JavaScript. Zaawansowane programowanie

Autor: praca zbiorowa
Tłumaczenie: Maciej Gónicki, Adam Jarczyk, Daniel Kaczmarek, Aleksandra Tomaszewska, Tomasz Wrona
ISBN: 83-7197-687-9
Tytuł oryginału: Professional JavaScript 2nd Edition
Format: B5, stron: 1222

JavaScript jest językiem sieci. Jest to język intuicyjny i łatwy do przyswojenia. Jego wielką zaletą jest dostępność w większości nowoczesnych przeglądarek WWW. Dzięki JavaScriptowi możemy tworzyć dynamiczne, interaktywne strony WWW. Język ten najczęściej wykorzystywany jest do sprawdzania poprawności formularzy, tworzenia dynamicznych menu, a także do programowania gier. Jednak możliwości tego języka są znacznie większe. Trudno powiedzieć, czy istnieje chociaż jedna komercyjna witryna WWW, która nie zawiera w ogóle JavaScriptu.

Książka „JavaScript. Zaawansowane programowanie" zawiera wszechstronny opis języka JavaScript, jego składni i zastosowań. Na początek zaprezentujemy najnowsze przeglądarki i standardy stosowane w WWW, a następnie przedstawimy praktyczne techniki w postaci krótkich przykładów oraz bardziej szczegółowych i złożonych analiz konkretnych rozwiązań. Niniejsza książka koncentruje się wyłącznie na wykorzystaniu JavaScriptu w przeglądarkach WWW, ponieważ głównie w nich język ten jest stosowany.

Dla kogo jest ta książka?
Dla każdego, komu potrzebny jest JavaScript do tworzenia aplikacji WWW uruchamianych po stronie klienta. Czytelnikom znającym już język JavaScript, książka ta może posłużyć jako aktualny, zaawansowany poradnik; natomiast tym, którzy znają już inny język programowania, pozwoli opanować JavaScript jako nową umiejętność.

Co opisuje ta książka?

- Podstawowe programowanie w JavaScripcie
- Wykorzystanie w skryptach obiektów przeglądarek
- Pracę z multimediami
- Standardy WWW, w tym XML, CSS i W3C DOM
- Dynamiczny HTML
- Techniki uruchamiania skryptów
- Wyrażenia regularne i walidację formularzy
- Analizę przykładów zastosowań praktycznych
- Propozycja ECMAScript Edition 4

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

Spis treści

O Autorach 17

Wprowadzenie 21

Część I JavaScript w Sieci 27

Rozdział 1. JavaScript, przeglądarki i Sieć 29

JavaScript i Sieć 29
Techniczny kącik 30
Stan i historia przeglądarek 34
Obecna użyteczność Sieci 37
Ograniczenia Sieci i trendy 40
Standardy sieciowe i JavaScript 42
Typy standardów 43
Podstawowe standardy znaczników 43
Pomocnicze standardy 44
Narzucanie standardów 46
Programowanie w JavaScripcie 47
Nad czym będziesz spędzał czas 47
Z kim będziesz spędzał czas 48
Narzędzia programowania WWW 49
Podsumowanie 51

Rozdział 2. Rdzeń języka JavaScript 53

Pisanie skryptów a standard Unicode 53
Dodawanie własnego kodu JavaScript do strony internetowej 55
Instrukcje, bloki oraz komentarze 56
Zmienne i stałe 57
Wyrażenia i warunki 59
Operatory arytmetyczne 59
Operatory relacji 60
Operatory logiczne 61
Rozmaite operatory jednoargumentowe 62
Operatory przypisania 64
Inne operatory 65
Typy danych 66
Proste typy danych 66
Inne podstawowe typy danych 69

Sterowanie przepływem ... 70
if … else ... 70
while ... 72
do … while ... 73
for ... 73
break and continue ... 75
switch ... 76
Funkcje ... 77
Tworzenie własnych funkcji ... 77
Zasięg zmiennych funkcji ... 79
Funkcje w wyrażeniach ... 80
Funkcje zaimplementowane w JavaScripcie ... 80
Obiekty ... 82
Czym są obiekty ... 82
Wbudowane obiekty w JavaScripcie ... 84
Operatory obiektowe i sterowanie przepływem ... 90
Funkcje to również obiekty ... 92
Konwersje typów danych i porównywanie ... 93
Prosta i bezpieczna zmiana typu ... 94
Konwersja prostych typów ... 94
Operatory i mieszane typy danych ... 96
Podsumowanie ... 98

Rozdział 3. Techniki programowania obiektowego i JavaScript ... 101

Czym są obiekty? ... 101
Abstrakcja danych ... 102
Części składowe obiektu ... 102
Rozszerzenia ... 103
Kapsułkowanie ... 105
Dlaczego obiekty są takie ważne? ... 106
JavaScript ... 107
Obiekty JavaScript ... 107
Kontekst wykonania, kod funkcji i zasięg ... 109
Konstruktory obiektu ... 113
Ponowne użycie kodu ... 113
Techniki programowania obiektowego i JavaScript ... 115
Konstruktory obiektów, klasy oraz instancje ... 115
Dziedziczenie obiektów ... 122
Dodawanie chronionych elementów danych do języka JavaScript ... 132
Metody chronione ... 133
Kod źródłowy dla metody chronionej ... 136
Podsumowanie ... 140

Rozdział 4. Okna i ramki ... 143

Ramki ... 143
Jak zbudowane są zestawy ramek ... 144
Porządek ładowania ramek ... 157
Kod różnych ramek ... 160
Zdarzenia ramki ... 168
Okna ... 168
Łatwe okna ... 169
Tworzenie nowego okna przeglądarki ... 170

Kod różnych okien ... 176
Zamykanie okien ... 181
Podsumowanie ... 182

Rozdział 5. Formularze i dane ... 185

Formularze ... 185
Obiekty formularza ... 186
Elementy formularza ... 192
Otrzymywanie danych bez formularzy ... 200
Dane ... 204
Trwałość danych na stronie internetowej ... 204
Porównywanie archiwów ... 210
Sprawdzanie poprawności ... 210
Układ formularza ... 215
Podsumowanie ... 226

Rozdział 6. Multimedia oraz moduły rozszerzające ... 227

Krótka historia multimediów w Internecie ... 227
Dołączanie multimediów do strony internetowej ... 229
EMBED ... 229
NOEMBED ... 230
Netscape 6 ... 231
OBJECT ... 231
Alternatywne przeglądarki ... 233
Względy specjalne ... 233
Rozpoznawanie i kontrolowanie modułu przy użyciu JavaScriptu ... 235
Rozpoznawanie komponentów za pomocą IE 5.0+ ... 235
Tworzenie instancji dla rozpoznania obiektów ActiveX ... 237
Problemy rozpoznawania — IE na Macintoshu ... 238
Użycie obiektów przeglądarki — obiekt navigator ... 239
Danie główne — skrypt rozpoznający moduły rozszerzające ... 241
Kontrolowanie popularnych formatów medialnych przy użyciu języka JavaScript ... 247
Macromedia ... 247
RealNetworks ... 249
Microsoft Windows Media Player ... 252
Apple QuickTime ... 253
JavaScript oraz Java ... 257
Porównanie i kontrast ... 257
Początek ... 257
Element APPLET ... 258
Pisanie prostych apletów Javy ... 258
Integracja apletów z JavaScriptem ... 260
Model bezpieczeństwa w Javie ... 262
Nowe technologie ... 263
SMIL ... 263
HTML+TIME ... 263
Skalowalna grafika wektorowa — SVG ... 267
Język interfejsu użytkownika oparty na XML (XUL) ... 270
Podsumowanie ... 271

Część II Ku standaryzacji 273

Rozdział 7. XML i XHTML 275

Wprowadzenie do języka XML 275
Historia języka XML 276
SGML 276
HTML i sieć WWW 276
Dane i XML 277
Dobrze sformułowany XML 277
Znaczniki, atrybuty i elementy 278
Deklaracja dokumentu XML 280
Instrukcje przetwarzania 281
Komentarze, obiekty i inne 281
Przestrzeń nazw 282
Przykład tekstu XML 284
Obowiązujący XML 284
XML DTD 285
W3C XML Schema 287
Inne technologie tworzenia schematów 290
Parsery i walidacja 290
Wyświetlanie XML 291
Wyświetlanie XML za pomocą CSS 291
Wyświetlanie XML za pomocą XSLT 292
XML i JavaScript 297
XML DOM 297
XHTML 300
XML i nowe specyfikacje 303
SVG — Scalable Vector Graphics 303
SMIL 304
RDF — Resource Description Framework 304
MathML 305
Podsumowanie 306

Rozdział 8. Kaskadowe arkusze stylów i JavaScript 307

Cele rozwoju CSS 308
CSS1 i CSS 2 310
Przyszłość — CSS 3 312
Korzystanie z CSS 313
Aktualny stan dostępności CSS 314
Składnia CSS 315
Jednostki miar 319
Modularyzacja CSS 321
Model obiektu CSS 322
Dostęp do obiektu stylu elementu 324
Modyfikowanie obiektu stylu elementu 326
Dokumenty i kolekcje arkuszy stylów 328
Obiekty StyleSheet 330
Reguły stylów 331
Zasięg sterowania stylami CSS 333
Pseudoklasy i elementy 337
Umieszczanie treści przed i za 340

Sterowanie czcionką ... 341
Sterowanie obszarem i ułożeniem tekstu ... 344
Rozmiar i położenie obiektu ... 346
Model kolorów i tła ... 348
Model obszaru — brzegi, marginesy i odstęp ... 349
Układ listy ... 353
Układ tabeli ... 355
Efekty wizualne ... 357
Automatycznie generowana zawartość ... 359
Typy nośników (stronicowane i przewijane) ... 359
Sterowanie interfejsem użytkownika ... 360
Style dźwiękowe ... 360
Proponowane przestrzenie nazw CSS poziom 3 ... 362
Inne zastosowania praktyczne ... 362
Przenośne rozmiary czcionek ... 362
Zmiana wyglądu obiektów wprowadzania danych ... 363
Zmiana stylu sterowana zdarzeniami ... 364
Sterowanie położeniem przez zdarzenia myszy ... 366
Przenośność a przeglądarki starego typu ... 367
Podsumowanie ... 369

Rozdział 9. DOM ... 371

Teoria i praktyka ... 372
DOM poziomy 0, 1, 2 i 3 ... 374
Struktura drzewa DOM ... 375
Obiekty węzłów ... 377
Relacje rodzic-potomek ... 380
Węzły Document ... 381
Węzły Element ... 382
Węzły Attribute ... 383
Węzły CharacterData ... 383
Węzły Text ... 384
Dostęp do arkusza stylów CSS ... 386
Aktorzy drugoplanowi ... 386
Analiza implementacji w przeglądarkach ... 387
Narzędzia analizy ... 387
Inne przydatne techniki inspekcji ... 394
Implementacje DOM w przeglądarkach IE i Netscape ... 395
Tłumaczenie reprezentacji tabeli według modelu DOM ... 406
Modyfikowanie drzewa ... 407
Punkt wyjścia ... 407
Rozrastanie i przycinanie ... 410
Zabawa z atrybutami ... 420
Obiekty HTMLElements ... 424
Co dzieje się z symbolami w węzłach Text? ... 426
Modularyzacja DOM ... 427
Moduły DOM ... 428
Strategia modularyzacji ... 429
Detekcja dostępnych cech ... 430
Modele zdarzeń DOM ... 433
Łączenie faz przechwytywania i bąbelkowania ... 433
Obiekty zdarzeń ... 434

Typy zdarzeń ... 435
Dołączanie obsługi zdarzeń — styl klasyczny ... 437
Przechwytywanie zdarzeń ... 438
Samodzielne przydzielanie obsługi zdarzeń ... 441
Obsługa zdarzeń złożonych ... 442
Obsługa przestrzeni nazw XML ... 443
Funkcje jeszcze niedostępne ... 444
DOM poziom 2 ... 444
DOM poziom 3 ... 444
Podsumowanie ... 445

Rozdział 10. Dynamiczny HTML ... 447

Przegląd historyczny ... 448
Technologie DHTML ... 449
document.write() ... 450
Techniki zamiany rysunków ... 455
InnerHTML i Friends ... 459
Pozycjonowanie obiektów ... 462
Zmiana porządku wewnątrz dokumentu ... 467
Efekty transformacji ... 468
Przykład selektora kolorów ... 470
Dynamiczny HTML i nowe modele zdarzeń ... 473
Model przechwytywania zdarzenia ... 473
Model bąbelkowania zdarzeń ... 475
Model zdarzeń DOM ... 476
Implementacja prostych funkcji obsługi zdarzenia ... 478
Funkcje nasłuchiwania zdarzeń ... 481
Selektor kolorów Netscape Navigator 6 ... 482
Podsumowanie ... 486

Część III Strategie programowania 487

Rozdział 11. Wybieranie narzędzi ... 489

Problemy produkcyjne ze skryptami ... 490
Tworzymy linię produkcyjną ... 490
Wybieramy narzędzia programistyczne ... 494
Narzędzia przyspieszające pisanie skryptów ... 494
Narzędzia do zarządzania bazą kodu ... 506
Próbny cykl produkcyjny ... 511
Ponowne wykorzystanie istniejącego kodu ... 513
Biblioteka Netscape do sprawdzania poprawności ... 513
Biblioteki cookie ... 515
Podsumowanie ... 516

