helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

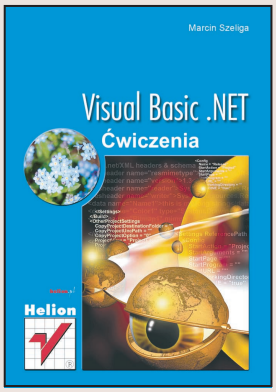
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Visual Basic .NET. Ćwiczenia



Autor: Marcin Szeliga
ISBN: 83-7361-432-X
Format: B5, stron: 176

Kolejna, siódma już wersja języka Visual Basic (Visual Basic .NET) to prawdziwa rewolucja — firma Microsoft opracowała jednolite środowisko programistyczne, a jednym z jego podstawowych języków (oprócz Visual C++ i Visual C#) jest właśnie Visual Basic. Dzięki wsparciu potężnej firmy, jaką jest Microsoft i rozpowszechnieniu systemu Windows, Visual Basic .NET stał się jednym z najczęściej używanych języków programowania na świecie.

Książka „Visual Basic .NET. Ćwiczenia" adresowana jest do osób, które chcą poznać podstawowe zasady tworzenia w języku VB .NET programów sterowanych zdarzeniami. Jak wszystkie książki z tej serii jest ona skonstruowana w formie praktycznych ćwiczeń, pozwalających stopniowo zagłębiać się w niuanse programowania bez zbędnych rozważań teoretycznych.

Poznasz:

- Platformę .NET
- Programowanie sterowane zdarzeniami
- Zmienne i stałe
- Sterowanie przebiegiem wykonywania programu
- Podstawy programowania obiektowego
- Korzystanie z baz danych
- Obsługę błędów w Visual Basic .NET

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

Spis treści

Wstęp 5

Rozdział 1. Microsoft Visual Studio .NET 9

Platforma .NET 9
.NET Framework 9
Serwery .NET 10
Visual Studio .NET 11
Visual Basic .NET 19

Rozdział 2. Programy sterowane zdarzeniami 20

Klasy 20
Obiekty 21
Konwencje nazewnicze 37
Zasady 37
Wskazówki 38
Jak czytelnie zapisywać kod programu? 38

Rozdział 3. Zmienne 41

Zmienne i stałe 41
Stałe 45
Operatory 46
Typy danych 49
Zmienne 50
Wskaźniki 50
Konwersja typów 51
Poprawność wprowadzanych przez użytkowników danych 54
Zakres zmiennych 55
Czas życia zmiennych 59
Struktury 60
Tablice 62
Konwencje nazewnicze 64
Zasady 64
Wskazówki 65

Rozdział 4. Procedury i funkcje ... 66
Procedury ... 66
Funkcje ... 73

Rozdział 5. Sterowanie wykonaniem programu ... 78
Instrukcje warunkowe ... 78
Instrukcja If … Then ... 79
Instrukcja Select Case ... 81
Pętle ... 84
Instrukcja For … Next ... 84
Instrukcja For Each … Next ... 87
Instrukcja Do … Loop ... 91

Rozdział 6. Programowanie obiektowe ... 93
Podejście proceduralne ... 93
Podejście obiektowe ... 94
Klasa ... 94
Obiekt ... 95
Abstrakcja ... 95
Dziedziczenie ... 95
Hermetyzacja ... 96
Interfejs ... 96
Polimorfizm ... 98
Projektowanie programów zbudowanych z obiektów ... 98
Projekt klas ... 99
Tworzenie programów zbudowanych z niezależnych obiektów ... 101
Podstawowe techniki programowania obiektowego ... 101
Składowe współdzielone ... 112
Zaawansowane techniki programowania obiektowego ... 114

Rozdział 7. Dane ... 120
Relacyjny model baz danych ... 120
XML ... 121
ADO .NET ... 122
Przestrzenie nazw ... 122
Klasy ... 122
Obiekt Connection ... 123
Obiekt Command ... 127
Kreatory danych ... 132
Lokalne kopie danych ... 136

Rozdział 8. Sprawdzanie poprawności danych ... 146
Uniemożliwianie użytkownikom programu wpisania niepoprawnych danych ... 147
Korzystamy z kontrolek ... 147
Korzystamy ze zdarzeń ... 154
Sprawdzanie poprawności poszczególnych danych ... 154
Sprawdzanie poprawności wszystkich danych ... 158

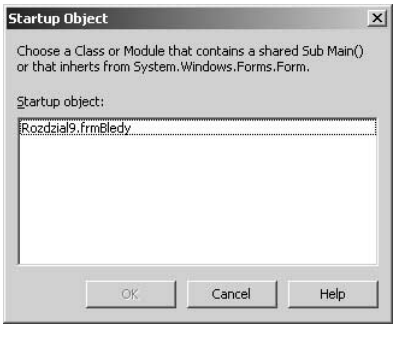
Rozdział 9. Wyszukiwanie i programowe przechwytywanie błędów ... 160
Wyszukiwanie błędów ... 160
Przechwytywanie błędów ... 171
Klasa Exception ... 172
Instrukcja Try … Catch … Finally ... 172

Ćwiczenie 9.1.

Wyszukujemy błędy syntaktyczne

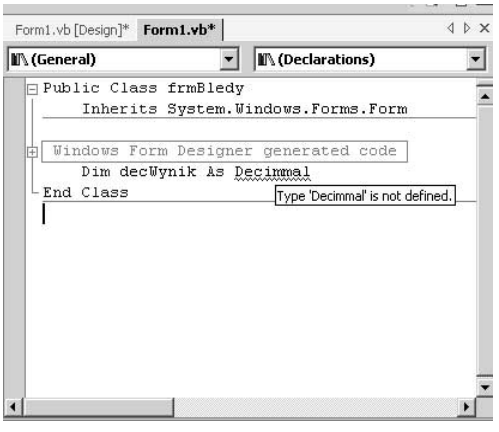
Błąd syntaktyczny polega na nieprzestrzeganiu reguł języka, w którym tworzony jest program. Na przykład, próba użycia instrukcji *For* bez odpowiadającej jej instrukcji *Next* jest błędem syntaktycznym.

