruktorów

Jeśli przypisujesz pamięć w momencie wywołania konstruktora, musisz również zwolnić ją, implementując do tego celu destruktora. `Sub New` jest używany podobnie jak `Class_Initialize` w VB6, a `Sub Finalize` w sposób analogiczny do `Class_Terminate`. Obiekty utworzone przez konstruktora muszą być usunięte za pomocą destruktora.

Generalnie, jeśli w Twojej klasie nie ma zdefiniowanego konstruktora, prawdopodobnie również nie potrzebujesz destruktora. Oto podstawowa forma, w jakiej występuje destruktor `Finalize`, zaimplementowana w przykładowej klasie `LoaderClass`:

```
Protected Overrides Sub Finalize()
  Dispose()
End Sub
```

Implementując destruktora na podstawie metody `Dispose`, możesz bezpośrednio zwalniać obiekty za pomocą tej metody, jeszcze zanim uczyni to GC. Istnieją określone zasady i reguły dotyczące zwalniania zasobów — więcej informacji na ten temat znajdziesz w następnym podrozdziale.

Destruktor `Finalize` jest chroniony (`Protected`); z tego względu, nie możesz go bezpośrednio wywołać — jest on wywoływany przez GC. Oto podstawowe zasady obowiązujące przy implementacji destruktora:

- Metody `Finalize` powodują obciążenie systemu. Nie definiuj przeciążonej metody `Finalize`, jeśli nie jest to konieczne.
- Nie definiuj metody `Finalize` jako publicznej.
- Zwalniaj posiadane obiekty za pomocą metody `Dispose`; korzystaj z `Finalize` wywołując `Dispose`.
- Nie likwiduj referencji w swoim destruktorze; jeśli Twój kod nie utworzył obiektu, jest to referencja, a nie agregacja.
- Nie twórz obiektów ani nie korzystaj z innych obiektów w metodzie `Finalize`; destruktory służą wyłącznie do czyszczenia pamięci i zwalniania zasobów.
- W metodzie `Finalize` jako pierwszą instrukcję zawsze stosuj `MyBase.Finalize`.

Ze względu na to, że nie wiemy, kiedy GC zwalnia obiekty, można zaimplementować metodę `Public Dispose` i wywoływać ją jawnie w bloku ochrony zasobów `Finally`; dzięki temu uzyskamy konkretną finalizację obiektów takich jak np. strumienie danych czy wątki.

Implementowanie metody Dispose

Destruktory są chronione. Konsumenci nie mogą i nie powinni mieć możliwości bezpośredniego wywoływania metody `Finalize`. Możesz jawnie uruchomić systemowy odzyskiwacz pamięci za pomocą instrukcji `System.GC.Collect`, lecz nie jest to zalecana praktyka i powoduje znaczne obciążenie systemu.

Jeśli chcesz wyczyścić znaczące obiekty, zastosuj metodę `Public Dispose` i ją wywołaj. Oto kilka pożytecznych wskazówek dotyczących implementacji metody `Dispose`:

- Dodawaj metodę `Dispose`, wykorzystując interfejs `IDisposable`. (`IDisposable` posiada jedną metodę, `Dispose`, a w systemie pomocy Visual Studio .NET znajdziesz przykłady klas z zaimplementowanym tym interfejsem).
- Jeśli posiadasz istotne zasoby, takie jak uchwyty Windows, zestawy rekordów lub strumienie plików, które muszą być zwolnione do systemu zaraz po zakończeniu korzystania z nich, wówczas utwórz publiczną metodę `Dispose`.
- Zaprojektuj mechanizm powstrzymujący usuwanie, jeśli użytkownik bezpośrednio wywoła metodę `Dispose`.
- Jeśli klasa bazowa implementuje `IDisposable`, wywołaj metodę `Dispose` z klasy bazowej.
- Zaimplementuj metodę `Finalize`, która wywoła `Dispose`. W ten sposób upewnisz się, że `Dispose` będzie zawsze wywoływana.
- Zwalniaj posiadane obiekty w metodzie `Dispose`.
- Rozważ możliwość wygenerowania wyjątku `ObjectDisposedException`, w przypadku gdy konsument próbuje skorzystać z obiektu po wcześniejszym wywołaniu metody `Dispose`.
- W swojej metodzie `Dispose` wywołaj `GC.SuppressFinalize(Me)` aby powiedzieć GC, że nie musi uruchamiać metody `Finalize`.
- Zezwól na wywoływanie obiektu `Dispose` więcej niż jeden raz. Drugie i kolejne wywołania nie powinny wykonywać żadnych operacji.

Przy wykorzystaniu tych wskazówek poniżej została utworzona metoda `Dispose` dla `LoaderClass`:

```
Public Sub Dispose() Implements IDisposable.Dispose
  Static FDisposed As Boolean = False
  If (FDisposed) Then Exit Sub
  FDisposed = True
  Close()
  FArrayList = Nothing
  GC.SuppressFinalize(Me)
End Sub
```

Metoda `Dispose` implementuje `IDisposable.Dispose` (przez to wiemy, że `Implements IDisposable` jest zawarte w naszym `LoaderClass`). Lokalna zmienna statyczna jest wykorzystana w celu umożliwienia bezpiecznego wywoływania `Dispose` więcej niż jeden raz. Druga instrukcja ustawia odpowiednią wartość `Boolean`, wskazując, że `Disposed` była już wywołana. Metoda `LoaderClass.Close` jest wywoływana w celu zamknięcia `TextReader`, co nie zostało jeszcze pokazane, a `ArrayList` zostaje przypisany `Nothing`. Na końcu, `GC.SuppressFinalize` informuje GC, że nie ma potrzeby wywoływania metody `Finalize` dla danego obiektu.

Specjalne słowa kluczowe

Podczas dyskusji o klasach zauważyłeś pewnie częste korzystanie z dwóch specyficznych słów kluczowych — `MyBase` oraz `MyClass`. `MyBase`, jak już widziałeś, zezwala na wywoływanie metod w klasie bazowej tworzonej klasy, które mogą być przeciążane; rozwiązuje to problem niejednoznaczności klas.

`MyClass` jest przybliżonym odpowiednikiem referencji do samej siebie `Me`. `MyClass` zakłada, że jakiekolwiek metody wywołane z niej są zadeklarowane jako `NotOverridable`. `MyClass` wywołuje metodę, nie biorąc pod uwagę typu obiektu, jaki posiada on w trakcie działania aplikacji, efektywnie omijając polimorficzne zachowanie. Ze względu na to, że `MyClass` jest referencją do obiektu, nie możesz z niej korzystać w metodach `Shared`.