Rozdział 12. Praktyka dobrego kodowania ... 519

Dlaczego potrzebne są standardy kodowania ... 519
Dobre praktyki ogólne ... 520
Prawidłowy układ kodu ... 520
Stosuj opisowe nazwy ... 521
Stosuj konwencje nazewnictwa ... 522
Stosuj zmienne dla wartości specjalnych ... 524
Komentarze ... 525

Czystość i prosta ... 526
Upraszczaj wyrażenia warunkowe ... 527
Stosowanie tablic w roli tabel wyszukiwania ... 527
Ogranicz do minimum liczbę punktów wyjścia z pętli ... 531
Pokrewny kod trzymaj razem ... 533
Zorganizuj swój kod ... 533
Szczegóły ... 537
Zmienne ... 537
Funkcje ... 539
Usuwamy skutki katastrofalnego kodowania ... 542
Podsumowanie ... 545

Rozdział 13. Obsługa błędów, uruchamianie, rozwiązywanie problemów ... 547

Typy błędów w JavaScripcie ... 547
Błędy czasu ładowania ... 548
Błędy czasu wykonania ... 549
Błędy logiczne ... 551
Najczęstsze błędy w JavaScripcie ... 552
Niezdefiniowane zmienne, pisownia i kolejność skryptu ... 552
Niedomknięte nawiasy okrągłe i klamrowe ... 553
Użycie metody nieobsługiwanej przez obiekt ... 554
Użycie słowa zastrzeżonego ... 555
Cudzysłowy ... 555
Brakujące znaki „+” przy konkatenacji ... 557
Metody jako własności i odwrotnie ... 558
Niejasne instrukcje else ... 558
Problemy z for … in ... 559
Zadzieranie z typami ... 560
Obcinanie łańcuchów ... 561
Jedyność nazw funkcji ... 561
Użycie wyjątków ... 562
Składnia wyjątków a „if” ... 565
Wyjątki, zdarzenia i błędy ... 567
Jak pisać kod czysty i bezbłędny ... 568
Kod modularny ... 568
Sprawdzanie istnienia obiektów ... 568
Techniki uruchamiania ... 568
JavaScript po stronie klienta ... 569
Microsoft Script Debugger ... 570
Jak zdobyć Microsoft Script Debugger ... 571
Jak włączyć Microsoft Script Debugger ... 571
Jak używać programu Microsoft Script Debugger ... 572
Netscape JavaScript Console ... 584
Podsumowanie ... 587

Część IV Rady, triki i techniki 589

Rozdział 14. Prywatność, bezpieczeństwo i cookies ... 591

Prywatność dla twórcy skryptów ... 591
Ukrywanie kodu źródłowego ... 592
Zniechęcanie do podglądania ... 596

Bezpieczeństwo ... 597
Bezpieczeństwo dla użytkowników przeglądarek ... 597
Cookies ... 605
Teoria cookies ... 606
JavaScript i cookies ... 609
Użycie cookies ... 610
Pułapki cookie ... 613
Zestaw narzędzi dla cookies ... 613
Podsumowanie ... 626

Rozdział 15. Wyrażenia regularne ... 629

Wyrażenia regularne w JavaScripcie ... 630
Tworzenie wyrażeń regularnych przez obiekt RegExp ... 630
Modyfikatory wyrażeń regularnych ... 632
Użycie obiektu RegExp() metodą test() ... 633
Użycie metody replace() obiektu String ... 634
Składnia wyrażeń regularnych ... 634
Proste wyrażenia regularne ... 634
Znaki specjalne ... 636
Użycie wyrażeń regularnych w JavaScripcie ... 653
Globalny obiekt RegExp ... 653
Obiekt RegExp ... 657
Obsługa wyrażeń regularnych w obiekcie String ... 661
Podsumowanie ... 665

Rozdział 16. Zatwierdzanie formularza ... 667

Klasa Validate ... 668
Zatwierdzanie informacji ... 671
Zatwierdzanie informacji podstawowych ... 671
Zatwierdzanie wieku ... 675
Zatwierdzanie formatów hasła ... 676
Zatwierdzanie numerów telefonów ... 678
Zatwierdzanie kodów pocztowych ... 680
Zatwierdzanie adresu poczty elektronicznej ... 682
Weryfikacja dat ... 683
Zatwierdzanie numerów kart kredytowych ... 687
Rozszerzanie klasy Validate ... 697
Jak to działa ... 698
Tworzenie rozszerzeń ... 701
Zastosowanie klasy Validate ... 708
Zatwierdzanie bez JavaScriptu ... 714
Podsumowanie ... 714

Rozdział 17. Dynamiczne strony WWW ... 715

Proste efekty dynamiczne ... 716
Dostęp do zawartości ... 722
Magiczne sztuczki ... 732
Przełączanie reklam ... 733
Przewijanie transparentów ... 736
Wyświetlanie dynamiczne ... 740

Elementy nawigacji ... 747
Etykietki narzędzi ... 748
Niestandardowe etykietki narzędzi ... 749
Panele z zakładkami ... 752
Kiedy łącze nie jest łączem? ... 755
Widgety funkcjonalne ... 756
Sortowanie danych w tabeli ... 757
Która godzina? ... 761
Przeciągnij i upuść ... 765
Ruch ... 768
Pong ... 771
Podsumowanie ... 775

Rozdział 18. Filtry Internet Explorera ... 777

Czym są filtry? ... 778
Multimedialne efekty wizualne ... 779
Definiowanie prostego filtra ... 780
Filtry i układ ... 780
Filtry i model obiektowy ... 783
Kolekcja Element.filters ... 785
Łańcuch style.filter ... 786
Dostęp do własności obiektu filtra ... 787
Efekty proceduralne ... 788
Statyczne efekty filtrów ... 791
Efekty przejściowe ... 795
Przejścia pomiędzy stronami ... 797
Zdarzenia i filtry ... 799
Optymalizacja filtrów statycznych ... 801
Optymalizacja zmian asynchronicznych ... 803
Optymalizacja filtrów przejściowych ... 805
Filtry i różne przeglądarki ... 807
Filtry Internet Explorera 4.0 ... 808
Wykrywanie wersji przeglądarki ... 809
Podsumowanie ... 810

Rozdział 19. Rozszerzanie obiektów standardowych i obiektów przeglądarek ... 811

Krótka powtórka ... 812
Prosty przykład rozszerzenia ... 812
Zastosowanie własności prototypu ... 813
Zalety ... 814
Obiekty standardowe ... 816
Przykład wykorzystujący wbudowany obiekt JavaScriptu ... 816
Array ... 817
Date ... 822
Function ... 831
Math ... 832
Number ... 833
Object ... 836
String ... 838
Obiekty przeglądarek ... 841
Window ... 842
Globalne metody obiektów ... 843
Formularze ... 845

Zastosowanie w Dynamicznym HTML-u ... 846
Zgodność z DOM W3C ... 846
Zgodność ze starszymi skryptami ... 849
Podsumowanie ... 851
Zasoby internetowe ... 852
Zasoby drukowane ... 852

Część V Studium konkretnych przypadków 853

Rozdział 20. Konsola audiowizualna BBC News ... 855

Dlaczego tak, a nie inaczej? ... 856
Ustalanie wymagań ... 857
Dostęp do kodu źródłowego ... 859
Zestaw materiałów źródłowych ... 859
Budowa fundamentów ... 861
Decyzje odnośnie implementacji ... 863
Kliknij i odtwórz ... 864
Ramki głównego zestawu ... 865
Procedura obsługi odtwarzania klipów ... 870
Ramka odtwarzacza wideo ... 874
Ramka odtwarzacza wideo (IE 4 Win) ... 876
Ramka odtwarzacza wideo (IE 4 Mac) ... 878
Ramka odtwarzacza wideo (Nav4 Win) ... 878
Ramka odtwarzacza wideo (Nav4 Mac) ... 880
Ramka odtwarzacza wideo (inne plug-iny) ... 880
Ramka wiadomości ... 880
Ramka bannera ... 881
Ramka dla panela menu ... 882
Ramka przewijających się wiadomości „na żywo” ... 884
Biblioteka funkcji ... 889
Dane zdarzeń sterowane przez wideo ... 890
Zmiany w Navigatorze 6 ... 891
Podsumowanie ... 896

Rozdział 21. Teleks BBC News ... 897

Dlaczego w JavaScripcie? ... 898
Zagadnienia systemowe ... 899
Wstawienie teleksu na stronę ... 900
Zasada działania ... 901
Sprawdzenie przeglądarki ... 901
Uruchomienie teleksu ... 901
Pętla główna ... 903
Importowanie danych ... 904
Efekty wizualne i prezentacja ... 905
Nadawanie wyglądu końcowego ... 906
Problemy z odświeżaniem ekranu ... 907
Wyciekanie pamięci i odśmiecanie ... 908
Specyfikacja uaktualnionego teleksu ... 909
Przekazywanie danych do teleksu ... 909
Ukryte pola formularza ... 910
Ukryte ramki i warstwy ... 911

Nawigacja poprzez DOM ... 911
Zhierarchizowane bloki DIV ... 912
Alternatywa wykorzystująca wyspy danych XML ... 913
Dołączany kod JavaScript ... 915
Implementacja nowego projektu teleksu ... 916
Podsumowanie ... 921

Rozdział 22. Koszyk na zakupy ... 923

Obraz ogólny ... 923
Cele ... 924
Proces zakupów ... 925
Spojrzenie od strony technicznej ... 928
Tworzenie programu koszyka na zakupy ... 930
Tworzymy podstawowe strony WWW ... 930
Tworzymy koszyk na zakupy ... 932
Tworzymy kasę ... 956
Podsumowanie ... 976

Rozdział 23. Drzewo genealogiczne ... 979

Aplikacja albumu z fotografiami ... 980
Wymogi projektu ... 981
Strategia implementacji ... 981
Projektujemy aplikację ... 982
Drzewo genealogiczne ... 984
Aplikacja DHTML dla Internet Explorera ... 985
Główna ramka aplikacji ... 985
Implementacja strony wyświetlającej fotografie ... 985
Implementacja modalnego okna dialogowego ... 988
Sterownik struktury drzewiastej w czystym JavaScripcie ... 992
Testowanie albumu z fotografiami ... 1005
Wyposażanie albumu w obsługę XML-a ... 1005
Wersja zgodna z przeglądarkami 6, 5, 4.x ... 1009
Wykorzystanie biblioteki Netscape z listami rozwijanymi ... 1009
Symulacja modalnego okna dialogowego ... 1014
Nowa strona wyświetlająca fotografie ... 1015
Podsumowanie ... 1022

Część VI Bieżące kierunki rozwoju JavaScriptu 1023

Rozdział 24. ECMAScript 4 ... 1025

ECMAScript == JavaScript? ... 1025
Początki JavaScriptu a ECMAScript ... 1026
Dlaczego potrzebny jest ECMAScript ... 1028
Ewolucja JavaScriptu i ECMAScriptu w edycji 3. ... 1028
JavaScript 2.0 i co dalej? ... 1029
Kompilacja a interpretacja ... 1030
Klasa a prototyp ... 1031
ECMAScript edycji 4. i co dalej? ... 1032
Utrzymanie starego sposobu funkcjonowania ... 1033
Dodanie nowego sposobu funkcjonowania ... 1033

JavaScript 2.0 > ECMAScript edycji 4. ... 1047
Zgodność wsteczna ... 1048
ECMAScript jest dobrą rzeczą ... 1049
Podsumowanie ... 1050
Odsyłacze WWW ... 1051

Rozdział 25. .NET, usługi web, JScript.NET i JavaScript ... 1053

Microsoft i .NET ... 1053
Czym jest .NET? ... 1054
Więcej o JScript.NET ... 1056
JScript.NET i przeglądarki WWW ... 1056
Visual Studio.NET ... 1057
ASP.NET ... 1058
Wprowadzenie do usług web dla nowicjusza ... 1060
Prawdziwa niezależność ... 1062
Internet Explorer jako konsument usług web ... 1063
Tworzymy naszą pierwszą usługę ... 1064
Wykorzystanie zachowania usługi WWW ... 1066
Metody useService() i callService() ... 1067
Obiekt wynikowy ... 1068
Składanie wszystkiego w całość ... 1069
Usługa sieciowa sklepu elektronicznego ... 1070
Metody ... 1071
Proces ... 1072
Tworzenie formularza zamówienia ... 1072
error i errorDetail ... 1073
Przetwarzanie formularza zamówienia ... 1074
Krok w przód ... 1077
Podsumowanie ... 1078

Dodatki 1079

Dodatek A Funkcje, instrukcje i operatory ... 1081

Ograniczniki ... 1081
Operatory ... 1082
Pierwszeństwo operatorów ... 1087
Konstrukcje językowe ... 1089
Funkcje wbudowane JavaScript ... 1093
Słowa zarezerwowane ... 1094

Dodatek B Obiekty, metody i własności ... 1095

Object ... 1096
Array ... 1097
Boolean ... 1098
Date ... 1099
Error ... 1102
Function ... 1104
Math ... 1105
Number ... 1107
RegExp ... 1108
String ... 1109
Window ... 1112

Obiekty specyficzne dla implementacji: Microsoft ... 1117
ActiveXObject ... 1117
Enumerator ... 1117
VBArray ... 1118
Obiekty specyficzne dla implementacji: Netscape ... 1119
JavaArray ... 1119
JavaClass ... 1120
JavaObject ... 1120
JavaPackage ... 1121

Dodatek C Typy danych i konwersje typów ... 1123

Proste typy danych ... 1123
Wbudowane stałe ... 1123
Konwersja typów ... 1124
Operator typeof ... 1126

Dodatek D Obsługa zdarzeń ... 1129

Kompendium typów zdarzeń ... 1129
Typy zdarzeń modelu DOM ... 1134
Obiekt zdarzenia ... 1135
Rozszerzenia DOM dla obiektu celu zdarzenia ... 1147
Rozszerzenia DOM obiektu Document ... 1148
Dodatkowe zdarzenia obsługi klawiszy w modelu DOM poziom 3 ... 1148

Dodatek E Słownik CSS ... 1153

Obiekty ... 1153
Obiekt style ... 1153
Obiekt CSSStyleSheet ... 1165
Obiekt CSSStyleRule ... 1166
Obiekt CSSMediaRule ... 1167
Obiekt CSSFontFaceRule ... 1168
Obiekt CSSPageRule ... 1168
Obiekt CSSImportRule ... 1168
Obiekt CSSUnknownRule ... 1169
Jednostki miar ... 1169

Dodatek F Model DOM ... 1171

Typy węzłów modelu DOM ... 1171
Węzeł Document ... 1174
Węzeł Element ... 1176
Węzeł Attribute ... 1177
Węzeł Character Data ... 1178
Węzły Text ... 1179
Węzły Comment ... 1179
Węzły CDATA Section ... 1179

Skorowidz ... 1181

13

Obsługa błędów, uruchamianie, rozwiązywanie problemów

Ponieważ JavaScript jest językiem interpretowanym, informacja zwrotne ogranicza się głównie do ostrzeżeń o problemach ze składnią naszych skryptów. Gdy się z tym uporamy, nieliczne komunikaty diagnostyczne czasu wykonania przekładają się w zasadzie na „Próbowałem to zrobić, ale coś, czego potrzebowałem, nie istnieje".