1. Utwórz nowy projekt typu *Windows Application* i nazwij go `Rozdzial9`.
2. Zmień nazwę formularza *Form1* na `frmBledy`, jego etykietę na `Wyszukujemy i przechwytujemy błędy`, a rozmiar na `328; 112`.
3. W oknie zadań wyświetlony zostanie komunikat błędu — po zmianie nazwy formularza nie istnieje obiekt startowy projektu. **Okno zadań nie tylko informuje o błędach, ale również ułatwia ich wyszukanie i naprawienie**. Aby naprawić błąd:
   a. dwukrotnie kliknij lewym przyciskiem myszy opis błędu,
   b. wyświetlone zostanie okno dialogowe pozwalające na wybranie nowego obiektu startowego projektu (rysunek 9.1) — zaznacz formularz *frmBledy* i kliknij przycisk *OK*.



Rysunek 9.1. *W tym projekcie jest tylko jeden formularz i nie ma procedury o nazwie Main*

4. Po zmianie obiektu startowego okno zadań nie będzie zawierać żadnych komunikatów. Dodaj do formularza przycisk polecenia i ustaw następujące wartości jego atrybutów: *Name* — `cmdSilnia`, *Text* — `Silnia`, *Location* — `16; 16`.
5. Wyświetl okno edytora kodu.
6. Zadeklaruj zmienną formularza *decWynik* typu *Decimmal*: `Dim decWynik As Decimmal`.
7. Ponieważ przy deklaracji zmiennej popełniliśmy błąd, niewłaściwe słowo zostało podkreślone niebieską, falistą linią — aby dowiedzieć się, na czym polegał nasz błąd, ustaw kursor myszy nad zaznaczonym słowem (rysunek 9.2).
8. Zauważ, że ten sam komunikat — *Type 'Decimmal' is not definied.* — wyświetlany jest w oknie zadań. Napraw błąd przez podanie prawidłowej nazwy typu `Decimal`.
9. Utwórz procedurę zdarzenia *Click* przycisku *cmdSilnia*.
10. W ramach procedury zdarzenia:



Rysunek 9.2. *W Visual Studio wyświetlony zostanie krótki opis błędów syntaktycznych*

   a. zadeklaruj zmienną *intLiczba* typu *Integer*: `Dim intLiczba As Integer`,

   b. przypisz zadeklarowanej zmiennej liczbę podaną przez użytkownika (zwróć uwagę na błędną nazwę funkcji): `intLiczba = InpurBox("Podaj liczbę, której silnia ma zostać wyliczona", , 1)`.

11. Zadeklaruj funkcję *SilniaIteracja* oczekującą na argument *liczba* typu *Integer* i zwracającą dane typu *Double*: `Function SilniaIteracja(ByVal liczba As Integer) As Double`.

12. Usuń z sekcji deklaracji formularza deklarację zmiennej *decWynik*.

13. W ramach funkcji *SilniaIteracja*:

   a. zadeklaruj zmienną *dblWynik* typu *Double*: `Dim dblWynik As Double`,

   b. zadeklaruj zmienną licznik typu *Integer*: `Dim licznik As Integer`,

   c. w pętli powtarzanej przekazaną jako argument liczbę razy wylicz wartość zmiennej *decWynik* mnożąc ją za każdym razem przez wartość zmiennej licznik:

```
For licznik = 1 To liczba
    decWynik *= licznik
```

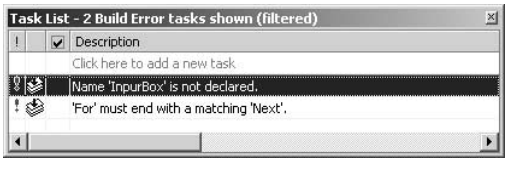
   d. zwróć wyliczoną silnię: `Return decWynik`.

14. W procedurze zdarzenia *cmdSilnia_Click* wywołaj funkcję *SilniaIteracja* z argumentem pobranym od użytkownika i wyświetl na ekranie otrzymany wynik: `MsgBox("Silnia wynosi " + SilniaIteracja(intLiczba))`.

15. W oknie zadań wyświetlone zostaną dwa komunikaty o błędach:

   a. pierwszy informuje o próbie odwołania się do nieistniejącej funkcji *InpurBox*,

   b. drugi — o brakującej instrukcji *Next* (rysunek 9.3).



Rysunek 9.3. *Lista błędów syntaktycznych (składniowych)*

16. Dwukrotnie kliknij pierwszy komunikat błędu — kursor zostanie ustawiony w miejscu występowania błędu. Popraw nazwę funkcji na `InputBox`.

17. W ten sam sposób zlokalizuj drugi błąd — tym razem zaznaczona zostanie instrukcja *For*, która nie została zakończona słowem kluczowym *Next*. Dopisz je przed zwracającą wynik działania funkcji instrukcją *Return*:

```
Function SilniaIteracja(ByVal liczba As Integer) As Double
        Dim dblWynik As Double
        Dim licznik As Integer
        For licznik = 1 To liczba
            decWynik *= licznik
        Next
        Return decWynik
End Function
```

18. Zapisz projekt.

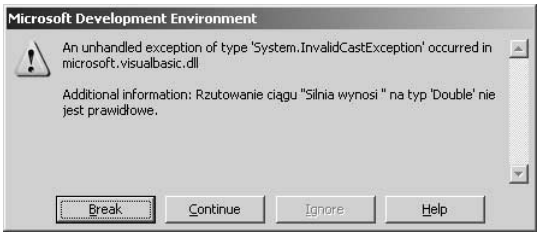
Ćwiczenie 9.2.