Referencja do siebie `Me` została przeniesiona do Visual Basic .NET. `Me` do działania wymaga egzemplarza. `MyClass` wywołuje metodę w tej samej klasie, natomiast `Me` wywołuje metodę w obiekcie przypisanym przez `Me`, to znaczy, w sposób polimorficzny. Przyjrzyj się następującym klasom.

```
Public Class Class1
  Public Overridable Sub Proc()
  End Sub
  Public Sub New()
    Me.Proc()
  End Sub
End Class

Public Class Class2
  Inherits Class1
  Public Overrides Sub Proc()
  End Sub
End Class
```

Gdy tworzony jest obiekt typu `Class2`, konstruktor w `Class1` wywołuje `Class2.Proc`. Jeśli `Me.Proc` zamienimy na `MyClass.Proc`, wówczas wywołana będzie `Class1.Proc`.

Dodawanie funkcji i procedur do metod

Metody są to funkcje i procedury, które zdefiniowane zostały w ramach klasy lub struktury. Jedynym wyzwaniem, które pojawia się podczas implementacji metod, jest opracowanie odpowiednich algorytmów rozwiązujących dany problem, napisanie dla nich jak najprostszego kodu oraz określenie dostępu do tworzonych metod wraz z użyciem odpowiednich modyfikatorów, które byłyby dla nich najbardziej odpowiednie.

Niestety, sposób definiowania metod jest zagadnieniem bardzo subiektywnym. Najlepszym sposobem na nauczenie się implementowania metod jest analizowanie i pisanie jak największej ilości kodu, a następnie wybranie stylu naszym zdaniem najlepszego. Nie bój się eksperymentować i zmieniać kod już napisany. Wydruk 7.6 przedstawia kompletną klasę `LoaderClass`.

Wydruk 7.6. ***Zawartość klasy LoaderClass***

```
 1: Imports System.IO
 2:
 3: Public Class LoaderClass
 4:   Implements IDisposable
 5:
 6:   Private FFileName As String
 7:   Private FReader As TextReader
 8:   Private FArrayList As ArrayList
 9:   Public Event OnText(ByVal Text As String)
10:
11:   Private Sub DoText(ByVal Text As String)
12:     RaiseEvent OnText(Text)
13:   End Sub
14:
15:   Public Property FileName() As String
16:     Get
17:       Return FFileName
18:     End Get
19:     Set(ByVal Value As String)
20:       FFileName = Value
21:     End Set
22:   End Property
23:
24:   Public Overloads Sub Open()
25:     If (FReader Is Nothing) Then
26:       FReader = File.OpenText(FFileName)
27:     Else
28:       Throw New ApplicationException("file is already open")
29:     End If
30:   End Sub
31:
32:   Public Overloads Sub Open(ByVal AFileName As String)
33:     FileName = AFileName
34:     Open()
35:   End Sub
36:
37:   Public Sub Close()
38:     If (FReader Is Nothing) Then Exit Sub
39:     FReader.Close()
40:     FReader = Nothing
41:   End Sub
42:
43:   Private Function Add(ByVal Text As String) As Boolean
44:     If (Text = "") Then Return False
45:     DoText(Text)
45:     FArrayList.Add(Text)
47:     Return True
48:   End Function
49:
50:   Private Function Reading() As Boolean
51:     Return Not FDisposed AndAlso Add(FReader.ReadLine())
52:   End Function
53:
54:   Public Sub Load()
55:     While (Reading())
56:       Application.DoEvents()
57:     End While
```

```
58:   End Sub
59:
60:   Public Sub New()
61:     MyBase.New()
62:     FArrayList = New ArrayList()
63:   End Sub
64:
65:   Public Sub New(ByVal AFileName As String)
66:     Me.New()
67:     FileName = AFileName
68:   End Sub
69:
70:   Private FDisposed As Boolean = False
71:
72:   Public Sub Dispose() Implements IDisposable.Dispose
73:     If (FDisposed) Then Exit Sub
74:     FDisposed = True
75:     Close()
76:     FArrayList = Nothing
77:   End Sub
78:
79:   Protected Overrides Sub Finalize()
80:     Dispose()
81:     MyBase.Finalize()
82:   End Sub
83:
84:   Public Shared Function Load(ByVal AFileName _
85:     As String) As LoaderClass
86:
87:     Dim ALoader As New LoaderClass(AFileName)
88:     ALoader.Open()
89:     ALoader.Load()
90:     Return ALoader
91:
92:   End Function
93: End Class
```

Klasa `LoaderClass` **definiuje metodę** `DoText`**, dwie metody** `Open` **oraz** `Close`, `Add`, `Reading` **i** `Load`. `DoText` **jest metodą prywatną; jej zadanie polega na wygenerowaniu zdarzenia w celu poinformowania o jakichkolwiek obiektach, które mogą chcieć podsłuchać proces ładowania. Obie metody** `Open` **są publiczne; zostały zdefiniowane z modyfikatorami** `Overloads`**, aby umożliwić konsumentom otwieranie pliku z przekazaniem jego nazwy w argumencie metody lub bez tego argumentu, zakładając w takim przypadku, że nazwa pliku została wcześniej przypisana w wywołaniu konstruktora lub poprzez modyfikację wartości właściwości. (Właściwość** `FileName` **jest zdefiniowana w wierszach 15. – 22.). Metoda** `Add` **eliminuje potrzebę istnienia zmiennej tymczasowej. Możemy przekazać wynik metody** `TextReader.ReadLine` **jako parametr do** `Add`**, która zwróci nam** `Boolean` **wskazujący, czy chcemy wartość czy nie. Funkcja** `Reading` **zwraca wartość** `Boolean` **określającą, że dodaliśmy tekst i że nie wywołaliśmy metody** `Dispose` **(wiersze 50. – 52.). Przy okazji w wierszu 51. widzimy przykład zastosowania operatora działania skróconego** `AndAlso`**. Jeśli metoda** `Dispose` **została wywołana,** `Not Disposed` **zostaje skrócona; w przeciwnym razie zostaje odczytywany następny wiersz tekstu. Jeśli** `FReader.ReadLine` **dojdzie do końca pliku, metoda** `ReadLine` **zwróci pusty ciąg (**`""`**), a** `Reading` **zwróci** `False`**. Ponieważ rozdzieliliśmy działanie** `Reading` **i** `Add`**, metoda** `Load` **jest bardzo prosta i składa się jedynie z pętli** `While` **(wiersze 54. – 58.).**

Jedynymi metodami publicznymi w naszej klasie są `Open`, `Close` i `Read`. Z tego względu klasa jest bardzo łatwa do obsługi przez konsumentów. Pozostałe metody wspomagają działanie metod publicznych i nie muszą być prezentowane użytkownikom, a tym bardziej przez nich wywoływane. Są one więc prywatne.

Porównaj pojedynczego muzyka z orkiestrą. Jeśli masz jednego muzyka, który gra na jednym instrumencie, rodzaj muzyki przez niego prezentowany jest siłą rzeczy ograniczony. Jeśli natomiast posiadasz całą orkiestrę z kilkudziesięcioosobowym składem, ich możliwości i różnorodność muzyki przez nich granej są nieporównywalnie większe.