JavaScript w założeniach ma być językiem szybko i łatwo interpretowanym i interaktywnym, często wymagającym zaledwie kilku instrukcji. Żaden kompilator nie sprawdza rygorystycznie skryptów zanim pozwoli ich użyć. Na skutek tego bardzo łatwo nabrać przyzwyczajeń do niedbałego programowania. Wiele problemów, takich jak błąd pominięcia testu obiektu przed użyciem, brak obsługi wyjątków lub zapominanie o zamykaniu nawiasów z uwagi na niewłaściwe wcięcia, bierze się z braku dobrych nawyków programistycznych.

W niniejszym rozdziale omówimy różne typy błędów JavaScriptu, ich przyczyny oraz sposoby ich odnajdywania i poprawiania. Na koniec omówimy dwa najważniejsze programy uruchomieniowe: Microsoft Script Debugger i Netscape JavaScript Console.

Typy błędów w JavaScripcie

Instrukcje JavaScriptu są interpretowane po załadowaniu przez interpreter programu gospodarza (na przykład przeglądarki WWW). Podczas ładowania skryptu przeprowadzana jest kontrola składni. Jeśli znalezione zostaną błędy składniowe, to wykonywanie skryptu jest przerywane i zostaje zgłoszony błąd. Po zakończeniu kontroli składni wykonywane są wszystkie polecenia globalne (nie zawarte w funkcjach), takie jak deklaracje zmiennych. Na tym etapie może wystąpić **błąd czasu wykonania**, spowodowany przez cokolwiek, począwszy od niezdefiniowanej zmiennej, aż po przekroczenie zakresu przez indeks tablicy. Błędy wykonania powodują przerwanie wykonywania funkcji, lecz pozwalają na dalsze działanie innych funkcji obecnych w skrypcie. Funkcje i procedury są rozpoznawane przy

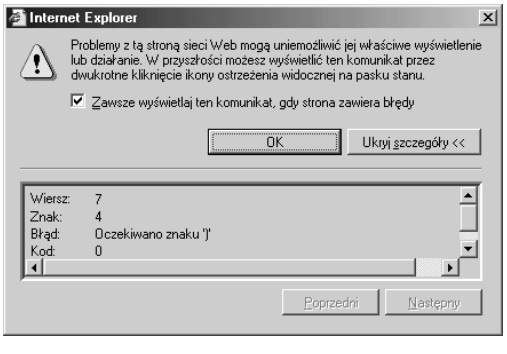
ładowaniu skryptu, lecz nie będą wykonane, dopóki nie zostaną wywołane przez inne funkcje lub zdarzenia. Podczas wykonywania funkcji mogą zostać wygenerowane błędy wykonania lub **błędy logiczne** (tzn. błędy, które powodują, że wynik nie jest zgodny z oczekiwaniami).

Błędy czasu ładowania

JavaScript przechwytuje i zgłasza **błędy czasu ładowania** (inaczej **błędy składni**) podczas ładowania skryptu. Błędy składni są powodowane przez poważne pomyłki w składni skryptu. Skrypty, które zawierają błędy składni generują komunikaty o błędach podczas ładowania skryptu i nie zostają uruchomione. Tego typu błędy najłatwiej chyba wychwycić i naprawić, ponieważ pojawiają się przy każdym ładowaniu skryptu, w przeciwieństwie do błędów czasu wykonania i błędów logicznych, które generowane są tylko po wywołaniu funkcji lub spełnieniu określonych warunków. Brakujące cudzysłowy, nawiasy okrągłe i klamrowe należą do najczęstszych przyczyn tych błędów. Weźmy na przykład poniższy fragment kodu:

```
<!-- loadtime.html -->
<HTML>
<HEAD>
   <TITLE>Błędy podczas ładowania</TITLE>
   <SCRIPT LANGUAGE='JavaScript'>
      alert('Witaj, świecie!'
   </SCRIPT>
</HEAD>
</HTML>
```

Po załadowaniu powyższego kodu do IE 6 (gdy *debuger* jest wyłączony a powiadamianie o błędach jest włączone) generowany jest następujący komunikat o błędzie:

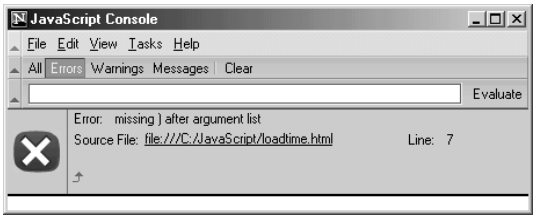


Aby wyłączyć odpluskwianie skryptów i włączyć powiadamianie w błędach skryptów w IE:

1. Otwórz okno przeglądarki Internet Explorer.
2. Kliknij *Narzędzia*.
3. Kliknij *Opcje internetowe*.
4. Kliknij zakładkę *Zaawansowane*.
5. Pod nagłówkiem Przeglądanie zaznacz opcję *Wyłącz debugowanie skryptu*.

6. Zaznacz opcję *Wyświetl powiadomienie o każdym błędzie skryptu*.
7. Kliknij *OK*.

Netscape 6 generuje podobny błąd na konsoli JavaScriptu (*Tasks/Tools/JavaScript Console*).



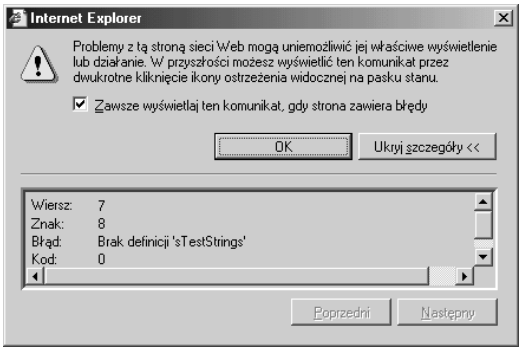
Jak widzimy, obie przeglądarki wyświetlają podobne informacje o błędzie. Każda wyświetla opis napotkanych błędów, plik, który spowodował błąd (plik źródłowy albo URL) oraz wiersz i położenie błędu w tym wierszu. Należy zwrócić uwagę, że wiersz i kolumna błędu (pozycja znaku) nie muszą przedstawiać dokładnego położenia błędu. Do takiego sposobu podawania numerów wierszy i położenia znaku trochę trudno się przyzwyczaić. Powodowane jest to faktem, że interpreter JavaScriptu wyświetla numer wiersza i kolumny, skąd nie potrafi przejść dalej. W powyższym przykładzie z IE interpreter JavaScriptu zatrzymał się na 7. wierszu, 4. znaku — dokładnie mówiąc, na < w wierszu `</SCRIPT>`. W Netscape 6 numery wartości mogą być inne z uwagi na różnice w interpreterach JavaScriptu i miejscach, od których nie potrafią interpretować skryptu. Z tego powodu warto korzystać ze środowiska IDE (np. Microsoft Visual .NET, Allaire HomeSite, Macromedia Ultradev itp.), w którym wyświetlane są numery wierszy.

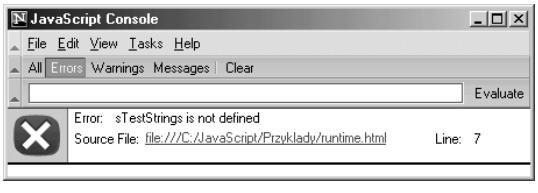
Błędy czasu wykonania

Błędy czasu wykonania są wychwytywane po załadowaniu skryptu i podczas działania. Błąd taki zachodzi, gdy skrypt usiłuje wykonać nieprawidłowe działanie. Gdy w skrypcie pojawi się błąd czasu wykonania, zostaje wyświetlony komunikat o błędzie i wykonanie skryptu ustaje. Inne skrypty nadal mogą działać, lecz za każdym razem, gdy zostaje wywołany skrypt z błędem, wyświetlany jest komunikat o błędzie. Do powszechnie spotykanych przyczyn występowania błędów czasu wykonania należą odwołania do niezdefiniowanych zmiennych, niewłaściwe zastosowanie obiektów, wstawianie niezgodnych typów i tworzenie pętli nieskończonych. Poniższy kod generuje typowy błąd czasu wykonania:

```
<!-- runtime.html -->
<HTML>
<HEAD>
   <TITLE>Przykład 2 - błąd wykonania</TITLE>
   <SCRIPT LANGUAGE='JavaScript'>
       var sTestString = "Witaj, świecie!";
       alert(sTestStrings); //uwaga na literę s na końcu nazwy sTestStrings
   </SCRIPT>
</HEAD>
</HTML>
```

Komunikaty o błędach generowane przez IE 5 i Netscape 6 są następujące:



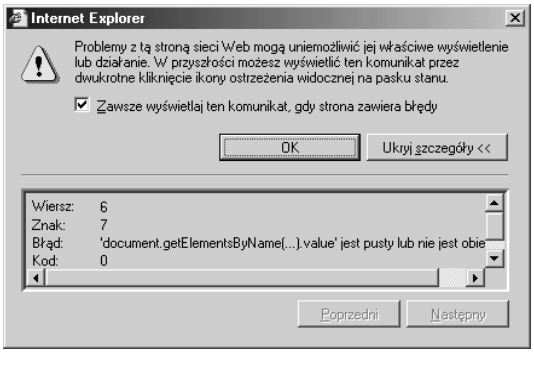


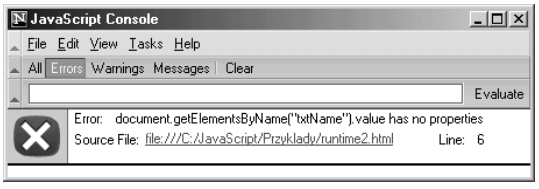
Obie przeglądarki informują użytkownika, że zmienna `sTestStrings` nie jest zdefiniowana. Ponieważ przeglądarka nie „wie", co ma wyświetlić w polu `alert()`, zwracany jest błąd.

Oto bardziej złożony przykład kodu generującego błąd wykonania:

```
<!-- runtime2.html -->
<HTML>
<HEAD>
   <TITLE>Przykład 3 - bardziej złożony błąd wykonania</TITLE>
   <SCRIPT LANGUAGE='JavaScript'>
      document.getElementsByName("txtName").value.length = "XX"
   </SCRIPT>
</HEAD>
<BODY>
   <INPUT TYPE='text' name='txtName' value='Fred'></INPUT>
</BODY>
</HTML>
```

Ten skrypt generuje następujące błędy:





Powodem, dla którego skrypt generuje błąd, jest jego wykonanie zanim zostanie utworzone pole tekstowe. Skrypt nie może przypisać wartości czemuś, co nie istnieje. Sam skrypt jest całkowicie poprawny i nie wygenerowałby żadnych błędów, gdyby był wykonany po utworzeniu pola tekstowego lub gdyby został umieszczony w funkcji wywołanej po załadowaniu tekstu HTML.