Wyszukujemy błędy konwersji typu

To, że w programie nie ma błędów syntaktycznych, nie oznacza, że będzie on działał. Wynika to stąd, że próba wykonania poprawnie zapisanych instrukcji może spowodować wystąpienie błędu wykonania, a błędy tego typu nie są zaznaczane podczas tworzenia programu.

1. Uruchom projekt `Rozdzial9`.

2. Po podaniu liczby, której silnię chcemy obliczyć, i kliknięciu przycisku *OK*, na ekranie wyświetlony zostanie komunikat o błędzie pokazany na rysunku 9.4.



Rysunek 9.4. *Uruchomienie poprawnego składniowo programu może zakończyć się błędem*

3. W tym przypadku błąd spowodowała próba połączenia ciągu znaków *"Silnia wynosi"* z liczbą typu *Double*. **Jeżeli w programie wystąpi błąd, jego działanie zostaje przerywane**, a na ekranie użytkownika wyświetlony jest niewiele mu mówiący komunikat o wystąpieniu błędu.

4. Kliknij przycisk *Break*. Wyświetlone zostanie okno edytora kodu z ***zaznaczoną na zielono instrukcją, której próba wykonania spowodowała błąd***.

5. W tym momencie program znajduje się w specjalnym trybie diagnostycznym, nazywanym trybem przerwania. Opis tego trybu znajduje się w dalszej części rozdziału, na razie zatrzymaj działanie tego trybu klikając znajdującą się na pasku narzędzi Debug ikonę *Stop Debugging* albo wybierając z menu *Debug* opcję *Stop Debugging*. W obu przypadkach tryb przerwania zostanie wyłączony i zniknie zaznaczenie błędnej instrukcji.

6. Popraw błąd przez skonwertowanie na dane tekstowe zwracanego przez funkcję *SilniaIteracja* wyniku:
`MsgBox("Silnia wynosi " + SilniaIteracja(intLiczba).ToString).`
7. Uruchom program i przetestuj jego działanie. Jak widać, **wyeliminowanie błędów wykonania nie oznacza, że program będzie działał prawidłowo**.
8. Zakończ działanie programu i zapisz wprowadzone w nim zmiany.

Ćwiczenie 9.3.

Wyszukujemy błędy związane z przekazaniem nieprawidłowych danych

1. Wyświetl projekt formularza *frmBledy*.
2. Utwórz nową funkcję *SilniaRekurencja* wywoływaną z jednym parametrem *liczba* typu *Integer* i zwracającą dane typu *Double*.
3. W ramach funkcji:
   a. zadeklaruj zmienną *dblWynik* typu *Double*,
   b. sprawdź, czy wartość parametru *liczba* jest mniejsza od zera i, jeżeli tak, zwróć wartość `1`, w przeciwnym razie wywołaj funkcję *SilniaRekurencja* z parametrem o jeden większym przypisując wynik jej wywołania do zmiennej *dblWynik*,



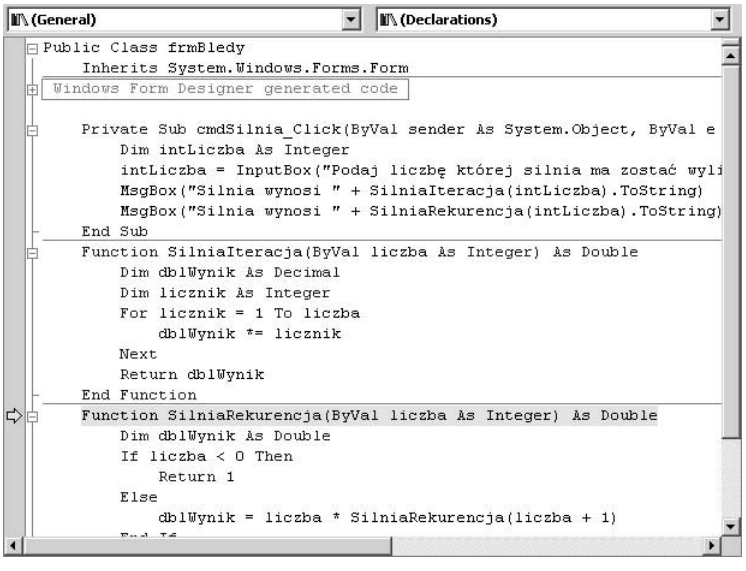
Wywołanie funkcji przez siebie samą jest charakterystyczne dla algorytmów rekurencyjnych.

   c. zwróć obliczoną wartość zmiennej *dblWynik*:

```
Function SilniaRekurencja(ByVal liczba As Integer) As Double
        Dim dblWynik As Double
        If liczba < 0 Then
            Return 1
        Else
            dblWynik = liczba * SilniaRekurencja(liczba + 1)
        End If
        Return dblWynik
End Function
```

4. W procedurze zdarzenia *cmdSilnia_Click* wywołaj funkcję *SilniaRekurencja* z argumentem pobranym od użytkownika i wyświetl na ekranie otrzymany wynik:
`MsgBox("Silnia wynosi " + SilniaRekurencja(intLiczba).ToString).`
5. Uruchom program. Po chwili jego działanie zostanie przerwane, a na ekranie zostanie wyświetlony komunikat o błędzie z informacją o przepełnieniu stosu. Naciśnij klawisz *Break*.
6. Tym razem na żółto zaznaczona została deklaracja funkcji, której wywołanie spowodowało wystąpienie błędu. W naszym przypadku funkcja wywoływała samą siebie bez końca (rysunek 9.5).
7. Przerwij tryb diagnostyczny i popraw wywołanie funkcji *SilniaRekurencja* na następujące: `dblWynik = liczba * SilniaRekurencja(liczba - 1).`

Rysunek 9.5. *Korzystając z rekurencji musimy tak sformułować warunek wywoływania funkcji przez samą siebie, aby proces zagnieżdżania został kiedyś zakończony*



8. Uruchom program i przetestuj jego działanie. Po raz kolejny okazuje się, że działający program nie musi być dobrze działającym programem.
9. Zakończ działanie programu i zapisz wprowadzone w nim zmiany.