Monolityczne metody są jak samotny muzyk; przyjemne, lecz niezbyt różnorodne. Wiele pojedynczych metod — myśląc o konstruktorach przeciążonych w roli sekcji dętej — oznacza, że każda metoda może się w czymś specjalizować i być wykorzystywana bardziej różnorodnie.

W naszym konkretnym przykładzie prawdopodobnie nie wykorzystamy ponownie metod `Add`, `Reading` czy `DoText`. To, co uzyskujesz dzięki pojedynczym funkcjom, to łatwość ich implementacji oraz małe prawdopodobieństwo, że z ich powodu wystąpią jakieś błędy. Kolejną zaletą, może nie już tak oczywistą, jest to, że takie metody mogą być przeciążane w podklasach. (Jeśli chciałbyś rozszerzyć na podklasę prywatne metody `LoaderClass`, musiałbyś zmienić je na chronione, co z drugiej strony jest czynnością trywialną). Jeśli natomiast będziesz korzystał z mniejszej liczby monolitycznych metod, będziesz miał mniej możliwości do ich przeciążania czy rozszerzania działalności.

Korzystanie z modyfikatorów metod

Modyfikatory metod pozwalają nam na uzyskanie większej kontroli nad ich implementacjami.

Przeciążanie metod

Modyfikator `Overloads` jest wykorzystywany do wskazania, że w danej klasie dwie lub więcej metod jest przeciążonych. Przykładem przeciążonej metody jest `LoaderClass.Open`.

Dzięki przeciążaniu możliwa jest implementacja metod, które wykonują semantycznie tę samą operację, ale różnią się liczbą lub typem argumentów. Weźmy dla przykładu metodę, która wypisuje tekst na konsolę, `Print`. Bez przeciążania musielibyśmy wprowadzić unikalne nazwy dla metod wypisujących poszczególne typy, na przykład `PrintInteger`, `PrintString`, `PrintDate`. Takie podejście wymagałoby od użytkowników nauczenia się nazw wielu metod, które wykonują to samo zadanie. Dzięki przeciążaniu wszystkie powyższe metody mogą nazywać się `Print`, a kompilator, na podstawie typu argumentu, sam wywoływałby odpowiednią metodę; pozostawmy więc wykonywanie nudnych zadań kompilatorowi, a sami zajmijmy się bardziej istotniejszymi sprawami.

Nagłówek procedury, liczba i typ argumentów oraz typ zwracany (jeśli procedura jest funkcją) tworzą wspólnie sygnaturę procedury.

Również właściwości mogą być przeciążane.

Oto podstawowe wskazówki dotyczące używania modyfikatora `Overloads`:

- Jeśli w Twoim kodzie występują metody przeciążone, użycie słowa `Overloads` jest niezbędne (na wydruku 7.6 możesz sprawdzić, w którym miejscu należy je stosować).
- Metody przeciążone muszą posiadać różną liczbę lub typ parametrów (Visual Basic .NET umożliwia przeciążanie metod z bardzo podobnymi typami, jak np. `Long` i `Integer`).
- Nie można przeciążać metod, zmieniając jedynie modyfikator dostępu, np. `Shared`.
- Nie można przeciążać metod, zmieniając tylko rodzaj przekazywanych argumentów (`ByVal` i `ByRef`)
- Nie możesz przeciążać metod, zmieniając wyłącznie nazwy parametrów, a pozostawiając takie same typy.
- Nie możesz przeciążać metod, modyfikując jedynie typ zwracany przez metodę, jednak oczywiście mogą istnieć dwie metody przeciążone różniące się typem zwracanej wartości. Na przykład, nie mogą być przeciążone procedura i funkcja o takich samych nazwach i argumentach, nie jest to również możliwe w przypadku dwóch funkcji z takimi samymi argumentami, które różnią się jedynie typem zwracanej wartości.

Jeśli znajdziesz się w sytuacji, gdy wykonywana operacja jest taka sama, a różni się typ danych, wówczas potrzebujesz metody przeciążonej. Gdy natomiast implementujesz dwie metody o takim samym kodzie, które różnią się jedynie wartością jednego lub kilku parametrów, zastosuj pojedynczą metodę z parametrem `Optional`.

Przesłanianie metod

Modyfikator `Overrides` umożliwia polimorfizm. Używasz go, gdy chcesz rozszerzyć lub zmodyfikować zachowanie się metody w klasie bazowej. Metoda klasy bazowej musi mieć taką samą sygnaturę jak metoda ją przesłaniająca.

Do modyfikatora `Overrides` powrócimy w rozdziale 10., „Dziedziczenie i polimorfizm”.

Modyfikatory Overridable, MustOverride i NotOverridable

Modyfikatory `Overridable`, `MustOverride` i `NotOverridable` są używane do obsługi metod, które *mogą* być przesłaniane i które *muszą* być przesłaniane.

`Overridable` i `NotOverridable` wzajemnie się wykluczają. Pierwszy z modyfikatorów, `Overridable`, wskazuje, że metoda może być przesłonięta. `NotOverridable` przy nazwie metody oznacza, że nie możesz jej przesłaniać. `MustOverride` wskazuje, że metoda jest abstrakcyjna, a klasy potomne muszą implementować tego typu metody w klasie macierzystej. Z `MustOverride` wynika jednoznacznie, że metoda jest przesłanialna, nie ma więc potrzeby stosowania w niej modyfikatora `Overridable`, a obecność `NotOverridable` w metodzie `MustOverride` nie ma żadnego sensu.

Metody `MustOverride` nie posiadają implementacji w klasie, w której zostały z takim modyfikatorem zadeklarowane. Metody te są odpowiednikiem czysto wirtualnych metod w C++ i wirtualnych metod abstrakcyjnych w Object Pascal; potomkowie muszą takie metody implementować.

Korzystanie z modyfikatora Shadows

Jeśli chcesz, aby klasa potomna mogła korzystać z nazwy poprzednio wprowadzonej w klasie nadrzędnej, skorzystaj ze słowa `Shadows`. Nazwy zakryte nie są usuwane z klasy macierzystej; słowo `Shadows` zezwala po prostu na powtórne wprowadzenie w klasie potomnej poprzednio użytej nazwy bez wystąpienia błędu kompilacji.