Błędy logiczne

Błędy logiczne są wolne od błędów składni i czasu wykonania, lecz prowadzą do niepoprawnych wyników. Błędy logiczne nie powodują zatrzymania wykonywania skryptu, chyba że niezamierzone wyniki błędu logicznego w połączeniu z innym poleceniem lub skryptem powodują błąd czasu wykonania. Usuwanie błędów logicznych jest często najtrudniejsze i może od programisty wymagać prześledzenia wartości wszystkich zmiennych w każdym kroku skryptu. Do częstych przyczyn błędów logicznych należą: użycie = zamiast == i niezgodne typy danych. Poniższy kod ilustruje, jak łatwo jest wygenerować błąd logiczny:

```
<!-- logical.html -->
<HTML>
<HEAD>
   <SCRIPT LANGUAGE='JavaScript'>
     var value;
      for (counter=1; counter<6; counter++)
      {
         value=counter;   //Aby zatrzymać nieskończoną pętlę ...
         if (value=3)   //Zmień "=" na "==" aby poprawić błąd
         {
            alert("Doszliśmy do połowy\nwartość = 3");
         }
         else
         {
            alert("value = " + value);
         }
      }
   </SCRIPT>
</HEAD>
</HTML>
```

W powyższym przykładzie wynik zawsze będzie taki:

Pierwsze wartościowanie zmiennej (instrukcja `if`) zawsze zwróci `true`, ponieważ za każdym razem ustawia wartość zmiennej równą `3` za pomocą operatora = zamiast użyć operatora porównania ==. Ponieważ instrukcja `if` daje `true` (gdyż mogłaby zwrócić `false` tylko wtedy, gdyby wartości `value` nie można było z jakiegoś powodu zmienić), za każdym razem otrzymamy alert jak na rysunku powyżej. Gdybyśmy zamienili = na == lub ===, wyniki byłyby zgodne z oczekiwaniami, zaś wartość w oknie *alert* reprezentowałaby spodziewaną wartość zmiennej, czyli zaczynała się od wartości `1` i zwiększała za każdym kliknięciem *OK* o 1, aż do `5`.

Najczęstsze błędy w JavaScripcie

Czytelnik, który spędził jakiś czas pracując JavaScriptem zapewne napotkał wiele z poniższych problemów. Zawsze jednak warto przyjrzeć się im po raz drugi, aby pamiętać o nich podczas kodowania.

Niezdefiniowane zmienne, pisownia i kolejność skryptu

Zmienna w JavaScripcie może być zdefiniowana na dwa sposoby: **niejawnie** i **jawnie**. Definicja niejawna pozwala zdefiniować zmienną po prostu przez przypisanie wartości:

```
iNumber = 3;
```

Definicja jawna wykorzystuje słowo kluczowe `var`:

```
var iNumber = 3;
```

lub:

```
var iNumber;//jawna definicja zmiennej
iNumber = 3;//inicjowanie zmiennej
```

Wszystkie trzy metody dają taki sam efekt (z małą różnicą dotyczącą zasięgu zmiennej: zmienne deklarowane niejawnie wewnątrz funkcji stają się zmiennymi globalnymi): zmienna o nazwie `iNumber` zostaje zdefiniowana i otrzymuje wartość `3`.

Jak widzimy, zdefiniować zmienną jest łatwo — wystarczy przypisać do niej wartość. Błąd otrzymamy tylko wtedy, gdy zmienna nie będzie w ogóle zdefiniowana.

Błędy niezdefiniowanych zmiennych są często powodowane przez literówki i błędnie stosowane duże litery w nazwach zmiennych.

```
// typo.html
var goToTown = "pójść do miasta";
var staysHome = "zostać w domu";

if (confirm("Kliknij OK, jeśli chcesz pójść do miasta"))
{
   alert("Chcę " + GoToTown);
}
else
```

```
{
   alert("Chcę " + stayHome);
}
```

W powyższym skrypcie są dwa poważne błędy. Po pierwsze, zmienna `goToTown` została niepoprawnie zapisana od dużej litery — `GoToTown`. Po drugie, zmienna `staysHome` została błędnie zapisana jako `stayHome`. Obie pomyłki powodują podobne błędy czasu wykonania: błąd braku definicji zmiennej.

Powyższe okno błędu jest wyświetlane, jeżeli Microsoft Script Debuger został zainstalowany i włączony. Więcej informacji o tym programie przedstawimy w podrozdziale „Microsoft Script Debuger" pod koniec rozdziału.

Chociaż definiowanie zmiennych w JavaScripcie jest bardzo łatwe, niezdefiniowane zmienne są bardzo częstym powodem błędów wykonania. Błędy zwykle powoduje nie to, *jak* zmienna została zdefiniowana, lecz *kiedy*. Weźmy na przykład poniższy kod:

```
function doAlert()
{
   alert(sWrox);
}
doAlert();
var sWrox = "Wrox";
```

W tym przykładzie, chociaż nie generuje błędu czasu wykonania jak poprzedni, bardzo łatwo odkryć, gdzie leży problem. Zmienna `sWrox` jest definiowana po wywołaniu funkcji `doAlert()`, wobec czego jest zwracana jako „niezdefiniowana". W rzeczywistych przykładach pomyłki tego typu mogą być trudne do znalezienia, zwłaszcza że określone skrypty są często dołączane jako pliki *.js*, więc faktyczny kod czasami jest niewidoczny na stronie.

Niedomknięte nawiasy okrągłe i klamrowe

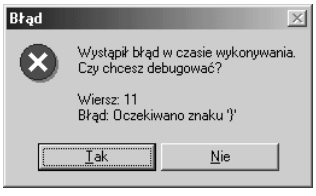
Kolejnym frustrującym problemem jest zapominanie o domknięciu nawiasów lub dopisywanie nawiasu tam, gdzie nie jest potrzebny. Pomyłki tego typu generują błąd składni podczas ładowania.

Jedną z najczęstszych pomyłek jest brak nawiasu klamrowego. Weźmy na przykład poniższy kod:

```
function returnBit(bValue)
{
   var bReturnValue = false;
   if (bValue)
```

```
    {
       bReturnValue = true;
    return bReturnValue;
}
```

Instrukcja `if` nie zawiera zamykającego nawiasu klamrowego i wywołuje następujący błąd:

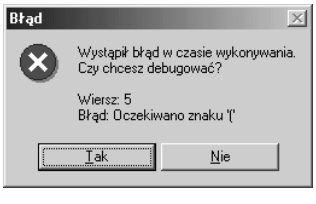


Właściwe stosowanie wcięć, wspomniane w poprzednim rozdziale, znacznie zmniejsza prawdopodobieństwo popełnienia tej pomyłki.

Kolejną częstą pomyłką jest zapominanie o wzięciu w nawias argumentu instrukcji `if`. Oto przykład:

```
function returnBit(bValue)
{
   var bReturnValue = false;
   if bValue
   {
      bReturnValue = true;
   }
   return bReturnValue;
}
```

W wierszu `if` mamy `bValue` bez nawiasów, więc wystąpi następujący błąd:

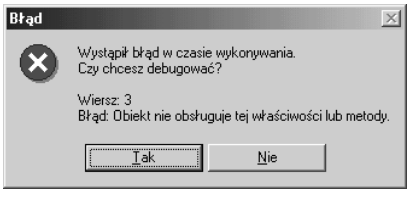


Użycie metody nieobsługiwanej przez obiekt

Użycie metody, której obiekt nie obsługuje jest powszechnie spotykaną pomyłką, zwłaszcza przy pracy z HTML DOM. Poniższy fragment kodu generuje błąd czasu wykonania:

```
var iValue = 123;
nValue = iValue.replace(2,4)
return iValue;
```

Powyższy kod usiłuje wykonać zastąpowanie łańcucha na liczbie całkowitej, co generuje poniższy błąd:



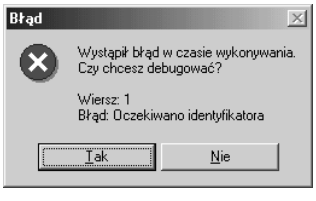
Użycie słowa zastrzeżonego

W JavaScripcie pewne słowa należą do składni języka, wobec czego mają w tym języku ustalone znaczenie. Są to **słowa zastrzeżone**. Większość z nich to słowa kluczowe, używane przez sam JavaScript, inne są zarezerwowane na przyszłość. Słowa zastrzeżone nie mogą być używane jako nazwy zmiennych, funkcji, obiektów ani metod. Jeśli spróbujemy użyć słowa zastrzeżonego, interpreter JavaScript wygeneruje błąd czasu ładowania.

W poniższym przykładzie spróbujemy zadeklarować zmienną o nazwie `case` (zastrzeżone słowo kluczowe):

```
var case;
```

Interpreter JavaScript zgłosi błąd:



Należy zwrócić uwagę, że `Case` jest dopuszczalną nazwą, ponieważ JavaScript rozpoznaje wielkość liter. Jednakże używanie słów kluczowych jest zawsze złym nawykiem, niezależnie od wielkości liter. Listę słów zastrzeżonych przedstawia dodatek A.

Cudzysłowy

Niewłaściwie umieszczone znaki cudzysłowu lub ich brak to pomyłki popełniane powszechnie w JavaScripcie , zwłaszcza gdy skrypty są generowane i kodowane dynamicznie, oraz gdy łańcuchy w JavaScripcie zawierają znaki cudzysłowu.

Cudzysłowy wokół nazw zmiennych i obiektów

Omyłkowe umieszczenie nazwy zmiennej lub obiektu w cudzysłowach powoduje, że JavaScript traktuje tę nazwę jak łańcuch. Może to powodować nieoczekiwane wyniki. Poniższy kod:

```
var FirstName = "John";
var LastName = "Doe";
alert("Witaj, " + FirstName + " " + "LastName" + ".");
```

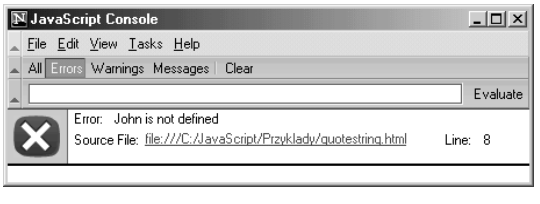
da wynik:

Brak cudzysłowów przy łańcuchach

Jeśli zapomnimy umieścić cudzysłowy wokół łańcucha, JavaScript potraktuje łańcuch jak zmienną lub obiekt, co może spowodować błąd czasu wykonania. Kod:

```
var FirstName = John;
var LastName = "Doe";
alert("Witaj, " + FirstName + " " + LastName + ".");
```

spowoduje błąd:

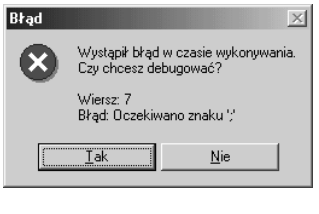


Niedomknięte cudzysłowy

Niedomknięte cudzysłowy również mogą wygenerować błędy JavaScriptu. Przy pracy z łańcuchami w JavaScripcie ważne jest poprzedzenie cudzysłowu wewnątrz łańcucha znakiem ucieczki. Oto przykład łańcucha generującego błąd:

```
var myString1 = "Potem powiedział "Zgoda!""
```

Komunikat o błędzie będzie wyglądać tak:



Łańcuch generuje błąd, ponieważ znaki cudzysłowu wewnątrz łańcucha muszą być poprzedzone znakiem ucieczki lub przynajmniej stosowane naprzemiennie: podwójne i pojedyncze. Oto kilka możliwych rozwiązań:

- Zastąp podwójne cudzysłowy w łańcuchu pojedynczymi:

```
var myString1 = "Potem powiedział 'Zgoda!'"
```

- Ujmij łańcuch w cudzysłów pojedynczy:

```
var myString1 = 'Potem powiedział "Zgoda!"'
```

- Preferowanym sposobem zapisywania cudzysłowów w łańcuchach jest użycie znaku ucieczki, zamiast zmiany cudzysłowów pojedynczych lub podwójnych:

```
var myString1 = "Potem powiedział \"Zgoda!\""
```

Dodatkowe informacje o znakach ucieczki zawiera rozdział 2., „Rdzeń języka JavaScript".

Brakujące znaki „+" przy konkatenacji

Przy konkatenacji łańcucha oraz zmiennej lub obiektu JavaScriptu łatwo zapomnieć o symbolu „+" pomiędzy łączonymi łańcuchami, zmiennymi lub obiektami. Brak operatora „+" bardzo łatwo przeoczyć, gdy usiłujemy odpluskwiać skrypt, który ciągnie się strona za stroną. Tak może wyglądać przykład, w którym nietrudno zapomnieć o operatorze konkatenacji:

```
var userName = "Joe";
var itemNumber = 1234;
var sURL;

sURL = "formPost.htm?user=" + userName + "&itemNumber=" itemNumber

document.location.href = sURL
```

Plus (+) powinien znaleźć się pomiędzy `"&itemNumber="` i `itemNumber`.

Operatory =, ==, ===

W JavaScripcie = jest operatorem przypisania, który wartość tego, co znajduje się przed nim, nastawia na to, co znajduje się po nim. == oznacza „jest równy", wykonuje porównanie wartości po obu stronach i zwraca wartość boolowską. === oznacza „jest ściśle równy" i dokonuje porównania zarówno wartości, jak i typu danych po obu stronach.

```
var i = 2;
alert(i=3)//zwraca 3
alert(i=="3")//zwraca true
alert(i==="3")//zwraca false
```

W tym przykładzie zmienna `i` jest deklarowana z wartością `2`. W drugim wierszu wartość zmiennej jest zmieniona na `3`, co zgłasza okno *alert*. `i=="3"` w trzecim wierszu zwraca `true`, ponieważ wartość `i` została zmieniona w poprzednim wierszu. Czwarty wiersz zwraca `false`, ponieważ `i` jest zmienną typu liczbowego.

Operatory <> i !=

Programiści ASP i VB zwykle używają <> jako „różny niż”. W JavaScripcie powoduje to błąd składni, ponieważ operatorem nierówności w tym języku jest `!=`.