Ćwiczenie 9.4.

Uruchamiamy tryb przerwania

Najłatwiejsze jest znalezienie i poprawienie błędów syntaktycznych — w tym przypadku przeważającą część pracy wykona za nas Visual Studio. Znalezienie i poprawienie błędów wykonania jest już trudniejsze i wymaga uruchamiania programu z różnymi zestawami testowymi danych wejściowych. Najtrudniejsze jest jednak znalezienie i poprawienie błędów logicznych, ponieważ ich wystąpienie nie powoduje przerwania działania programu, ale jego nieprawidłowe działanie. Wyszukiwanie błędów tego typu ułatwia specjalny tryb przerwania (tryb diagnostyczny) i dostępne w tym trybie narzędzia.



Tryb przerwania jest dostępny tylko wtedy, gdy program został skompilowany w wersji *Debug*. Po zmianie trybu na *Release* kompilator wygeneruje kod wynikowy, z którego usunięte zostaną wszystkie informacje diagnostyczne. Ta wersja programu przeznaczona jest dla użytkowników końcowych.

1. Wyświetl okno edytora kodu formularza *frmBledy*.
2. Możemy przejść w tryb przerwania na kilka sposobów. Po pierwsze, ***wstawiając w wykonywalnym wierszu programu pułapkę*** (ang. *Breakpoint*) — najprościej wstawić ją przez kliknięcie lewego marginesu okna edytora kodu. Kliknij lewy margines na wysokości instrukcji `intLiczba = InputBox("Podaj liczbę, której silnia ma zostać wyliczona", , 1)`. Wybrany wiersz zostanie zaznaczony.

3. Uruchom program. Po kliknięciu przycisku *Silnia* jego działanie zostanie wstrzymane, a na ekranie wyświetlone zostanie okno edytora kodu z zaznaczoną na żółto wykonywaną właśnie instrukcją.

4. ***Skasuj ustawioną pułapkę przez kliknięcie jej znaku wyświetlanego na lewym marginesie***.

5. Wznów działanie programu (np. naciskając klawisz *F5* lub klikając ikonę paska narzędzi *Debug Contine*).

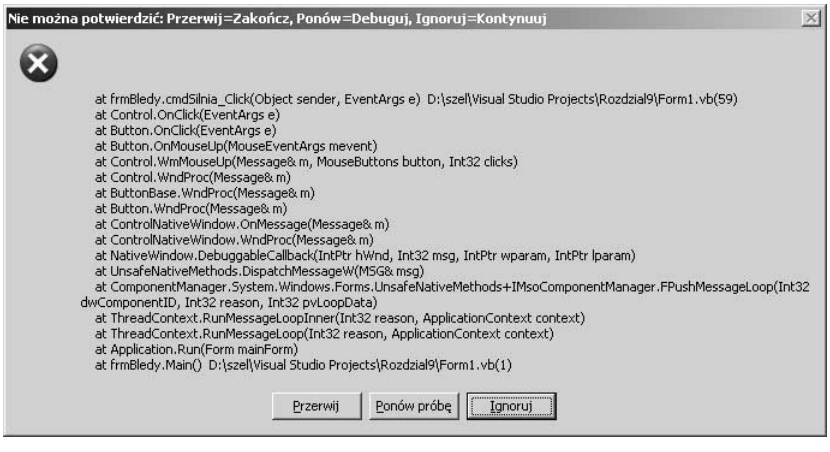
6. Przerwij działanie programu. ***Innym sposobem na przejście w tryb przerwania jest wykonanie instrukcji*** *Stop*. Wpisz tę instrukcję przed wywołaniem funkcji *SilniaIteracja* i uruchom program.

7. Tym razem jego działanie zostało przerwane po zamknięciu okna dialogowego umożliwiającego podanie liczby i ponownie zaznaczona na żółto jest ostatnio wykonywania instrukcja — instrukcja *Stop*.

8. **W trybie przerwania nie jest możliwa zmiana kodu programu**. Przerwij jego działanie i usuń instrukcję *Stop*.

9. ***Sposobem na warunkowe przejście w tryb przerwania jest wywołanie metody*** *Assert* ***obiektu specjalnego*** *Debug*. Działanie programu zostanie wstrzymane, jeżeli **wynikiem testu logicznego będzie fałsz**. W miejsce usuniętej instrukcji *Stop* wpisz: `Debug.Assert(intLiczba > 1)`.

10. Uruchom program i podaj liczbę większą niż 1. Ponieważ warunek został spełniony, działanie programu nie zostało wstrzymane. Raz jeszcze kliknij przycisk *Silnia*, ale nie zmieniaj domyślnej liczby 1. Tym razem warunek nie został spełniony (1 nie jest większe niż 1) i wyświetlone zostało okno umożliwiające wstrzymanie działania programu (rysunek 9.6).



Rysunek 9.6. *Wyjątkowo nieeleganckie rozwiązanie — na pasku tytułu okna wyświetlone zostało zmienione znaczenie przycisków. W tym przypadku kliknięcie przycisku Przerwij kończy działanie programu (uruchamia tryb projektowania), kliknięcie przycisku Ponów próbę uruchamia tryb diagnostyczny, a kliknięcie przycisku Ignoruj kontynuuje wykonywanie programu*

11. Kliknij przycisk *Ponów próbę* — wyświetlone zostanie okno edytora kodu z zaznaczoną ostatnio wykonywaną instrukcją.

12. Zatrzymaj działanie trybu przerwania i skasuj dodany wiersz.

13. ***Możemy w każdej chwili wstrzymać działanie programu naciskając kombinację klawiszy*** *Ctr+Break* ***lub klikając ikonę paska narzędzi*** *Debug Break All*.

Ćwiczenie 9.5.