Składowe w klasie potomnej nie muszą być tego samego typu jak składowe zakryte; dwie składowe muszą mieć jedynie identyczne nazwy. Jakakolwiek składowa w klasie potomnej może zakryć dowolną składową klasy nadrzędnej. Metoda w klasie potomnej może zakryć pole, właściwość, metodę lub zdarzenie klasy nadrzędnej. Poniższy fragment demonstruje wykorzystanie modyfikatora `Shadows` do ponownego wprowadzenia metody w klasie potomnej z taką samą nazwą jak metoda w klasie nadrzędnej.

```
Public Class A
  Public Sub Foo()
  End Sub
End Class

Public Class B
  Inherits A
  Public Shadows Sub Foo()
  End Sub
End Class
```

Wydruk pokazuje dwie klasy, A i B. B jest podklasą A; to znaczy, A jest klasą nadrzędną (rodzicem) klasy B. Obie klasy posiadają metodę `Foo`. Skorzystanie ze słowa `Shadows` w `B.Foo` oznacza, że `B.Foo` zakrywa `A.Foo`. Jeśli posiadasz dwie identyczne nazwy w dwóch klasach związanych ze sobą dziedzicznością, musisz skorzystać albo z modyfikatora `Shadows`, albo z `Overrides`. (W rozdziale 10., „Dziedziczenie i polimorfizm", znajdziesz szczegółowe informacje na temat używania `Shadows` i `Overrides`).

Modyfikator Shared

Składowe `Shared` są dostępne bez potrzeby tworzenia egzemplarzy typów referencyjnych lub bezpośrednich — odpowiednio klas lub struktur. W rozdziale 11., „Składowe współdzielone", znajdziesz dokładne omówienie składowych współdzielonych.

Korzystanie ze specyfikatorów dostępu

Klasy wspierają bardzo mądre powiedzenie *divide et impera*, dziel i rządź. Jako konstruktor klasy, możesz skupić się podczas definiowania wyłącznie na sposobie jej implementacji. Gdy korzystasz z klasy, stajesz się jej konsumentem. W tym momencie interesują Cię jedynie składowe publiczne, a w przypadku klas potomnych — składowe publiczne i chronione.

Zaletą modyfikatorów dostępu jest to, że jako konsument nigdy nie musisz martwić się o składowe prywatne i zazwyczaj nie musisz również myśleć o składowych chronionych. Jeśli będziesz pamiętał o ogólnej zasadzie mówiącej o definiowaniu nie więcej niż sześciu składowych publicznych w pojedynczej klasie, jako konsument takiej klasy będziesz zwolniony z ciężaru większości detali implementacyjnych, które ukryte zostaną w składowych prywatnych i chronionych. Poprzez pewne zarządzanie kodem i stosowanie specyfikatorów dostępu ograniczających liczbę składowych, z jakimi muszą się uporać konsumenci, upraszczasz swój kod, przez co staje się on łatwiejszy do zarządzania i utrzymania.

W metodach możesz korzystać ze specyfikatorów dostępu `Public`, `Protected`, `Friend` i `Protected Friend`. Na początku tego rozdziału, w podrozdziale „Używanie specyfikatorów dostępu do klas", zostały omówione poszczególne specyfikatory. Poniższa lista bardzo krótko omawia ich wpływ na metody:

- Metody `Public` mogą być wywołane wewnętrznie lub przez dowolnego konsumenta.
- Metody `Protected` mogą być wywołane przez generalizatora (klasę potomną) lub wewnętrznie.
- Metody `Private` mogą być wywoływane wyłącznie wewnętrznie.
- Metody `Friend` mogą być wywołane jedynie w obrębie tej samej aplikacji.
- Metody `Protected Friend` mogą być wywołane wewnętrznie lub przez potomka w obrębie tej samej aplikacji.

W książce znajdziesz setki przykładów wykorzystujących specyfikatory dostępu, a kilkanaście w tym rozdziale. W pierwszym podrozdziale, „Definiowanie klas", znajdziesz ogólne uwagi na temat liczby metod i stosowania specyfikatorów.

Dodawanie zdarzeń do klas

Gdy chcesz poinformować konsumenta klasy, że coś się wewnątrz tej klasy wydarzyło, możesz to przedstawić za pomocą zdarzenia:

```
Public Event OnText(ByVal Text As String)

Private Sub DoText(ByVal Text As String)
  RaiseEvent OnText(Text)
End Sub
```

Klasa `LoaderClass` (zobacz wydruk 7.6) wywołuje zdarzenie `OnText` dla każdego wiersza odczytywanego z pliku tekstu. Zaletą korzystania ze zdarzeń jest to, że klasa nie musi wiedzieć, którzy konsumenci są zainteresowani otrzymywaniem informacji o zajściu zdarzenia. Kod `LoaderClass` nie zmienia się w zależności od tego, czy któryś z konsumentów obsługuje zdarzenie czy nie.

Instrukcja `Public Event OnText(ByVal Text As String)` tworzy niejawnie typ `Delegate`. Konsument, który chce obsłużyć zdarzenie `OnText` wygenerowane przez `LoaderClass`,

może do tego celu użyć instrukcji `WithEvents` lub `AddHandler`. Oto przykład użycia drugiej z nich:

```
AddHandler FLoader.OnText, AddressOf OnText
```

Obiekt zawierający metodę `OnText` (w powyższej instrukcji) jest dodawany do listy wywołań `Delegate` delegacji `FLoader.OnText`.

Zdarzenia nie są polimorficzne. Oznacza to, że nie możesz jawnie przeciążać metod w klasach potomnych. Ogólną strategią stosowaną w takim przypadku jest opakowanie instrukcji `RaiseEvents` w metodę i użycie tej metody (zobacz `DoText`) jako proxy w celu wywołania zdarzenia. Poprzez skorzystanie z proxy przy wywoływaniu zdarzeń, możemy przesłonić go w klasach potomnych, gdy tylko chcemy zmienić zachowanie się zdarzenia w momencie jego wywołania.

Obsługa zdarzeń zmieniła się znacząco w Visual Basic .NET, włączając w to wprowadzenie klas `Delegate` i `MulticastDelegate`, które bardzo usprawniły tę obsługę. Rozdział 8., „Dodawanie zdarzeń" oraz 9., „Delgacje" szczegółowo omawia tematy obsługi zdarzeń i delegacji.

Definiowanie klas zagnieżdżonych

Klasa zagnieżdżona jest po prostu klasą zdefiniowaną w klasie. Klasy zagnieżdżone mogą być definiowane ze wszystkich elementów dowolnej innej klasy. Oto szkielet kodu dla klasy zagnieżdżonej:

```
Public Class Outer
  Public Class Nested
  End Class
End Class
```

Celem tworzenia klas zagnieżdżonych jest zgromadzenie grupy elementów posiadających wspólne cechy w obrębie zawierającej je klasy. Na przykład, jeśli masz klasę i jakieś pola, właściwości i metody definiujące pewną koncepcję, jednak są one osobno, możesz zaimplementować klasę zagnieżdżoną, w której zgromadzisz te elementy.

Generalnie, nie będziesz często wykorzystywał klas zagnieżdżonych. Jakakolwiek implementacja, która może być zrobiona z użyciem klas zagnieżdżonych, może być z powodzeniem zrobiona bez nich. Jeśli temat ten Cię bardziej interesuje, możesz poczytać na temat bardziej zaawansowanych tworów, np. o idiomie list-koperta, w książce Jamesa Copliena „Advanced C++". Wydruk 7.7 przedstawia przykład klas zagnieżdżonych, prezentując sposoby ich implementacji i gramatykę języka.