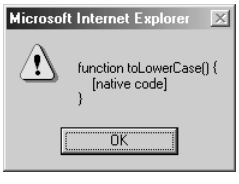
Metody jako własności i odwrotnie

W JavaScripcie para nawiasów musi następować po każdej metodzie, nawet dla metod nie przyjmujących parametrów. Po własnościach nigdy nie występuje para nawiasów.

W tym przykładzie po metodzie `toLowerCase` brak pary nawiasów, więc JavaScript usiłuje zinterpretować ją jako własność:

```
var myVar = "Witaj, Świecie!";
alert(myVar.toLowerCase);
```

Daje to dość ciekawe wyniki:

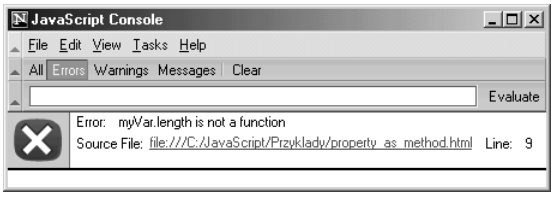


Wyniki w ramce ostrzegawczej są zasadniczo prototypem metody `toLowerCase`, którego metody i własności są nie wyliczonym kodem własnym obiektu JavaScript, do którego należy `toLowerCase()`.

Analogicznie, jeśli umieścimy nawias po własności, interpreter JavaScriptu będzie próbował zinterpretować własność jako metodę:

```
var myVar = "Witaj, Świecie!";
alert(myVar.length());
```

Generuje to błąd wykonania:



Niejasne instrukcje else

Czasami nie jest oczywiste, do której instrukcji `if` należy dana instrukcja `else`. W poniższym przykładzie niepoprawna jest zarówno logika, jak i formatowanie:

```
if(result!="wygrana")
   if(result=="przegrana")
      do_lose();
else
   do_win();
```

W tym przykładzie nie wygramy nigdy. Ramię `else` należy do drugiej z instrukcji `if`. Użycie nawiasów klamrowych (opisane w poprzednim rozdziale) eliminuje ten błąd.

```
if(result!="wygrana")
{
   if(result=="przegrana")
   {
      do_lose();
   }
}
else
{
   do_win();
}
```

Problemy z for ... in

Jak powiedzieliśmy w rozdziale 2., składnia `for ... in` jest używana do znajdowania nazw wszystkich własności obiektu JavaScript. Problem polega na tym, że metoda ta nie zawsze działa zgodnie z oczekiwaniami; niektóre własności nie pojawiają się, zaś część obiektów pozornie nie ma w ogóle własności. Niewiele możemy z tym zrobić, ponieważ taką decyzję o produkcie podjęli autorzy interpretera JavaScript. Problem taki zwykle pojawia się tylko wtedy, gdy dla swoich celów wycinamy własną ścieżkę do przeglądarki..

Możemy tylko:

- poznać z góry nazwy własności,
- wiedząc, dlaczego własności są ukryte przed wyliczeniem, zaakceptować to.

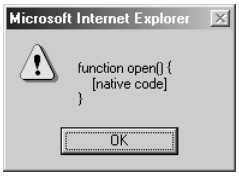
Aby poznać z góry własności obiektu, polecamy odwołać się do dokumentacji, na przykład w dodatkach do tej książki. Alternatywną taktyką jest sondowanie danego obiektu, własność po własności, przy użyciu składni tablicowej. Pochłania to wiele czasu i jest ostatnią deską ratunku.

Zgodnie ze standardem ECMA, dana własność obiektu ma kilka **atrybutów** opisujących tę własność, na przykład nazwę i wartość własności. Jednym z takich atrybutów jest `DontEnum`. Jeśli atrybut ten jest obecny, własność nie zostanie ujawniana przez pętlę `for ... in`. W ten sposób własności obiektów są pomijane w takich pętlach.

Własności obiektów mogą mieć atrybut `DontEnum` z kilku powodów. Najprostszym jest fakt, że niektóre własności są nieciekawe. Każda metoda obiektu jest również jego własnością. Nie ma sensu próbować oddziaływać z metodą jako własnością, jeśli obiekt jest obiektem macierzystym. Na przykład:

```
alert(document.open);
document.open = "Sałatka owocowa";
alert (document.open);
```

Skrypt wyświetli poniższy alarm i przy okazji wywoła zamieszanie w obiekcie `document`. Metoda `open()` nie będzie od tego momentu dostępna do tworzenia nowych okien przeglądarki.



Drugim powodem stosowania własności, których nie można wyliczyć jest to, że niektóre obiekty macierzyste nie trzymają się tych samych reguł co JavaScript. Przykładem mogą być klasy języka Java, udostępniane jako obiekty macierzyste JavaScriptu w przeglądarce. Własność JavaScriptu `java.lang` nie jest naprawdę obiektem, lecz **pakietem** typów obiektowych (inaczej zwanym biblioteką klas). `java.lang.io` jest pakietem w `java.lang`, ale nie jest obiektem. Pakiet to po prostu grupa pokrewnych obiektów, zgromadzonych razem. Klasy są zapisane na dysku i przesiewanie ich w poszukiwaniu składowych `java.lang`, które mogą być obiektami, nie jest zadaniem łatwym, efektywnym lub choćby przydatnym. I tak przeglądarka na to nie pozwala. Wobec tego ani on, ani inne pakiety związane z klasami języka Java, nie mogą być wyliczane przez JavaScript.

Powyższy opis dotyczy tylko klas języka Java używanych z JavaScriptu, a nie obiektów Java używanych z JavaScriptu. Obiekt Javy można wyliczyć przez JavaScript jak normalny obiekt.

Zadzieranie z typami

Łatwo jest naprędce sklecić skrypt JavaScriptu uruchamiany po stronie klienta. Język o słabych typach oszczędza mnóstwo czasu, który w przeciwnym razie trzeba by poświęcić na organizowanie danych właściwego rodzaju dla właściwego rodzaju zmiennych. Po ukończeniu pracy przychodzą ludzie ze szczególnymi zdolnościami, zwani potocznie użytkownikami, i wynajdują wielkie luki w naszych skryptach. Wprawdzie JavaScript posiada beztypowe zmienne, lecz same dane są już określonego typu, a użytkownicy na pewno znajdą sposób, aby skrypt wyłożył się na tej różnicy.

W rozdziale 1. opisano działanie konwersji typów w JavaScripcie. Najczęstszą pułapką jest pisanie wszystkich skryptów pod Netscape 4.0, aby potem odkryć, że nie działają w żadnej innej przeglądarce, ponieważ polegaliśmy na specjalnym zachowaniu operatorów JavaScript 1.2 `!=` i `==` w Netscape. Odradzamy użycie podczas programowania `<SCRIPT LANGUAGE='JavaScript`

1.2'>, chyba że przenośność nie jest w ogóle wymagana. Zapewne najbezpieczniej jest całkiem pominąć atrybut LANGUAGE, ponieważ JavaScript i tak jest domyślnym językiem skryptowym przeglądarek. Jeśli musimy go zastosować, lepiej użyć LANGUAGE='JavaScript'. Jeśli mamy pewność, że użytkownicy będą korzystać z naprawdę nowych przeglądarek, to dobrą opcją jest też LANGUAGE='JavaScript 1.3' lub LANGUAGE='JavaScript 1.4', ponieważ wtedy będziemy dysponować zdarzeniami obsługi błędów w tych wersjach.

Konwersja danych od użytkownika na liczby może być problematyczna. Jeśli program CGI wciąż pada, poczta e-mail nie dochodzi a JavaScript powoduje wyskakiwanie błędów w jakiejś przeglądarce, to najlepiej wrócić do sprawdzania poprawności wszystkich wartości liczbowych wprowadzanych przez użytkownika. Sprawdanie poprawności formularzy opisano w rozdziale 16.

Na koniec, starsze przeglądarki posiadają ograniczoną obsługę typów JavaScriptu (omówionych rozdziale 1.). Jeśli chcemy obsługiwać takie przeglądarki, lepiej nie przyzwyczajać się do null, NaN, typeof() lub obiektów Array.

Obcinanie łańcuchów

Uwaga na stałe łańcuchowe, które zawierają ponad 80 znaków. Niektóre starsze przeglądarki nie potrafią ich obsługiwać. Lepiej podzielić je na mniejsze części — tak łatwiej je czytać. W poniższym przykładzie są tylko krótkie łańcuchy, lecz ilustruje on zasadę:

```
var big_string = "pierwszy kawałek"
               + "drugi kawałek"
               + "trzeci kawałek";
```

Uwaga na problem pustego znaku w przeglądarkach Netscape. Wartość "xxx\000yyy".length wynosi w nich 3, z wyjątkiem późniejszych wersji Netscape 4.0; znaki yyy gubią się. Dla wszystkich przeglądarek próba pokazania takiego łańcucha za pomocą alert() lub document.write() jest ogólnie bezowocna, ponieważ łańcuch będzie rozumiany jako xxx a nie xxx\000yyy.

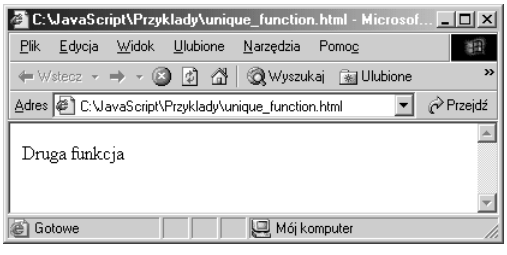
Jedyność nazw funkcji

Każda funkcja i zmienna muszą mieć własną niepowtarzalną nazwę. Powielanie nazw zmiennych lub funkcji niekoniecznie musi powodować błąd, lecz daje wyniki inne od spodziewanych:

```
function alert(s)
{
   document.write(s);
}
function myFunction()
{
   alert("Pierwsza funkcja");
}
function myFunction()
```

```
{
   alert("Druga funkcja");
}

myFunction();
```

W powyższym kodzie standardowa funkcja alertu JavaScript jest redefiniowana tak, że wykonuje zapis `document.write`. Następnie definiujemy funkcję, po czym redefiniujemy ją w kolejnym bloku. Wynik ilustruje, jak ważne są niepowtarzalne nazwy funkcji:



Użycie wyjątków

W JavaScripcie wszystko, co zgłasza błąd lub usiłuje wykonać niedozwoloną operację, jest definiowane jako **wyjątek**. Wyjątki, które nie są obsługiwane przez kod, generują znajome tajemnicze komunikaty o błędach systemowych i powodują przerwanie wykonywania skryptu. Poprzez obsługę wyjątków możemy wprowadzać własne komunikaty o błędach i uruchamiać własne funkcje obsługi błędów.

Gdy wywoływana jest funkcja lub metoda, jedynym opisanym jak dotąd mechanizmem przekazywania informacji z powrotem do fragmentu skryptu, który wywołał metodę lub funkcję, jest instrukcja `return`. Co się dzieje, gdy coś idzie nie tak? Pokazuje to poniższy przykład funkcji:

```
function add_like_a_child(nOne, nTwo)
{
   var vReturnValue;
   if (nOne + nTwo > 20)
   {
      vReturnValue = "Brakło palców u rąk i stóp do liczenia!";
   }
   else if (nOne + nTwo >10)
  {
      vReturnValue = "Brakło palców u rąk do liczenia!";
   }
   else
   {
      vReturnValue = nOne + nTwo;
   }
   return vReturnValue;
}
```

Problem z tą funkcją polega na tym, że przy każdym jej użyciu wymagany jest dodatkowy kod do sprawdzenia wartości zwracanej. Nigdy nie będziemy wiedzieć przed tym sprawdzeniem, czy funkcja zadziałała w sposób przydatny, czy nie. W takim przypadku możemy być zmuszeni do przeprowadzenia dwóch kontroli, po jednej dla każdego nietypowego przypadku. Gdyby funkcja była złożona, musielibyśmy przeprowadzać wiele kontroli. Oto przykład użycia tej funkcji:

```
var vResult = add_like_a_child(3, 5)
if ( vResult == "Brakło palców u rąk i stóp do liczenia!")
{
   // zrób coś, żeby poradzić sobie z problemem
}
else if ( vResult == "Brakło palców u rąk do liczenia!")
{
   // zrób coś innego, żeby poradzić sobie z problemem
}
//jeśli doszliśmy do tego miejsca, kontynuuj jak zwykle
```

W Javie trudne przypadki tego typu są obsługiwane za pomocą wyjątków. Ogólnym zadaniem wyjątków jest udostępnienie mechanizmu zgłaszającego nietypowe zdarzenia. Gdy mamy taki działający mechanizm, możemy być pewni, że zwracane wartości będą zgłaszać tylko normalne, poprawne dane wyjściowe funkcji. Gdy zatroszczymy się o przypadki wyjątkowe, nie będzie trzeba dodatkowo sprawdzać wartości zwracanych funkcji.

JavaScript do pewnego stopnia naśladuje składnię Javy. Wyjątki staną się w przyszłości ważną funkcjonalnością JavaScriptu. Wynika to stąd, że gdy chcemy użyć obiektu w skrypcie, musimy dysponować punktami dostępu, przez które można dostać się do obiektu, jego własności i metod. Takie punkty dostępu razem tworzą **interfejs**, inaczej **sygnaturę** obiektu. W terminologii komputerowej trzema głównymi cechami interfejsu obiektu są jego własności (inaczej atrybuty), metody i wyjątki. Wobec tego w JavaScripcie wyjątki są obecne.