Modyfikujemy pułapki

Visual Studio pozwala na warunkowe wstrzymywanie działania programu za pomocą ustawionych w nim pułapek, dzięki czemu niektóre błędy mogą zostać wykryte bez **konieczności krokowego uruchamiania programu** (uruchamiania programu wiersz po wierszu). Wykonując bieżące ćwiczenie, nauczymy się konfigurować ustawione pułapki.

1. Wyświetl okno edytora kodu formularza *frmBledy*.

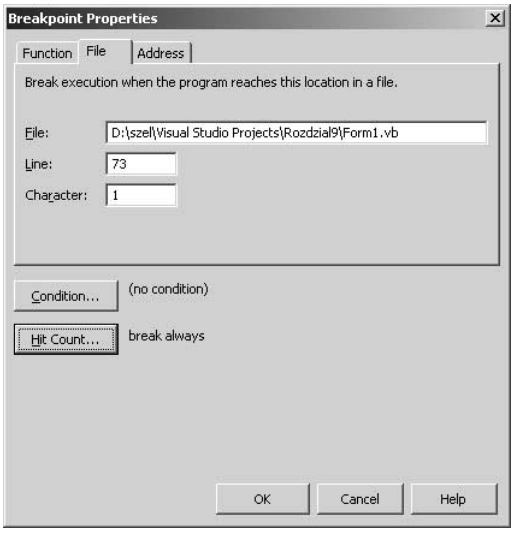
2. Ustaw pułapkę w wierszu `If liczba < 0 Then` funkcji *SilniaRekurencja*.

3. Przywróć poprzednie, błędne wywołanie funkcji:
`dblWynik = liczba * SilniaRekurencja(liczba + 1)`.

4. Kliknij prawym przyciskiem myszy symbol koła oznaczający ustawioną pułapkę i z menu kontekstowego wybierz opcję *Breakpoint Properties…*.

5. Wyświetlone zostanie okno dialogowe pokazane na rysunku 9.7. Pozwala ono na warunkowe wstrzymanie programu oraz na wstrzymanie programu po którymś z kolei wykonaniu instrukcji, dla której ustawiono pułapkę.

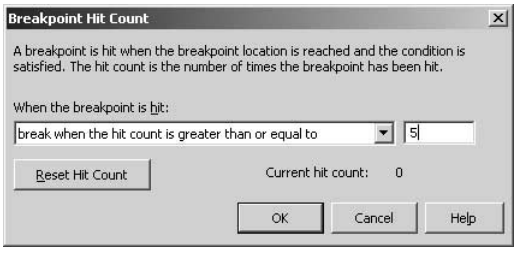
Rysunek 9.7.
Okno właściwości ustawionej pułapki



6. Kliknij przycisk *Hit Count…*.

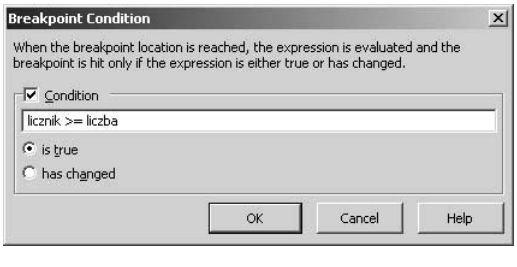
7. Wyświetlone zostanie okno dialogowe pozwalające określić, kiedy (po ilu wykonaniach instrukcji) wstrzymać działanie programu. Skonfiguruj pułapkę tak, aby działanie programu zostało wstrzymane po 5 lub więcej wywołaniach (rysunek 9.8).

Rysunek 9.8.
Często pojedyncze wykonanie instrukcji jest poprawne, a wielokrotne jej wykonanie świadczy o błędzie programu



8. Uruchom program i oblicz silnię liczby 4.
9. Działanie programu zostało wstrzymane. Ponieważ obliczenie silni liczby 4 nie wymaga pięciu lub więcej wywołań funkcji *SilniaRekurencja* (4! = 1 ∗ 2 ∗ 3 ∗ 4) błędu należy szukać albo w źle określonym warunku brzegowym, albo w sposobie wywołania funkcji *SilniaRekurencja*.
10. Ustaw kursor nad nazwą zmiennej *liczba*. Zostanie wyświetlona aktualna wartość zmiennej (`8`). Aby dowiedzieć się, ile razy instrukcja została wykonana, wyświetl okienko *Breakpoints*. Można to zrobić klikając znajdującą się na pasku narzędzi *Debug* ikonę *Breakpoints*.
11. Licznik pułapki wskazuje wartość `5`, czyli warunek *If liczba < 0 Then* sprawdzany jest po raz piąty. Skoro wartość zmiennej licznik wynosi `8`, a warunek sprawdzany jest piąty raz przy obliczaniu silni liczby 4 błąd znajduje się w sposobie wywołania funkcji, a nie w sposobie sprawdzania warunku brzegowego rekurencji. Rzeczywiście tak jest — zamiast przy kolejnych wywołaniach zmniejszać liczbę o jeden, my dodajemy do niej jeden, co powoduje, że warunek brzegowy nigdy nie zostanie spełniony.
12. Popraw znaleziony błąd i usuń pułapkę.
13. Ustaw pułapkę w wierszu `dblWynik *= licznik` funkcji *SilniaIteracja*.
14. Wyświetl właściwości pułapki i kliknij przycisk *Condition…*.
15. Zostanie wyświetlone okno dialogowe pozwalające na określenie warunku logicznego, którego spełnienie spowoduje wyzwolenie pułapki (rysunek 9.9).

Rysunek 9.9.
Warunkowe wywoływanie ustawionej pułapki



16. Wpisz warunek, którego spełnienie spowoduje wywołanie pułapki: `licznik >=liczba`. W ten sposób działanie programu zostanie wstrzymane, jeżeli pętla będzie wykonana tyle samo lub więcej razy, niż podana przez użytkownika liczba, której silnia ma zostać wyliczona.
17. Uruchom program. Po chwili jego działanie zostanie wstrzymane.