Wydruk 7.7. ***Klasy zagnieżdżone, dziedziczenie i wielowątkowość w programie demonstracyjnym TrafficLight.sln***

```
1: Public Class Signal
2: #Region " Składowe publiczne "
3:   Public Event OnChange(ByVal sender As System.Object, _
4:     ByVal e As System.EventArgs)
```

```
 5:
 6:   Public Sub Draw(ByVal Graphic As Graphics)
 7:     Graphic.FillRectangle(Brushes.Brown, Rect)
 8:     DrawLights(Graphic)
 9:   End Sub
10:
11:   Public Sub New(ByVal Rect As Rectangle)
12:     FRect = Rect
13:     PositionLights()
14:     TurnOnGreen()
15:     CreateThread()
16:   End Sub
17:
18:   Public Sub Dispose()
19:     Static Done As Boolean = False
20:     If (Done) Then Exit Sub
21:     Done = True
22:     KillThread()
23:     GC.SuppressFinalize(Me)
24:   End Sub
25:
26:   Public Sub NextSignal()
27:     While (FThread.IsAlive)
28:       FThread.Sleep(SleepTime())
29:       ChangeState()
30:       DoChange()
31:     End While
32:   End Sub
33:
34: #End Region
35:
36: #Region " Składowe chronione "
37:
38:   ' Destruktor
39:   Protected Overrides Sub Finalize()
40:     Dispose()
41:   End Sub
42:
43:   Protected Sub CreateThread()
44:     FThread = New Threading.Thread(AddressOf NextSignal)
45:     FThread.IsBackground = True
46:     StartSignal()
47:   End Sub
48:
49:   Protected Sub StartSignal()
50:     FThread.Start()
51:   End Sub
52:
53:   Protected Sub StopSignal()
54:     FThread.Abort()
55:     FThread.Join()
56:   End Sub
57:
58:   Protected Sub KillThread()
59:     StopSignal()
60:     FThread = Nothing
61:   End Sub
```

```
62:
63:   Protected Sub DrawLights(ByVal Graphic As Graphics)
64:     Dim I As Integer
65:     For I = 0 To FLights.GetUpperBound(0)
66:       FLights(I).Draw(Graphic)
67:     Next
68:   End Sub
69:
70: #End Region
71:
72: #Region " Składowe prywatne "
73:   Private FLights() As Light = _
74:     {New GreenLight(), New YellowLight(), New RedLight()}
75:   Private FRect As Rectangle
76:   Private FThread As Threading.Thread
77:
78:   Private ReadOnly Property Green() As Light
79:     Get
80:       Return FLights(0)
81:     End Get
82:   End Property
83:
84:   Private ReadOnly Property Yellow() As Light
85:     Get
86:       Return FLights(1)
87:     End Get
88:   End Property
89:
90:   Private ReadOnly Property Red() As Light
91:     Get
92:       Return FLights(2)
93:     End Get
94:   End Property
95:
96:   Private ReadOnly Property Rect() As Rectangle
97:     Get
98:       Return FRect
99:     End Get
100:  End Property
101:
102:  Private Sub PositionLights()
103:
104:    Dim I As Integer
105:    For I = 0 To FLights.GetUpperBound(0)
106:      FLights(I).Rect = GetRect(I)
107:    Next
108:
109:  End Sub
110:
111:  Private Sub TurnOnGreen()
112:    Green.State = True
113:  End Sub
114:
115:  Private Sub DoChange()
116:    RaiseEvent OnChange(Me, Nothing)
117:  End Sub
118:
```

```
119:
120:   Private Function GetRect(ByVal Index As Integer) As Rectangle
121:     Return New Rectangle(Rect.Left + 10, _
122:         Rect.Top + 10 + CInt(Math.Round((2 - Index) / 3 * Rect.Height)), _
123:         Rect.Width - 20, CInt(Math.Round(Rect.Height / 3)) - 20)
124:   End Function
125:
126:   Private Sub ChangeState()
127:     Static Current As Integer = 0
128:     Current = (Current + 1) Mod 3
129:     Dim I As Integer
130:
131:     For I = FLights.GetLowerBound(0) To _
132:       FLights.GetUpperBound(0)
133:       FLights(I).State = I = Current
134:     Next
135:   End Sub
136:
137:   Private Function SleepTime() As Integer
138:     Static T() As Integer = {1000, 5000}
139:     If (Yellow.State) Then
140:       Return T(0)
141:     Else
142:       Return T(1)
143:     End If
144:   End Function
145:
146: #End Region
147:
148:
149: #Region " Składowe zagnieżdżone "
150:
151:   ' Virtual Abstract Nested Light Class
152:   ' Base class for the light signal light classes
153:
154:   Private MustInherit Class Light
155:     Public State As Boolean = False
156:     Public Rect As Rectangle
157:
158:     Protected MustOverride ReadOnly Property _
159:       BrushArray() As Brush()
160:
161:     Protected Function GetBrush() As Brush
162:       If (State) Then
163:         Return BrushArray(1)
164:       Else
165:         Return BrushArray(0)
166:       End If
167:     End Function
168:
169:     Public Sub Draw(ByVal Graphic As Graphics)
170:       Graphic.FillEllipse(GetBrush(), Rect)
171:     End Sub
172:
173:     Protected Overrides Sub Finalize()
174:       Rect = Nothing
175:     End Sub
176:   End Class
```

```
177:
178:   ' Zagnieżdżona klasa RedLight, dziedziczy z Light
179:   Private Class RedLight
180:     Inherits Light
181:
182:     Protected Overrides ReadOnly Property BrushArray() As Brush()
183:       Get
184:         Static Brush() As Brush = {Brushes.Gray, _
185:           Brushes.Red}
186:         Return Brush
187:       End Get
188:     End Property
189:   End Class
190:
191:   ' Zagnieżdżona klasa YellowLight, dziedziczy z Light
192:   Private Class YellowLight
193:     Inherits Light
194:
195:     Protected Overrides ReadOnly Property BrushArray() As Brush()
196:       Get
197:         Static Brush() As Brush = {Brushes.Gray, _
198:           Brushes.Yellow}
199:         Return Brush
200:       End Get
201:     End Property
202:
203:   End Class
204:
205:   ' Zagnieżdżona klasa GreenLight, dziedziczy z Light
206:   Private Class GreenLight
207:     Inherits Light
208:     Protected Overrides ReadOnly Property BrushArray() As Brush()
209:       Get
210:         Static Brush() As Brush = {Brushes.Gray, _
211:           Brushes.Green}
212:         Return Brush
213:       End Get
214:     End Property
215:   End Class
216: #End Region
217:
218: End Class
```

Ze względu na to, że wydruk ten jest bardzo długi, jego szczegółowe omówienie zostało podzielone na poszczególne sekcje. Każda z nich prezentuje pewien określony aspekt kodu.