W Javie musimy ściśle deklarować typ rzeczy, którą jest wyjątek. Zgodnie z duchem języka, w JavaScripcie wyjątkiem może być dowolny znany element, na przykład obiekt lub łańcuch. Oficjalną składnię instrukcji JavaScriptu obsługujących wyjątki przedstawiliśmy poniżej. Najpierw tworzymy wyjątek:

```
throw wyrażenie;
```

Następnie obsługujemy wyjątek:

```
try
   blok instrukcji
catch (identyfikator)    //wariant pierwszy, może być powtarzany
   blok instrukcji
catch (identyfikator if warunek)    //wariant drugi, może być powtarzany
   blok instrukcji
finally
   blok instrukcji    //opcjonalny, najwyżej jeden
```

Tak będzie wyglądać napisana na nowo poprzednia funkcja licząca na palcach:

```
function add_like_a_child(num1, num2)
{
   if ( num1 + num2 > 20 )
      throw "Brakło palców u rąk i stóp do liczenia!";
```

```
    if ( num1 + num2 > 20 )
       throw "Brakło palców u rąk do liczenia!";
    return num1 + num2;
}
```

Jeśli w tej funkcji instrukcja `throw` będzie w ogóle wykonana, to przetwarzanie funkcji zostanie zatrzymane i funkcja zwróci informację natychmiast, lecz zamiast zwykłej wartości zwracanej zostanie zwrócony wyjątek.

Napisany na nowo kod wywołujący funkcję będzie wyglądać tak (działa w przeglądarkach Netscape 6 i Internet Explorer 5):

```
var iResult;
try
{
   iResult=add_like_a_child(3,5);
   if (iResult == 8)
   {
      alert("Correct");
   }
   else
   {
      alert("Please try again");
   }
}
catch (sError)
{
   if ( sError == "Brakło palców u rąk i stóp do liczenia!" )
   {
      alert("Brakło palców u rąk i stóp do liczenia!");
   }
   if ( sError == "Brakło palców u rąk do liczenia!" )
   {
      alert("Brakło palców u rąk do liczenia!");
   }
}
finally
{
   alert (iResult);
}
```

W tym skrypcie zmienna `iResult` może nigdy nie zostać ustawiona w czwartym wierszu. Jeśli funkcja zwróci wyjątek, przetwarzanie przeskakuje natychmiast do instrukcji `catch`, która ma dostęp do wyjątku w zmiennej błędu, i zamiast zwykłych, wykonane zostają instrukcje w bloku po `catch`. Jeśli nie będzie wyjątku i funkcja normalnie zwróci wartość, to wykonane będą następne instrukcje w bloku `try`, a po jego zakończeniu blok `catch` zostanie całkowicie pominięty, podobnie jak nieużywane rozgałęzienie instrukcji `if`.

Całe objaśnienie będzie o wiele prostsze, jeśli znamy trochę żargonu. Gdy coś idzie źle, mówimy że funkcja zgłasza wyjątek. Funkcja jest wywoływana w bloku `try`, a wyjątki są obsługiwane w blokach `catch`. Blok `finally` jest wykonywany zawsze, jeśli na końcu nie pozostanie nierozstrzygnięty wyjątek.

Zasady działania tego procesu są następujące:

- Jeśli w funkcji (lub w dowolnym bloku kodu) dotrzemy do instrukcji `throw`, funkcja ta nie będzie dalej przetwarzana, nie zwróci wartości i nie zwróci `void`. Zamiast tego przestanie działać natychmiast i zostanie zgłoszony wyjątek.
- Jeśli instrukcja lub funkcja powodująca wyjątek nie znajduje się wewnątrz bloku `try`, wystąpi błąd interpretera i skrypt zostanie zatrzymany. Jeśli instrukcja lub funkcja jest wywołana ze środka innej funkcji, metody lub bloku `try`, to cały ten zbiór instrukcji zostanie również zaniechany, zaś wyjątek będzie przekazany wyżej do następnego poziomu sterowania, dopóki nie zostanie napotkany blok `try` lub nie wystąpi błąd.
- Jeśli instrukcja lub funkcja powodująca wyjątek znajduje się wewnątrz bloku `try`, wszystkie kolejne instrukcje w bloku są ignorowane, a interpreter przegląda wszystkie bloki `catch`, jakie mogą być obecne. Jeśli znajdzie blok z zadowalającymi kryteriami, to ten blok `catch` będzie wykonany. Po wykonaniu wszystkich bloków `try` i `catch` zgodnie z wyjątkiem, zostaje wykonany blok `finally`.

Jak wyglądają kryteria bloku `catch`? Możliwe są dwa przypadki, przedstawione poniżej:

```
catch (błąd) {Wykonaj instrukcje obsługi błędu}
```

oraz

```
catch (błąd if błąd > "Błąd 1")) {Wykonaj instrukcje obsługi błędu}
```

W pierwszym przypadku blok `catch` pasuje do wszystkich wyjątków. W drugim przypadku blok `catch` pasuje tylko do wyjątków, które spełniają warunek `if`. Jeśli bloków `catch` jest więcej, to pierwszy przypadek wyłapie wszystko, podobnie jak warunek `default:` instrukcji `switch`, więc rozsądnie jest umieścić go na końcu listy bloków `catch`. Zmienna `error` śledzi wyjątek i w bloku `catch` gra rolę podobną do parametru funkcji. Blok `catch` traktuje zmienną `error` w sposób bardzo zbliżony do tego, jak poniższy skrypt traktuje zmienną `error2`:

```
function HandleError(error2)
{
   if(error2 == "ErrorType1")
   {
      //procedura wyjątku dla warunku 1
   }
   if(error2 == "ErrorType2")
   {
      //procedura wyjątku dla warunku 2
   }
}
```

Składnia wyjątków a „if"

Ktoś, kto nigdy nie używał wyjątków, może być bardzo sceptyczny względem składni, która jest wymagana, aby zadziałały. Najczęstsze zastrzeżenia do powyższego przykładu są takie:

- Jest bardziej opisowy. Zajmuje więcej wierszy niż zwykłe wystąpienie `if`, więc wymaga też więcej pracy.
- Strukturalnie bardzo przypomina pierwszy przykład. W czym jest lepszy?
- Język jest obcy. Co jest nie tak ze starym, poczciwym `if`?

W prostych przykładach istotnie argumenty za wyjątkami są mało przekonujące; kod oparty na wyjątkach może być zdecydowanie dłuższy. Skrypty, które po prostu wykonują ciąg niezawodnych operacji, tak naprawdę w ogóle nie potrzebują wyjątków. Bardziej ogólna argumentacja za użyciem składni wyjątków jest następująca:

- Składnia wyjątków jest bardziej przejrzysta, gdy wiele rzeczy na raz idzie źle.
- Gdy się do niej przyzwyczaimy, składnia wyjątków jest łatwiejsza do odczytania, ponieważ uczymy się ignorować bloki `catch`. Bloki `catch` to około 1% faktycznie uruchamianego kodu (błędy są rzadkie) i zawierają 99% nieistotnych szczegółów (naprawianie błędów jest kłopotliwe).
- Wyjątki są przykładami „czystego kodu" i dobrego stylu w programowaniu.
- Wyjątki są jedną z trzech głównych funkcjonalności obiektów (dwie pozostałe to atrybuty i metody).

Oto przykład pokazujący, że składnia wyjątków pracuje na korzyść autora skryptu, a nie przeciwko niemu:

```
//kawałek kodu, który może zawieść na 20 różnych sposobów
try
{
   //możliwy powód jednego z 5 wyjątków
   ryzykowne_zadanie_1();
   //możliwy powód jednego z 4 wyjątków
   ryzykowne_zadanie_2();
   //możliwy powód jednego z 7 wyjątków
   ryzykowne_zadanie_3();
   //możliwy powód jednego z 4 wyjątków
   ryzykowne_zadanie_4();
}
catch (everything)
{
   // drzyj się wniebogłosy
}
```

Ponieważ instrukcja `throw` natychmiast przerywa wykonanie bloku `try`, każda funkcja może nastąpić zaraz po poprzedniej i na pewno późniejsze nigdy nie zostaną wykonane. W tym przypadku zaoszczędziliśmy wielu testów `if` przez użycie bloku `try`. Proszę zwrócić uwagę, że funkcje nie muszą mieć wartości zwracanej innej niż `void`, aby móc skorzystać z wyjątków.

Oprócz tego nie musimy ograniczać się do zgłaszania prostych typów. Oto przykład:

```
// zgłasza obiekt
throw { err_string:"Już po tobie", err_number: 23 };
```

Powyższa instrukcja utworzyła obiekt o dwóch własnościach i zwróciła go jako zgłoszony element. Z drugiej strony, poniższa instrukcja oczekuje, iż wyjątek będzie obiektem z własnością `err_number`, co ewidentnie się stanie, gdy poprzednia instrukcja `throw` będzie odpowiedzialna za wyjątek:

```
// przechwytuje tylko błąd nr 23
catch ( e if ( e.err_number == 23) ) { … }
```

Wyjątki, zdarzenia i błędy

Czytelnik, który eksperymentował z JavaScriptem w przeglądarkach, być może wie, że istnieją zdarzenia wyzwalające, które mogą powodować uruchomienie fragmentów skryptu JavaScript, zwłaszcza gdy dana strona zawiera formularze. Tego typu zdarzenia ogólnie nie są uważane za wyjątki. Zdarzenia występujące jako pewien rodzaj polecenia użytkownika lub działania z sieci, są zjawiskiem normalnym i zwykle obsługiwane są tak samo jak zwykłe dane. Wyjątki nie są uważane za zjawiska normalne i zwykle używane są do warunków błędów, gdy pójdzie źle coś, co skrypt mógł „przewidzieć" (używając bloków `try ... catch`), lecz nie mogą być normalnie obsługiwane.

Czasami można pomylić zdarzenia z wyjątkami. Powodem jest to, że sposób przygotowania się skryptu do zdarzeń jest zwykle nieco odmienny od sposobu przyjmowania bardziej podstawowych poleceń i danych wprowadzanych przez użytkownika, i może trochę przypominać nietypowo wyglądającą obsługę wyjątków. Zdarzenia i wyjątki są jednakże od siebie różne i odrębne. Rozdział 5., „Formularze i dane", dał pojęcie, jak wyglądają zwykłe zdarzenia, a jeśli wrócimy do niego, zobaczymy że nie wspomina o składni wyjątków w JavaScripcie.

Wyjątki mogą się przydać dla obiektów macierzystych

Dodatkowy powód wprowadzenia obsługi wyjątków w JavaScripcie możemy znaleźć w obiektach macierzystych (wbudowanych obiektów i metod, takich jak obiekt `date` lub metoda `replace`). Bez obiektów macierzystych JavaScript jest niemal bezużyteczny. A gdyby te obiekty generowały wyjątki w ramach swojego interfejsu atrybut-metoda-wyjątek? W JavaScripcie musi istnieć mechanizm obsługujący ten typ komunikacji.

Wyjątki mogą się przydać przy błędach interpretera

Ze skryptami w JavaScripcie mogą dziać się różne bardzo złe rzeczy. Jeśli autor skryptu zrobił literówkę w jakimś ciemnym kącie dużego skryptu, błąd taki może nie zostać wykryty, dopóki skrypt nie zostanie zainstalowany w jakimś strategicznym miejscu. Takie jest ryzyko związane z językami interpretowanymi.

Jeśli interpreter odczyta skrypt i znajdzie ten błąd, warto by mógł coś z tym zrobić, zamiast zatrzymania całego skryptu. Taki jest przyszły kierunek standardu ECMAScript — błędy pisania skryptu mogą być zgłaszane jako wyjątki i obsługiwane przez system, który uruchamia interpreter JavaScript.

Jak pisać kod czysty i bezbłędny

Najszybszą metodą usuwania błędów w skrypcie jest ...napisanie go tak, aby od razu nie zawierał błędów. Pisanie czystego kodu jest najlepszym sposobem na unikanie pomyłek przy kodowaniu. „Czysty kod" oznacza dobrą organizację, samodokumentowanie, wcięcia i komentarze. Poprzedni rozdział zawiera kilka solidnych wytycznych dotyczących pisania czystego kodu.

Kod modularny

Przy pisaniu skryptów w JavaScripcie ważna jest jak najbardziej modularna konstrukcja kodu. Kod modularny jest podzielony na wiele różnych funkcji, z których idealnie każda wykonuje jedno określone zadanie. Modularność kodu redukuje powielanie i ogólną złożoność skryptu. Nigdy też nie odwołuje się bezpośrednio do zmiennych lub elementów HTML, lecz przyjmuje je jako dane wejściowe. Jedynymi zmiennymi deklarowanymi w funkcjach są zmienne lokalne. Zmienne zewnętrzne nie są ustawiane bezpośrednio przez funkcje, lecz przez wartość zwróconą przez funkcję. Można zajść za daleko z modularyzacją, zwłaszcza gdy zaczynamy pisać skrypty, które wykonują lub powielają wbudowane funkcje JavaScriptu.