18. Odczytując (przez ustawienie kursora myszy) wartości zmiennych *licznik* i *dblWynik* dowiemy się, że niezależnie od tego, ile razy wykonywane jest mnożenie, wartość zmiennej *dblWynik* zawsze jest równa *0*.
19. Pewnie już domyślasz się, gdzie znajduje się błąd w tej funkcji. Nie poprawiaj go na razie — wykorzystamy go w następnym ćwiczeniu.
20. Skasuj ustawioną pułapkę.

Ćwiczenie 9.6.

Poznajemy tryb przerwania

Tryb przerwania pozwala na:

- ❖ wykonywanie programu wiersz po wierszu,
- ❖ śledzenie wyników wykonania poszczególnych instrukcji wykonania,
- ❖ odczytywanie i modyfikowanie wartości zmiennych,
- ❖ natychmiastowe wykonanie dowolnej instrukcji, procedury lub funkcji.

Operacje te możemy wykonywać za pomocą paska narzędzi *Debug* (rysunek 9.10). Poszczególne ikony (zaczynając od lewej) umożliwiają:

Rysunek 9.10. *Pasek narzędzi diagnostycznych*

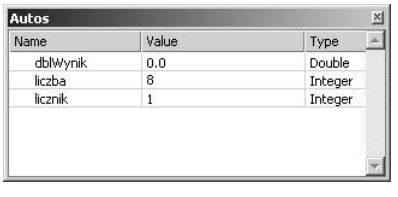


1. Uruchomienie programu.
2. Włączenie trybu przerwania.
3. Zatrzymanie wykonywania programu.
4. Wznowienie wykonywania programu.
5. Ustawienie kursora w wierszu instrukcji, która właśnie będzie wykonana.
6. Trzy kolejne przyciski służą do sterowania wykonaniem programu:
   a. wybranie opcji *Step Into* powoduje wykonanie programu instrukcja po instrukcji; jeżeli wywołana zostanie procedura lub funkcja, jej wykonanie również będzie odbywało się instrukcja po instrukcji,
   b. kliknięcie ikony *Step Over* także spowoduje wykonanie programu instrukcja po instrukcji; jeżeli jednak zostanie wywołana podprocedura lub funkcja, zostanie ona wykonana w całości, tak jakby była pojedynczą instrukcją,
   c. wybranie opcji *Step Out* spowoduje wykonanie całej podprocedury lub wywoływanej funkcji i ustawienie kursora w następnym wierszu procedury nadrzędnej.
7. Wyświetlanie danych w postaci heksadecymalnej.

8. Wyświetlenie okna *Breakpoints* oraz, po kliknięciu skierowanej w dół strzałki, m.in. okienek:
   a. *Locals* zawierającego nazwę, aktualną wartość i typ wszystkich zmiennych zdefiniowanych dla procedury, w której nastąpiło przerwanie wykonywania programu,
   b. *Immediate* pozwalającego na wykonywanie dowolnych instrukcji języka Visual Basic lub dyrektyw kompilatora,
   c. *Autos* pozwalającego na śledzenie aktualnych wartości wybranych zmiennych,
   d. *Call Stack* zawierającego hierarchię nazw wszystkich procedur lub funkcji, z których wywołany został zaznaczony fragment podprogramu.

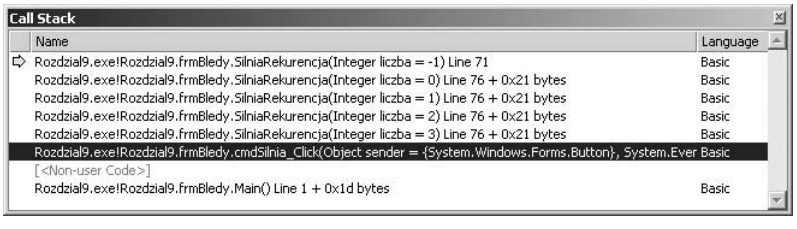
1. Ustaw pułapkę w wierszu `dblWynik *= licznik` funkcji *SilniaIteracja*.
2. Uruchom program i podaj liczbę `8`.
3. Działanie programu zostanie wstrzymane i wyświetlone zostanie okno edytora kodu.
4. Wyświetl na ekranie okno *Autos* (rysunek 9.11).

Rysunek 9.11. *Okno Autos pozwala odczytywać i modyfikować wartości zmiennych w trybie przerwania programu*



5. W tym momencie wynik równa się *0*, pętla jest wykonywana po raz pierwszy, a przekazaną liczbą jest *8*. Wykonaj następną instrukcję naciskając klawisz *F8* lub klikając ikonę paska narzędzi *Debug Step Into*.
6. Po wykonaniu mnożenia *dblWynik *= licznik* wartość zmiennej *dblWynik* nadal wynosi zero. Przekonaj się, wykonując kilka kolejnych instrukcji, że jej wartość nigdy się nie zmienia.
7. W tym przypadku błąd polegał na braku przypisania wartości początkowej `1` zmiennej *dblWynik*. Domyślnie, po zadeklarowaniu zmiennej liczbowej uzyskuje ona wartość *0*, a ponieważ zero pomnożone przez dowolną liczbę daje zero, nasza funkcja zwraca zero niezależnie od wartości argumentu jej wywołania. Przekonaj się, czy nasze podejrzenia są słuszne ***zmieniając w oknie** Autos **wartość zmiennej** dblWynik **na** `1` **i wykonując kilka kolejnych iteracji***.
8. Zatrzymaj tryb przerwania i zmień deklarację zmiennej *dblWynik*: `Dim dblWynik As Double = 1`. **Dobrą praktyką programowania jest jawne nadawanie zmiennym, podczas ich deklarowania, wartości początkowych**.
9. Ponownie uruchom program.
10. Prześledź aktualne wartości zmiennych w trakcie wykonywania kolejnych iteracji.
11. Zatrzymaj działanie programu, skasuj ustawioną pułapkę i raz jeszcze przetestuj jego działanie.