Zrozumienie celu korzystania z klas zagnieżdżonych

Motywacja do używania klas zagnieżdżonych wydaje się być dosyć niejasna, zwłaszcza że każda zagnieżdżona klasa może być technicznie zaimplementowana jako zwykła klasa. Istnieją pewne ściśle określone powody do wprowadzania klas zagnieżdżonych, jednak jest ich zaledwie kilka i będziesz się z nimi bardzo rzadko spotykał.

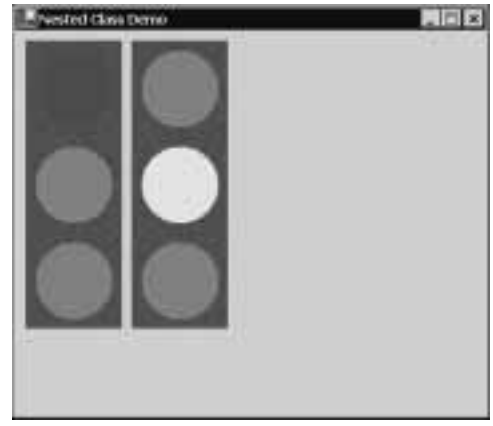
Aby zademonstrować techniczne aspekty wprowadzania klas zagnieżdżonych, napisałem przykładowy program *TrafficSignal.sln*. Światła sygnalizacji świetlnej — czerwone na górze, żółte w środku i zielone na dole — idealnie nadają się na elementy klasy zagnieżdżonej. Jest raczej mało prawdopodobne, że spotkasz na ulicy pojedynczy sygnał świetlny — tylko w jednym kolorze — którego stan będzie zależeć on innych świateł nie towarzyszących jemu. (Może istnieć pojedyncze żółte światło pulsujące, jednak w tym przypadku można je zaimplementować jako element osobnego sygnału świetlnego. Przykład ten ilustruje wątpliwości, jakie często można napotkać przy korzystaniu z klas zagnieżdżonych: za każdym razem, kiedy korzystasz z tego typu klasy, znajdzie się wyjątek, który sugeruje zrezygnowanie z zagnieżdżania).

`Signal` zawiera trzy egzemplarze klasy `Light`. Stanowią one sens jedynie jako całość (tak zakładamy w naszym przykładzie), potrzebujemy więc więcej niż jedno światło. Wprowadzimy je zatem jako klasę zagnieżdżoną i utworzymy trzy egzemplarze tej klasy.

Definiowanie klasy Signal

Klasa `Signal` reprezentuje trzystanowe, czerowno-żółto-zielone światło sygnalizacji drogowej. `Signal` będzie zdefiniowany tak, aby rysował na ekranie wyjściowym swój symbol za pomocą obiektu `Graphics` oraz będzie reprezentować poszczególne stany za pomocą odpowiednich kolorów.

Klasa `Signal` posiada tablicę trzech obiektów `Light`, których stany są ze sobą odpowiednio powiązane i będą się zmieniać tak, jak zmieniają się światła w rzeczywistości. Dwa sygnały z różnymi stanami są pokazane na rysunku 7.4.



Rysunek 7.4. *Dwa egzemplarze klasy Signal z rozwiązania TrafficSignal.sln*

Wszystkie składowe klasy `Signal` są zdefiniowane w wierszach 1. – 146. wydruku 7.7. W poniższych podrozdziałach omawiam poszczególne aspekty tej klasy.

Korzystanie z funkcji skracania kodu w celu lepszej organizacji

Pierwszą rzeczą, jaką zauważysz na wydruku 7.7, jest to, że wykorzystana została funkcja skracania kodu za pomocą dyrektywy `#Region`. Długim, skomplikowanym wydrukom

klas typu `Signal` na pewno przyda się trochę uporządkowania. Górna połowa wydruku zawiera składowe klasy `Signal`, a dolna — począwszy od wiersza 149. — definiuje klasy zagnieżdżone. Przez dodanie dyrektywy `#Region` możesz zwijać i rozwijać fragmenty kodu, ułatwiając sobie tym samym pracę z programem.

Warto również deklarować egzemplarze w kolejności ich ważności. Konsumentów interesują tylko składowe `Public`, więc je umieść jako pierwsze w kodzie. Generalizatorów dotyczą również składowe `Protected`, więc one niech będą po publicznych. Na końcu umieść składowe `Private`, które dostępne są jedynie twórcy klasy.

Takie uporządkowanie na pewno się przydaje, jednak w przypadku stosunkowo prostych i krótkich klas składowe mogą być umieszczane w kolejności ich tworzenia, bez zachowywania jakiejś szczególnej kolejności.

Jako ostatnie i najmniej znaczące w wydruku definiujemy klasy zagnieżdżone. Gdyby były one istotne dla konsumentów, nie byłyby klasami zagnieżdżonymi.

Definiowanie konstruktora i destruktora

Klasa `Signal` posiada trzy egzemplarze klasy `Light` i obiekt `Thread`. Z tego powodu konstruktor, destruktor oraz metody `Dispose` zostały dodane do klasy `Signal`. Konstruktor i `Dispose` zostały zdefiniowane odpowiednio w wierszach 11. – 16. i 18. – 24. Dla pewności, fragment ten został przedstawiony na wydruku 7.8. Metoda `Protected Finalize` służy do wywoływania metody `Dispose` — wiersze 39. – 41. — i nie została poniżej przedstawiona.

Wydruk 7.8. ***Konstruktor oraz metoda Dispose klasy Signal***

```
11:   Public Sub New(ByVal Rect As Rectangle)
12:     FRect = Rect
13:     PositionLights()
14:     TurnOnGreen()
15:     CreateThread()
16:   End Sub
17:
18:   Public Sub Dispose()
19:     Static Done As Boolean = False
20:     If (Done) Then Exit Sub
21:     Done = True
22:     KillThread()
23:     GC.SuppressFinalize(Me)
24:   End Sub
```

Konstruktor `Sub New` deleguje wszystkie swoje zadania do dobrze znanych metod, których nazwy bezpośrednio „mówią”, co jest w danej chwili robione. Inną strategią jest dodanie metody `Initialize` i wydelegowanie do niej całości kodu inicjującego; technika ta umożliwi ponowne zainicjowanie obiektu bez konieczności tworzenia nowego obiektu. W konstruktorze tym: określono, że ograniczający prostokąt jest przechowywany w polu, ustawiono światła `Lights` na określonej pozycji, włączono światło zielone oraz utworzony został wątek `Thread`.

Metoda `Dispose` wykorzystuje zmienną lokalną w roli wartownika. Dzięki niemu drugie i kolejne wywołanie `Dispose` nie wykonuje żadnych czynności. Za pierwszym razem natomiast metoda usuwa wszystkie wątki, które kontrolują stan świateł i powstrzymuje systemowy odzyskiwacz pamięci. Same światła nie są zwalniane, gdyż zostały utworzone na zewnątrz konstruktora w wierszach 73. i 74. poprzez element inicjujący tablicę.