Sprawdzanie istnienia obiektów

Błędy czasu wykonania „Brak definicji …" są błędami chyba najczęściej spotykanymi przy pisaniu kodu w JavaScripcie. Możemy wyeliminować takie błędy niemal całkowicie, sprawdzając czy obiekt istnieje przed próbą dostępu do jego własności lub metod. Ogólnie mówiąc, blok podobny do poniższego powinien być wstawiany gdzieś na początku skryptu i (lub) każdej funkcji, w której odwołujemy się do obiektu:

```
try
{
   if(obiekt lub własność)
   {
      //postępuj zgodnie z planem
   }
}
catch (error)
{
  // tu wstaw obsługę błędu
}
```

Techniki uruchamiania

Dla programów w JavaScripcie dostępnych jest wiele skutecznych technik uruchamiania, od banalnych do wyczerpujących. Techniki uruchamiania, które wykorzystują nowe okna, pisanie do bieżącego dokumentu lub nowego okna oraz alerty powinny być usunięte lub oznaczone jako komentarz przed wprowadzeniem kodu do eksploatacji.

JavaScript po stronie klienta

Dla JavaScriptu osadzonego w HTML-u najprymitywniejszym narzędziem uruchomieniowym jest `document.write()`. Nawet jeśli spowoduje zamieszanie w dokumencie HTML, to przynajmniej da nam jakieś informacje zwrotne podczas testowania. W połączeniu z poleceniem `window.open()` możemy te informacje wysłać do innego okna, jeśli tego chcemy. Na przykład:

```
var newWindows = window.open("", "newWindow"); newWindow.document.write("Witaj,
  świecie!");
```

Skrypty JavaScriptu na potrzeby plików konfiguracji *proxy* dla połączenia internetowego oraz dla plików preferencji w przeglądarkach Netscape (np. *prefs.js*, omówiony w dalszej części rozdziału) nie poddają się uruchamianiu — musimy je testować metodą prób i błędów.

Alarmy

Użycie funkcji `window.alert()`, osadzonych w skryptach JavaScript, jest najprostszym sposobem zatrzymania interpretera JavaScript w danym punkcie lub punktach podczas ładowania dokumentu. Wiadomość w oknie alarmu może też zawierać dowolne interesujące nas dane. Jest to najprostszy sposób na uruchamianie złożonej logiki, zawierającej mnóstwo instrukcji `if` i wywołań funkcji — wystarczy dodać `alert()` do podejrzanego odgałęzienia funkcji i będziemy wiedzieć czy jest wykonywane, czy nie. Alarmy są też szczególnie skuteczne w połączeniu z URL JavaScriptu. Poniższy przykład zatrzymuje wykonywanie skryptu za pomocą alarmu, aby sprawdzić wartość zmiennej po każdej instrukcji:

```
var myString = "Część pierwsza";
alert(myString);

var myString += ", Część druga";
alert(myString);
```

Konwencja URL javascript:

Po pełnym załadowaniu dokumentu HTML funkcja `document.write()` nie jest już zbyt przydatna, wbudowane alarmy nie pomogą, jeśli skrypt zakończył pracę. Notacja URL `javascript:`, którą możemy w każdej chwili wpisać w polu adresu lub lokalizacji w oknie przeglądarki, pozwala nam sondować przeglądarkę i dokument i sprawdzić, jak wygląda aktualny stan. Poniższy przykład pokazuje URL okna:

```
javascript:alert(top.location.href);
```

Jednakże konwencji URL `javascript:` możemy używać bardziej ogólnie. Przy pewnej cierpliwości i ostrożności możemy wprowadzić dowolną ilość JavaScriptu. Poniższy przykład wyświetla numery wszystkich elementów pierwszego formularza, posiadających własności z wartością:

```
javascript:var i; for (i=0;i<document.forms[0].elements.length;i++) { alert(i); }
```

Możemy wywołać każdą funkcję i metodę w przeglądarce, które są w jakiś inny sposób dostępne dla JavaScriptu. Oznacza to, że możemy bezpośrednio wymusić `submit()` formularza, wykonać `click()` dla przycisku lub wywołać dowolną własną funkcję. Poniższy przykład usuwa cookie za pomocą popularnej procedury, która, jak zakładamy, jest zawarta gdzieś w dokumencie:

```
javascript:DeleteCookie("biscuit3");
```

Rejestrowanie na konsoli Javy

W Netscape Navigatorze 4+, jeśli nie chcemy zakłócić ładowania dokumentu, lecz chcemy zapisać, co zaszło, możemy rejestrować informacje w konsoli *Javy*. Możemy obserwować konsolę podczas ładowania dokumentu lub przeanalizować później. Podejście takie jest bardzo podobne do `document.write()`, wystarczy w strategicznych punktach skryptu wstawić instrukcje w rodzaju:

```
java.lang.System.out.println("Nazwa okna to: "+top.name);
```

Druga wersja nie zapisuje na konsoli znaków końca wiersza:

```
java.lang.System.out.print('więcej...');
```

Rejestrowanie w pliku

Jeśli rejestrowanie wyjścia na konsoli nie jest wystarczająco trwałe, skrypty JavaScript po stronie klienta mogą pisać do plików na dysku lokalnym testowego komputera. Wystarczy wyłączyć w przeglądarce zabezpieczenia nie pozwalające na tego typu zachowanie:

```
var err = "Jakiś błąd skryptu";
var oFSO = new ActiveXObject("Scripting.FileSystemObject");
var oFile = oFSO.CreateTextFile("C:\ErrLog.txt");
oFile.WriteLine(err);
```

Microsoft Script Debugger

Script Debugger Microsoftu jest imponującym środowiskiem uruchomieniowym w stylu IDE, które testuje skrypty na obecność błędów i poprawia je w skryptach napisanych w dowolnym języku skryptowym, obsługującym ActiveX (JavaScript, VBScript, WSH itp.). Microsoft Script Debugger współpracuje ściśle z Internet Explorerem, dostarczając kontekstowych informacji uruchomieniowych po napotkaniu błędu w skrypcie. Microsoft Script Debugger może też być używany samodzielnie do tworzenia i uruchamiania skryptów poza przeglądarką.

Jak zdobyć Microsoft Script Debugger

Microsoft Script Debugger możemy pobrać za darmo z *http://msdn.microsoft.com/scripting/default.htm?/scripting/debugger/default.htm*. Jeśli powyższy URL będzie niepoprawny, otwórz *http://www.microsoft.com/* i szukaj *Microsoft Script Debugger*.

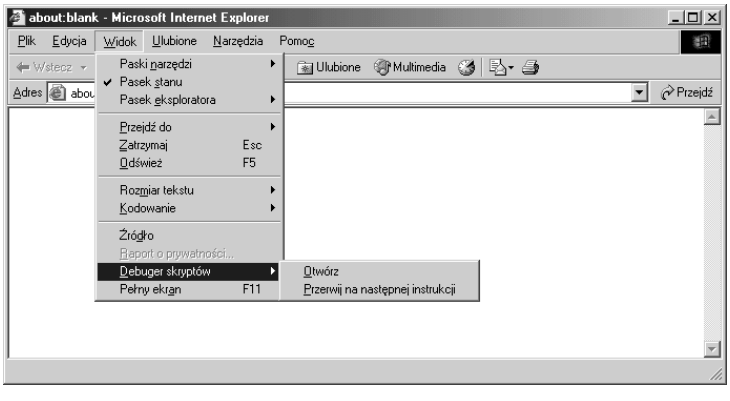
Program może być już zainstalowany, jeśli w komputerze jest zainstalowany dowolny z poniższych:

- Składniki Microsoft Visual Studio (Visual Interdev, Visual Basic, Visual C++ itp.).
- IIS lub Microsoft PWS.
- Windows 2000.

Z powyższymi aplikacjami instalowana jest zaawansowana wersja programu, która pełni te same funkcje co wersja podstawowa (dostępna przez URL podany powyżej), lecz ma więcej funkcji i nieco inny układ ekranu. Wszystkie ekrany w niniejszym rozdziale pochodzą z wersji podstawowej 1.0 z IE 6.0, ponieważ jest powszechnie dostępna i może być zainstalowana razem z wersją zaawansowaną i działać niezależnie od niej.

Jak włączyć Microsoft Script Debugger

Po zainstalowaniu najłatwiej ustalić, czy Microsoft Script Debugger jest włączony poprzez uruchomienie Microsoft Internet Explorera i kliknięcie pozycji *Widok* z paska menu.



Jeśli w rozwijanym menu pojawi się opcja *Debuger skryptów*, oznacza to, że program jest już dostępny. W przeciwnym razie, aby włączyć debuger skryptów, kliknij opcję *Narzędzia* na pasku menu i wybierz *Opcje internetowe*.

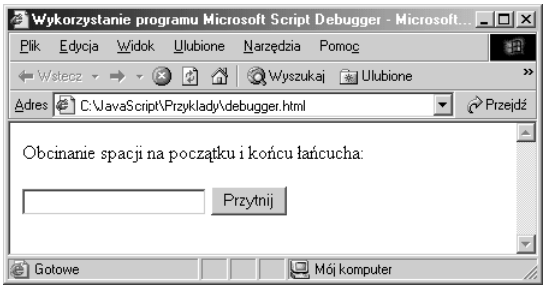
W oknie *Opcje internetowe*, które się otworzy wybierz zakładkę *Zaawansowane*. Jeśli opcja *Wyłącz debugowanie skryptów* pod nagłówkiem *Przeglądanie* jest zaznaczona, usuń jej zaznaczenie i kliknij *OK*. Opcja uruchamiania skryptów jest już włączona. Uwaga: niektóre systemy wymagają restartu IE, aby zmiana ustawienia przyniosła skutek.

Jak używać programu Microsoft Script Debugger

Zainstalowany i włączony debuggger możemy użyć do uruchamiania skryptów sprawiających problemy. Do większości przykładów użyjemy poniższego dokumentu, zapisanego pod nazwą *VBTrim.html*:

```
<!-- VBTrim.html -->
<HTML>
<HEAD>
   <TITLE>Wykorzystanie programu Microsoft Script Debugger</TITLE>

   <SCRIPT LANGUAGE='JavaScript'>
   function VBTrim(txtInput)
   {
      //Najpierw sprawdzimy, czy łańcuch wejściowy zawiera przynajmniej jeden znak
      if (txtInput.length > 0)
      {
         //Ustawiamy w zmiennej lokalnej wartość łańcucha wejściowego
         var sToTrim = txtInput;

         //Teraz usuwamy spacje z początku
         while (sToTrim.substring(0,1) == ' ')
         {
            sToTrim = sToTrim.substring(1, sToTrim.length);
         }

         //Na koniec usuwany spacje z końca łańcucha
         while (sToTrim.substring(sToTrim.length-1,sToTrim.length) == ' ')
         {
            sToTrim = sToTrim.substring(0, sToTrim.length-1);
         }

         //Zwracamy przycięty łańcuch
         return sToTrim;

      }

      else //W polu tekstowym nie zostały wpisane żadne znaki
      {
         alert("Nie mam żadnego łańcucha do przycięcia!");
         return false;
      }

   }
   </SCRIPT>

</HEAD>
<BODY>
   Obcinanie spacji na początku i końcu łańcucha: <BR><BR>
   <INPUT TYPE='text' name='txtValue'>
   <input type='button'
   onClick="txtValue.value = VBTrim(txtValue.value)" value="Trim">
</BODY>
</HTML>
```

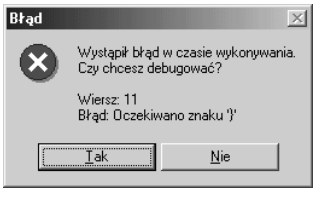
Czy potrafisz wskazać błąd? Wskazówka: jest to błąd logiczny, który pojawia się tylko w określonych warunkach. Powyższy kod daje następujący wynik w przeglądarce:



Jak uruchomić Microsoft Script Debugger

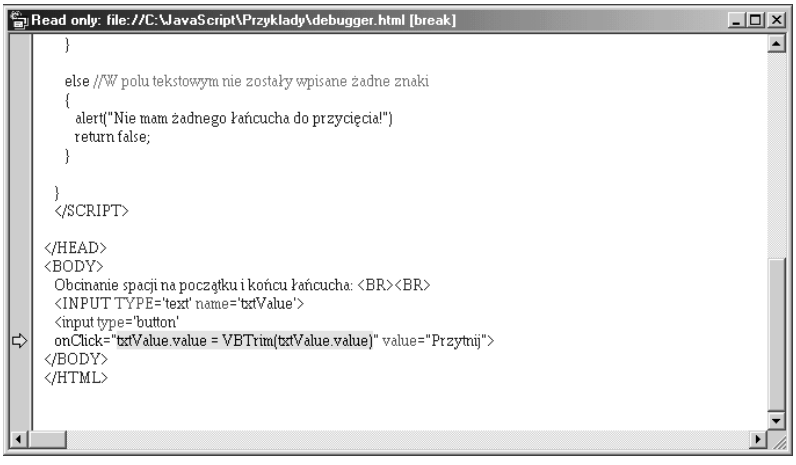
Podobnie jak w większości aplikacji, w programie Microsoft Script Debugger możemy wykonywać zadania w różnoraki sposób. Możemy go uruchomić:

- Z programu wykonywalnego w katalogu, w którym Script Debugger został zainstalowany. W ten sposób zostaje załadowana nowa, pusta kopia debugera. Aby dalej używać programu, musimy otworzyć dokument lub utworzyć nowy.
- Z Internet Explorera przez *Widok/Debuger skryptów/Otwórz* w pasku menu. W ten sposób do debuggera ładowany jest bieżący dokument Internet Explorera (tylko do odczytu). Możemy też użyć opcji *Przerwij na następnej instrukcji* z tego samego menu. Debuger skryptów zostanie otwarty przy następnym uruchomieniu skryptu.
- Ze skryptu. Słowo kluczowe `debugger` zapisane w skrypcie powoduje otwarcie programu Microsoft Script Debugger z bieżącym dokumentem, gdy tylko wykonywanie skryptu dojdzie do tego wiersza (VBScript stosuje słowo kluczowe `stop`).
- Z komunikatu o błędzie skryptu. Gdy debuger skryptów jest włączony, przy każdym napotkaniu błędu w skrypcie zobaczymy okno podobne do poniższego, dające nam okazję do usunięcia błędu:

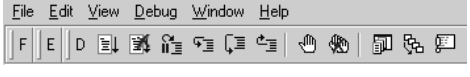


Znamy już szereg sposobów uruchomienia programu, więc otwórzmy nasz testowy dokument HTML (patrz powyżej) w Internet Explorerze 6. Po otwarciu dokumentu w IE wybierz *Widok* z paska *Menu*, a następnie *Debuger skryptów* i *Przerwij na następnej instrukcji*. Na koniec kliknij przycisk *Przytnij* (odświeżenie strony nie zadziała w naszym przypadku, ponieważ w chwili załadowania dokumentu HTML nie są wykonywane żadne instrukcje).