12. Pierwsza funkcja działa już poprawnie, druga nadal zwraca zero. W jej przypadku błąd nie jest spowodowany brakiem nadania wartości początkowej zmiennej.
13. Ustaw pułapkę w wierszu `If liczba < 0 Then` funkcji *SilniaRekurencja* i uruchom program.
14. Oblicz silnię liczby 3. Pierwsza funkcja zwróci wynik, a następnie działanie programu zostanie wstrzymane.
15. Tym razem do wyszukania błędu posłużymy się oknem *Call Stack*. Skoro algorytmy rekurencyjne polegają na wielokrotnym wywoływaniu danej procedury lub funkcji (zagnieżdżaniu wywołań aż do osiągnięcia warunku brzegowego, po czym następuje cykliczne przekazywanie obliczonych wartości zewnętrznym wywołaniom), śledzenie ich działania umożliwi właśnie to okno.
16. Wykonuj krok po kroku funkcję śledząc jednocześnie w oknie *Call Stack* poziom jej zagnieżdżenia (rysunek 9.12).



Rysunek 9.12. *Dzięki oknie Call Stack dowiedzieliśmy się, że funkcja SilniaRekurencja wywoływana jest o jeden raz za dużo — do obliczenia silni 3 powinna zostać wywołana 4 razy (jedno wywołanie zewnętrzne i trzy rekurencyjne: 3! = 1 ∗ 2 ∗ 3)*

17. Przekonaj się, w jaki sposób, po osiągnięciu warunku brzegowego, kolejno obliczone wartości przekazywane są zewnętrznym wywołaniom funkcji i zatrzymaj działanie programu. Wykorzystaj do tego celu możliwość wykonywania programu instrukcja po instrukcji.
18. Popraw definicję warunku brzegowego rekurencji: `If liczba = 0 Then`.
19. Usuń pułapkę i uruchom program.
20. Wylicz silnię kilku liczb.
21. Zakończ działanie programu i zapisz wprowadzone w nim zmiany.

Przechwytywanie błędów

Program, który doskonale działa w przypadku 95% danych, może nie działać albo działać niepoprawnie (zwracać fałszywe wyniki) dla nietypowych danych wejściowych lub w nietypowych sytuacjach. Tworząc program, powinniśmy zabezpieczyć się przed takimi błędami.

Klasa Exception

Visual Basic po raz pierwszy, dzięki zaimplementowaniu bazowej klasy platformy .NET *Exception* (wyjątek), pozwala na strukturalną obsługę błędów. ***Wyjątek jest wywoływany z tego miejsca programu, w którym wystąpił błąd wykonania***. W rezultacie, tworzony jest obiekt specjalny *Exception*, którego wartości atrybutów możemy programowo odczytać.

Najważniejsze atrybuty obiektu *Exception* to:

- *Message* — zwraca komunikat błędu,
- *Source* — zwraca oraz pozwala ustawić nazwę programu, w ramach którego wystąpił błąd,
- *StackTrace* — zwraca listę metod, których wywołanie spowodowało wystąpienie błędu.

Instrukcja Try ... Catch ... Finally

Strukturalna obsługa błędów polega na ich przechwytywaniu i reagowaniu na błędy wykonania programu. Po przechwyceniu możemy sprawdzić, jaki wyjątek został wywołany (jaki błąd wystąpił) i spróbować programowo go naprawić.



Jeżeli w programie wystąpi nieprzechwycony błąd wykonania, działanie programu zostaje zatrzymane, a na ekranie użytkownika wyświetlany jest systemowy komunikat o błędzie.

Ćwiczenie 9.7.

Odczytujemy komunikaty o błędach

Instrukcje, których wykonanie może spowodować wystąpienie błędu, należy zabezpieczać umieszczając je w bloku *Try* instrukcji *Try ... Catch ... Finally*. Następnie, w ramach bloku *Catch* należy sprawdzić, jaki wyjątek został wywołany i prawidłowo na niego zareagować.

1. Wyświetl okno edytora kodu formularza *frmBledy*.
2. Instrukcją, której wykonanie może spowodować wystąpienie błędu, jest przypisanie do zmiennej liczbowej podanej przez użytkownika wartości. Dodaj nowy wiersz powyżej instrukcji: `intLiczba = InputBox("Podaj liczbę, której silnia ma zostać wyliczona", , 1)`.
3. Wpisz `Try` i naciśnij *Enter*.
4. Zostaną dodane klauzule *Catch* oraz *End Try*. Przenieś do sekcji *Try* instrukcję, której wykonanie powinno zostać zabezpieczone.
5. W klauzuli *Catch* wyświetl użytkownikowi opis błędu i miejsca jego wystąpienia:

```
Try
          intLiczba = InputBox("Podaj liczbę, której silnia ma zostać wyliczona", , 1)
Catch ex As Exception
          MsgBox("Wystąpił błąd: " + ex.Message + vbCrLf _
          + "Wywołany przez: " + ex.StackTrace)
End Try
```



Jeżeli planujesz jedynie przechwycić błąd i zapobiec w ten sposób przerwaniu działania programu, pozostaw pustą klauzulę *Catch*.

6. Uruchom program i przetestuj jego działanie przez podanie nieprawidłowych danych wejściowych. Zwróć uwagę, że działanie programu nie jest dłużej przerywane, a po podaniu nieprawidłowych danych przyjmowana jest domyślna wartość 1.
7. Zakończ działanie programu i zapisz wprowadzone w nim zmiany.

Ćwiczenie 9.8.

Sprawdzamy typ błędów

Klasa *Exception* jest klasą bazową dla następujących klas:

- *System.ApplicationException*,
- *System.IO.IsolatedStorage.IsolatedStorageException*,
- *System.Runtime.Remoting.MetadataServices.SUDSGeneratorException*,
- *System.Runtime.Remoting.MetadataServices.SUDSParserException*,
- *System.SystemException*,
- *System.Windows.Forms.AxHost.InvalidActiveXStateException*.