Tworzenie wątka świateł

W wierszu 15. (zobacz wydruk 7.7 i 7.8) wywoływana jest metoda `CreateThread`. Wszystkie metody zarządzające wątkami zostały pokazane na wydruku 7.9, będącym fragmentem kodu z wydruku 7.7.

Wydruk 7.9. ***Metody obsługujące wątki w klasie Signal***

```
43:   Protected Sub CreateThread()
44:     FThread = New Threading.Thread(AddressOf NextSignal)
45:     FThread.IsBackground = True
46:     StartSignal()
47:   End Sub
48:
49:   Protected Sub StartSignal()
50:     FThread.Start()
51:   End Sub
52:
53:   Protected Sub StopSignal()
54:     FThread.Abort()
55:     FThread.Join()
56:   End Sub
57:
58:   Protected Sub KillThread()
59:     StopSignal()
60:     FThread = Nothing
61:   End Sub
```

W wierszu 44. tworzony jest egzemplarz klasy `System.Threading.Thread`. Konstruktor klasy `Thread` pobiera adres procedury `AddressOf`; procedura ta jest w miejscu, gdzie wątek się rozwidli podczas swojego startu. Wiersz 45. oznacza, że wątek staje się wątkiem w tle, umożliwiając aplikacji wyjście i zatrzymanie wątków sygnału. W wierszu 46. wywoływana jest metoda `StartSignal`, która załącza sygnał świetlny poprzez uruchomienie procedury z wiersza 49. Wraz z wywołaniem metody `Dispose` wywoływana jest `KillThread`. Z wydruku można wywnioskować, że `KillThread` uruchamia `StopSignal` i zwalnia obiekt wątka poprzez przypisanie go do `Nothing`. `StopSignal` wywołuje metodę `FThread.Abort`, a `FThread.Join` czeka na zakończenie działania wątku. `FThread.Abort` unicestwia wątek poprzez wywołanie wyjątku `ThreadAbortException`, którego nie da się przechwycić; wyjątek ten umożliwia wykonanie wszystkich bloków `Finally`, potrzebne jest więc wywołanie `Join`, aby upewnić się, że wątek został zakończony.

Wielowątkowość jest całkowicie nowym zagadnieniem w Visual Basic .NET. Jeśli wcześniej nie programowałeś w języku umożliwiającym obsługę wielu wątków, może to być dla Ciebie nie lada wyzwaniem. Klasa `Thread` i programowanie wielowątkowe w Visual Basic .NET zostały dokładnie omówione w rozdziale 14, „Aplikacje wielowątkowe".

Rysowanie sygnalizatora z wykorzystaniem zdarzenia

Obiekty `Signal` są rysowane dla każdej kontrolki, która przekaże swój obiekt `Graphics` metodzie `Signal.Draw`. (W rzeczywistości sygnał powinien być kontrolką, ale odeszlibyśmy znacząco od tematu klas zagnieżdżonych, dyskutując teraz o tym). Przyjrzyj się poniższym fragmentom kody z wydruku 7.7:

```
  6:   Public Sub Draw(ByVal Graphic As Graphics)
  7:     Graphic.FillRectangle(Brushes.Brown, Rect)
  8:     DrawLights(Graphic)
  9:   End Sub
...
 63:   Protected Sub DrawLights(ByVal Graphic As Graphics)
 64:     Dim I As Integer
 65:     For I = 0 To FLights.GetUpperBound(0)
 66:       FLights(I).Draw(Graphic)
 67:     Next
 68:   End Sub
...
161:     Protected Function GetBrush() As Brush
162:       If (State) Then
163:         Return BrushArray(1)
164:       Else
165:         Return BrushArray(0)
166:       End If
167:     End Function
168:
169:     Public Sub Draw(ByVal Graphic As Graphics)
170:       Graphic.FillEllipse(GetBrush(), Rect)
171:     End Sub
```

W tych trzech metodach zrealizowane jest rysowanie obiektów. W wierszach 6. – 9. `Signal.Draw` rysuje `Signal` jako `Rectangle` (prostokąt) i wywołuje `Signal.DrawLights` w celu dodania do prostokąta świateł. `DrawLights` wykorzystuje pętlę do narysowania każdego ze świateł, przy okazji prezentując dywersyfikację zadań oferowanych przez klasy i klasy zagnieżdżone. `Light.Draw` wywołuje `Graphics.FillEllipse` (używając argumentu `Graphic`), używając `Rect` jako obszaru ograniczającego dane światło oraz `Light.GetBrush` do określenia rodzaju pędzla na podstawie aktualnego stanu światła. Gdy wywoływana jest metoda rysująca, zajmuje się ona wyłącznie obiektami `Ellipses`, `Brushes` i `Rect`, a nie zmianą poszczególnych stanów czy obliczaniem granic obszarów.

Wiersze 163. i 165. wykorzystują polimorficzną właściwość `BrushArray` w celu uzyskania poprawnej tablicy obiektów pędzla na podstawie egzemplarza lampy. Jeśli `State = False`, zwracany jest pędzel wyłączony; w przeciwnym wypadku właściwość zwróci pędzel o określonym przez egzemplarz lampy kolorze.

Rozdział 10., „Dziedziczenie i polimorfizm”, zawiera bardziej obszerną dyskusję na temat dziedziczenia i zagadnień związanych z polimorfizmem; przedstawia również sposoby definiowania i implementacji interfejsów.

Definiowanie abstrakcyjnej klasy bazowej Light

`Light`, `GreenLight`, `YellowLight` oraz `RedLight` są klasami zagnieżdżonymi. `Light` wykorzystuje modyfikator `MustInherit`. Klasa `Light` jest przykładem abstrakcyjnych klas bazowych w Visual Basic .NET, co oznacza, że ma ona być raczej potomkiem i nie ma być utworzona bezpośrednio. Musi korzystać z modyfikatora `MustInherit`, ponieważ deklaruje abstrakcyjną właściwość:

```
Protected MustOverride ReadOnly Property BrushArray() As Brush()
```

Zauważ, że `BrushArray` w wierszach 158. i 159. na wydruku 7.7 nie posiada bloku procedury; jest to wyłącznie deklaracja. Instrukcje z `MustOverride` odnoszą się do wirtualnych i abstrakcyjnych składowych, co oznacza, że muszą być implementowane w klasie potomnej.

Wirtualne, abstrakcyjne składowe są analogią do deklaracji interfejsu w klasach COM. `GreenLight`, `YellowLight` i `RedLight` — każda z nich musi implementować właściwość `BrushArray`, w przeciwnym razie będą wirtualne i abstrakcyjne.

Instrukcje deklarujące są wykorzystywane w celu pomocy przy wprowadzaniu klas potomnych. Metoda `Draw` klasy `Light` zależy od `GetBrush`, a `GetBrush` w rezultacie zależy od tego, czy uda się uzyskać odpowiedni pędzel i kolor na podstawie stanu określonej lampy. W językach strukturalnych (lub niepoprawnie w językach obiektowych) ten rodzaj zachowania był implementowany z wykorzystaniem instrukcji `case`.