Najciekawsza z naszego punktu widzenia klasa *SystemException* jest z kolei klasą bazową dla kilkudziesięciu klas definiujących konkretne typy błędów, np.:

- *System.ArithmeticException*,
- *System.IndexOutOfRangeException*,
- *System.InvalidCastException*,
- *System.NullReferenceException*,
- *System.OutOfMemoryException*,
- *System.UnauthorizedAccessException*.

Umożliwiają nam one dokładne sprawdzenie typu błędu wykonania i odpowiednie zareagowanie.

1. Wyświetl projekt formularza *frmBledy*.
2. Dodaj do formularza przycisk polecenia i ustaw następujące wartości jego atrybutów: *Name* — `cmdOtworz`, *Text* — `Otwórz`, *Location* — `16; 56`.
3. Utwórz procedurę zdarzenia *Click* przycisku *cmdOtwórz*.
4. W ramach procedury zdarzenia spróbujemy otworzyć plik tekstowy. W trakcie wykonywania tej operacji może wystąpić kilka błędów: plik o podanej nazwie może nie istnieć, niewłaściwie może zostać podana jego lokalizacja, sam plik może zwierać niepoprawne dane itp. Dlatego pierwszą instrukcją procedury powinna być instrukcja *Try*.

5. W ramach bloku *Try* otwórz do odczytu plik tekstowy:
```
Dim fs As New System.IO.FileStream("c:\plik.txt", IO.FileMode.Open).
```

6. Po pierwsze, sprawdzimy, czy wykonanie tej instrukcji nie spowodowało wywołania błędu związanego z odwołaniem się do nieistniejącego pliku:
```
Catch ex As System.IO.FileNotFoundException
    MsgBox("Nie znalazłem określonego pliku")
```

7. W drugiej kolejności zweryfikujmy poprawność lokalizacji pliku:
```
Catch ex As System.IO.DirectoryNotFoundException
    MsgBox("Niepoprawna lokalizacja pliku")
```

8. Następnie sprawdzimy, czy dostęp do pliku nie został zablokowany:
```
Catch ex As System.UnauthorizedAccessException
    MsgBox("Odmowa dostępu")
```

9. I wreszcie sprawdzimy, czy plik nie został uszkodzony:
```
Catch ex As System.IO.FileLoadException
    MsgBox("Uszkodzony plik z danymi")
```

10. Ponieważ nieprzechwycony błąd spowoduje zatrzymanie programu, na końcu wyświetlimy komunikat pozostałych błędów:
```
Catch ex As Exception
     MsgBox(" Nieoczekiwany błąd:" + ex.Message)
```



Wyjątki są sprawdzane w określonej przez programistę kolejności, a po spełnieniu pierwszego warunku kolejne nie są już sprawdzane. ***Wynika z tego, że najogólniejszy warunek powinien być sprawdzany jako ostatni***.

11. Uruchom program i przetestuj jego działanie. Wywołaj kolejno wyjątek klasy *FileNotFoundException* — wystarczy wpisać w wierszu *Dim fs As New System.IO.FileStream("c:\plik.txt", IO.FileMode.Open)* nazwę nieistniejącego pliku; *DirectoryNotFoundException* — np. wpisując literę nieistniejącego napędu i *UnauthorizedAccessException* — np. podając poprawną nazwę i lokalizację pliku z ustawionym atrybutem *Tylko do odczytu*.

12. Zakończ działanie programu i zapisz wprowadzone w nim zmiany.

Ćwiczenie 9.9.

Wywołujemy własne wyjątki

Chociaż lista klas predefiniowanych wyjątków jest dość długa, pisząc zorientowane obiektowo programy będziemy chcieli wywołać własne, odpowiadające logice programu, błędy. W naszym przypadku próba wywołania funkcji *SilniaRekurencja* z ujemnym argumentem spowoduje błąd.

1. Wyświetl okno edytora kodu formularza *frmBledy*.

2. Umieść wywołanie funkcji *SilniaRekurencja* w utworzonym bloku *Try*.

3. W bloku *Catch* wyświetl użytkownikowi komunikat o błędzie.

4. Przed instrukcją *End Try* dodaj blok *Finally*. ***Instrukcje umieszczone w tym bloku wykonywane są zawsze, niezależnie od tego, czy wystąpi jakikolwiek błąd***.
5. W ramach bloku *Finally*:
   a. zadeklaruj zmienną statyczną *licznikWywolan* typu *Integer*,
   b. zwiększ wartość tej zmiennej o jeden,
   c. wyświetl bieżącą wartość zmiennej na ekranie.
6. Zmodyfikowane wywołanie funkcji powinno wyglądać następująco:

```
Try
            MsgBox("Silnia wynosi " + SilniaRekurencja(intLiczba).ToString)
Catch ex As Exception
            MsgBox(ex.Message)
Finally
            Static licznikWywolan As Integer
            licznikWywolan += 1
            MsgBox("Licznik wynosi: " + licznikWywolan.ToString)
End Try
```

7. Znajdź definicję funkcji *SilniaRekurencja*.
8. W pierwszym wierszu funkcji dodaj warunek sprawdzający, czy wartość przekazanego argumentu jest ujemna: `If liczba < 0 Then`.
9. ***Możemy wywołać wyjątek za pomocą instrukcji*** *Throw* podając typ wyjątku i ewentualnie komunikat o błędzie — wywołaj wyjątek klasy *ArgumentOutOfRangeException*:

```
If liczba < 0 Then
            Throw New ArgumentOutOfRangeException("Ujemny argument wywołania")
End If
```

10. Uruchom program i przetestuj jego działanie. Zwróć uwagę, że **błąd, który wystąpił w wywołanej funkcji, został przekazany do procedury wywołującej i tam dopiero został przechwycony**.
11. Zakończ działanie programu i zapisz wprowadzone w nim zmiany.