Implementowanie świateł sygnalizacji

Klasy świateł sygnalizacji same w sobie są łatwe do implementacji. Każde poszczególne światło wymaga jedynie dziedziczenia z `Light` oraz zaimplementowania właściwości tylko do odczytu `BrushArray`. Wydruk 7.10 przedstawia jedną z tych klas — `RedLight`. (Treść wszystkich świateł różni się jedynie wartościami wykorzystywanymi do zainicjowania `BrushArray`).

Wydruk 7.10. *Sposób implementacji jednego ze świateł*

```
178:   ' Zagnieżdżona klasa RedLight, dziedziczy z Light
179:   Private Class RedLight
180:     Inherits Light
181:
182:     Protected Overrides ReadOnly Property BrushArray() As Brush()
183:       Get
184:         Static Brush() As Brush = {Brushes.Gray, _
185:           Brushes.Red}
186:         Return Brush
187:       End Get
188:     End Property
189:   End Class
```

W wierszu 182. wprowadzona jest chroniona, przesłonięta i tylko do odczytu właściwość o nazwie `BrushArray`. Statyczna tablica pędzli jest inicjowana pędzlami globalnymi zdefiniowanymi w obiekcie `Brushes`. Klasa `RedLight` wykorzystuje pędzle koloru szarego i czerwonego.

Istnieje kilka sposobów, na jakie moglibyśmy zaimplementować kolorowe lampy sygnalizacji świetlnej. Moglibyśmy na przykład użyć jednej klasy `Light` i metod fabrycznych, które inicjują wartości pędzla na podstawie rodzaju światła; rodzaj ten mógłby być zdefiniowany z wykorzystaniem typu wyliczeniowego. W programowaniu, jak we wszystkich dziedzinach życia, musimy przyjąć pewien sposób postępowania. W tym rozdziale wybrałem podejście, które najlepiej obrazuje klasy zagnieżdżone. W rezultacie otrzymaliśmy ogólnie bardzo dobry kod.

Podsumowanie programu TrafficLight

W systemie produkcyjnym przykładowy program obsługi świateł sygnalizacyjnych nadawałby się lepiej jako aplikacja niż komponent. Jeśli modelowałbyś światła miejskie, na pewno chciałbyś jeszcze raz przemyśleć wszystkie wątki. W mieście wielkości Gliwic miałbyś 40 wątków, ale w Warszawie komputer eksplodowałby, przełączając wszystkie niezbędne do prawidłowego funkcjonowania miasta wątki. W dodatku światła sygnalizacyjne wymagają koordynacji. Lepszą implementacją byłby na przykład menedżer wątków, który koordynowałby światła na poszczególnych skrzyżowaniach oraz przydzielałby czasy trwania poszczególnym sygnałom.

W przypadku symulacji wizualnych aspektów zarządzania ruchem samochodowym należałoby również dopracować grafikę naszej aplikacji i dołączyć inne rzeczy związane z tym zagadnieniem.

Tworzenie egzemplarzy klas

Do tej pory spotkałeś się już z wieloma przykładami tworzenia egzemplarzy klas. Jednakże ten rozdział jest pierwszym, w którym temat klas omawiany jest oddzielnie. Przyjrzyjmy się więc jeszcze raz składni tworzenia obiektów.

Klasy są określane mianem typów referencyjnych. Typy takie, w przeciwieństwie do typów bezpośrednich, są tworzone poprzez deklarację zmiennej i użycie słowa `New`, a po nim nazwy klasy i nawiasów:

```
Dim reference As New Class()
```

Przykład ten wygląda, jak gdybyśmy wywoływali procedurę o nazwie `Class`. Jest to jednak forma służąca do konstrukcji obiektu. W rzeczywistości kod taki wywołuje konstruktora `Sub New()` zdefiniowanego w klasie `Class`. Jeśli masz parametry, które chcesz przekazać do konstruktora, umieszczasz je w nawiasach przy nazwie klasy, cały czas jest to jednak wywoływanie `Sub New`, tym razem tylko przeciążonej. Składnia taka może wprowadzać małe zamieszanie. Oto przykład tworzenia egzemplarza klasy `Signal` z poprzedniego podrozdziału:

```
Dim ASignal As New Signal(New Rectangle(10, 10, 100, 300))
```

Jeśli procedura pobiera obiekt, rozważ możliwość utworzenia obiektu — tak jak `New Rectangle(10, 10, 100, 300)` — podczas wywołania procedury. Technika ta pomoże w usuwaniu niepotrzebnych zmiennych tymczasowych.

Powyższa instrukcja deklaruje i inicjuje egzemplarz klasy `Signal`. Wywołuje ona konstruktora `Signal.New()`, przekazując egzemplarz nowego prostokąta (zobacz wiersz 11. na wydruku 7.7.). Taki sposób wywoływania jest dość dziwny, jednak decyzje w tej sprawie podejmował Microsoft podczas projektowania Visual Basic .NET.

W innych językach można spotkać inne formy konstruktorów i destruktorów. C++ używa operatora `new`, który może być przeciążony, przy czym konstruktor ma on taką samą nazwę jak klasa, a nazwa destruktora jest również nazwą klasy, lecz poprzedzoną znakiem tyldy (~). Object Pascal korzysta z metod `Create` i `Destroy`, ale w rzeczywistości konstruktory i destruktory oznacza słowami kluczowymi `constructor` i `destructor`.

Podsumowanie

Konstrukcja klasy różni się między VB6 a Visual Basic .NET. W VB6 klasy są interfejsami i literalnie używają słowa `interface`. Klasy Visual Basic .NET wykorzystują słówko `Class` i umożliwiają dziedziczenie, cechę wcześniej niespotykaną. Ten rozdział jako pierwszy z rozdziałów tej książki podejmuje się przedstawienia bardziej zaawansowanych zagadnień.

Przedstawiono tu podstawowe sposoby definiowania klas, dodawania pól, właściwości, zdarzeń i metod; pokazano, jak radzić sobie ze złożonością specyfikatorów dostępu i jak stosować modyfikatory. W tym rozdziale zostały pokazane zmiany wprowadzone w Visual Basic .NET do interfejsów, jak również sposoby implementacji metod interfejsu i klas zagnieżdżonych. Dowiedziałeś się, jak korzystać z wielowątkowości i dziedziczenia.

Rozdziały 8., „Dodawanie zdarzeń", i 9., „Delegacje", rozszerzają przedstawione w tym rozdziale zagadnienia zdarzeń i delegacji, a rozdział 10., „Dziedziczenie i polimorfizm", szczegółowo prezentuje dziedziczenie i polimorfizm. W rozdziale 12., „Definiowanie atrybutów", dowiesz się, jak wydajnie wykorzystywać atrybuty.