Debuger otwiera okno podobne do poniższego, z instrukcją, która wywołała otwarcie debugera podświetloną na żółto:



Teraz jest dobra chwila na to, by objaśnić znaczenie różnych przycisków i menu, dostępnych w oknie *Microsoft Script Debuggera*:



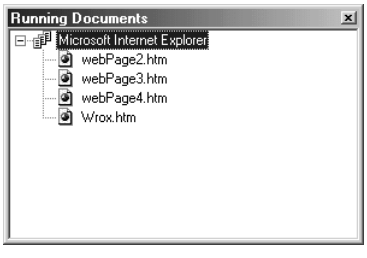
Nazwa funkcji	Opis	Ikona	W menu
Run	Zaczyna lub kontynuuje wykonywanie aktualnie wybranego skryptu lub dokumentu.		*Debug*
Stop Debugging	Anuluje uruchamianie aktualnie wybranego skryptu lub dokumentu.		*Debug*
Break at Next Statement	Wchodzi do debugera przy wykonaniu następnej instrukcji.		*Debug*
Step Into	Wykonuje następną instrukcję.		*Debug*
Step Over	Pomija następną instrukcję.		*Debug*
Step Out	Wychodzi z bieżącej funkcji.		*Debug*
Toggle Breakpoint	Załącza lub wyłącza punkt kontrolny.		*Debug*
Clear All Breakpoints	Usuwa wszystkie punkty kontrolne.		*Debug*
Running Documents	Otwiera lub zamyka okno *Running Documents* (opisane poniżej).		*View*
Call Stack	Otwiera lub zamyka okno *Call Stack* (opisane poniżej).		*View*
Command Window	Otwiera lub zamyka okno *Command* (opisane poniżej).		*View*

Okna debugera

Oprócz głównego okna, które jest otwierane przy każdym wywołaniu debugera, dostępne są jeszcze trzy mniejsze okna, które możemy otworzyć, dając większą elastyczność narzędzia.

Okno Running Documents

Okno *Running Documents* wyświetla listę wszystkich aplikacji mieszczących dokumenty, które zawierają aktywne skrypty. Kliknij + obok nazwy aplikacji (np. Internet Explorer), aby rozwinąć listę uruchomionych aktualnie dokumentów. W poniższym przykładzie w Internet Explorerze otwarte są cztery różne dokumenty *.htm*.



Kliknij dwukrotnie dokument, aby załadować do debuggera skryptów lub kliknij prawym przyciskiem myszy, aby ustawić punkt kontrolny dla następnej instrukcji.

Okno Call Stack

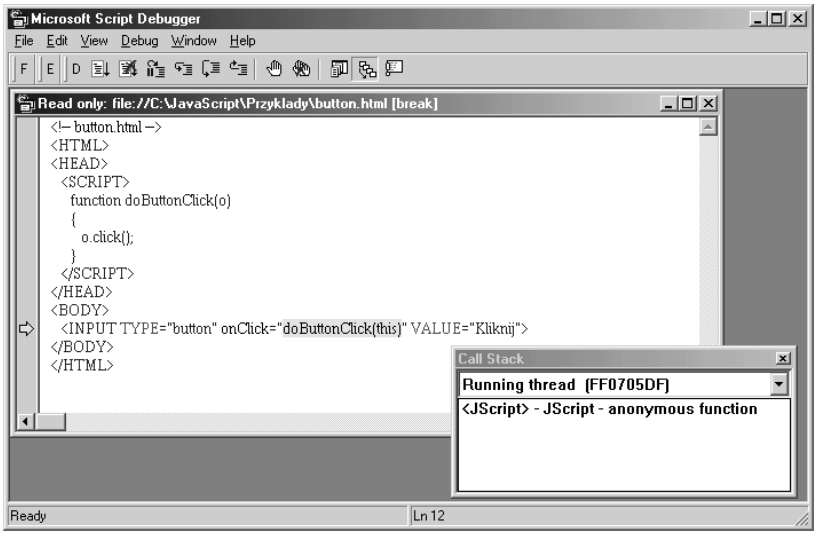
Okno *Call Stack* wyświetla listę wszystkich aktualnie uruchomionych wątków i procedur.

1. Otwórz poniższy dokument HTML (raczej bezsensowny) w IE.

```
<!-- button.html -->
<HTML>
<HEAD>
   <SCRIPT>
      function doButtonClick(o)
      {
         o.click();
      }
   </SCRIPT>
</HEAD>
<BODY>
   <INPUT TYPE="button" onClick="doButtonClick(this)" VALUE="Kliknij">
</BODY>
</HTML>
```

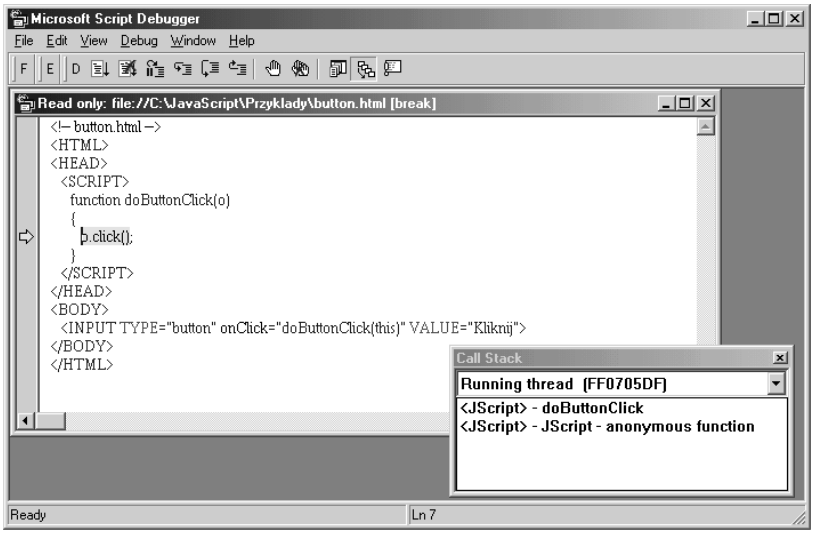
2. Wybierz z paska menu IE *Widok/Debuger skryptów/Przerwij na następnej instrukcji*.
3. Kliknij przycisk *Kliknij* na stronie WWW.
4. Gdy otworzy się debuger skryptów, kliknij przycisk *Call Stack*, aby otworzyć okno *Call Stack* (jeśli nie jest już otwarte).

5. W oknie debugera powinno być widoczne podświetlone zdarzenie `onClick`, które wywołało debuger, natomiast okno *Call Stack* powinno wyświetlić wartość *JScript — anonymous function*, przypisaną do zdarzenia `onClick`:



6. Klikaj przycisk *Step Into* aż do podświetlenia wiersza zawierającego `o.click()`.

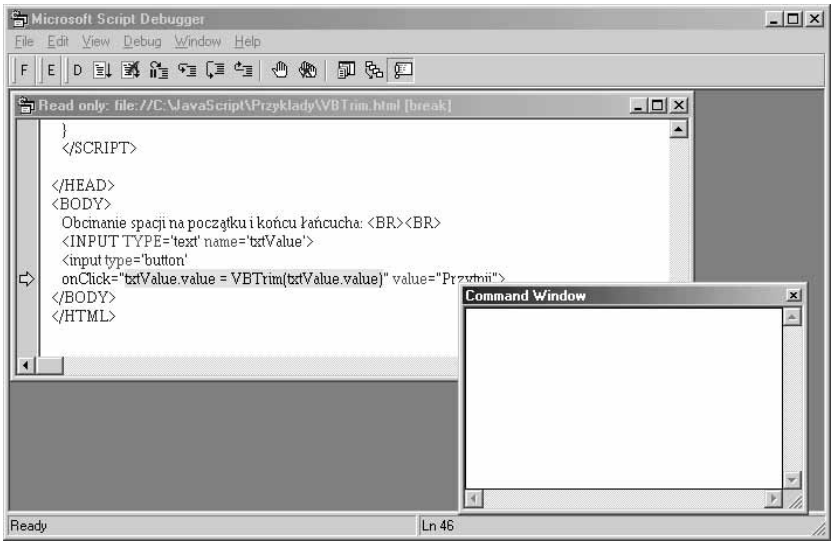
7. Debuger powinien wyglądać teraz jak ekran poniżej, z podświetlonym wierszem `o.click();` i oknem *Call Stack* zawierającym `doButtonClick()` (naszą własną funkcję).



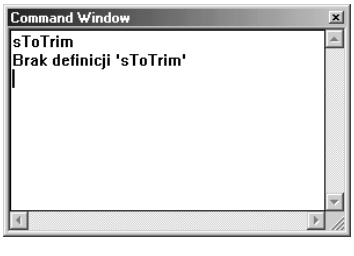
Każda wykonywana instrukcja jest dodawana u góry okna *Call Stack*. Zostaje usunięta po zakończeniu wykonywania i zwróceniu sterowania do procedury wywołującej. Aby przejść do wiersza z instrukcją zawierającą funkcję z okna *Call Stack*, po prostu kliknij dwukrotnie ten wiersz. Jeśli w uruchomionym dokumencie jest więcej niż jeden wątek, poszczególne wątki mogą być wybierane z listy rozwijanej.

Okno Command

Okno *Command* wyświetla informacje uruchomieniowe oparte na wpisywanych przez nas poleceniach uruchomieniowych. Na przykład, otwórz *VBTrim.html* w IE i kliknij *Przerwij na następnej instrukcji*, wpisz „ 5” w polu tekstowym (3 spacje i cyfrę 5), a następnie kliknij *Przytnij*. Otwórz okno poleceń przyciskiem *Command Window*. Debuger powinien wyglądać tak:



Wpisz w oknie poleceń `sToTrim` (nazwę naszej zmiennej) i naciśnij *Enter*. W oknie powinien pojawić się tekst:

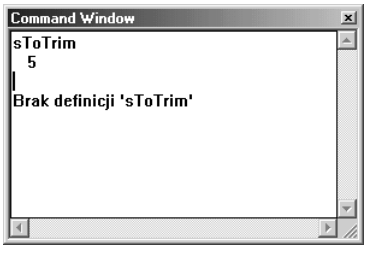


Mówi to nam, że zmienna nie jest zdefiniowana, ponieważ ma tylko zasięg funkcji, a funkcja ta jeszcze nie została wywołana.

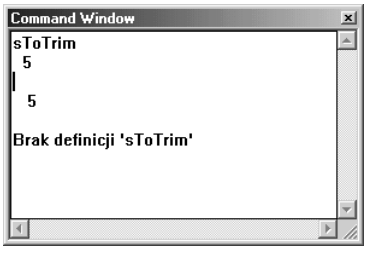
Kliknij kilkakrotnie przycisk *Step Into*, aż do podświetlenia instrukcji:

```
while (sToTrim.substring(0,1) == ' ')
```

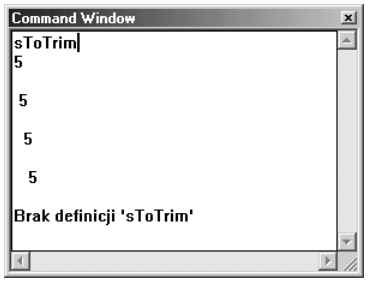
Przemieść kursor do pierwszego wiersza w oknie poleceń (powinna w nim być zmienna `sToTrim`) i naciśnij *Enter*. Teraz okno poleceń powinno wyglądać tak:



Wartość `sToTrim` wynosi teraz „ 5” — taką wartość przekazaliśmy. Kliknij jeszcze dwa razy *Step Into*, aż wróci do pierwszego wiersza `while`. Ponownie sprawdź wartość `sToTrim`. Okno poleceń powinno teraz wyglądać tak:



Pierwsza spacja została usunięta i wartość `sToTrim` wynosi teraz „ 5” (dwie spacje i cyfra 5). Powtórz te same kroki i kliknij jeszcze dwa razy *Step Into*, aż wrócimy do pierwszego wiersza `while` i sprawdź wartość `sToTrim`, która powinna teraz wynosić „ 5”. Powtórz sekwencję jeszcze raz. Okno poleceń powinno teraz wyświetlać:



Zmienna `sToTrim` została całkowicie pozbawiona spacji z początku. Następne kliknięcie przycisku *Step Into* powoduje przejście do instrukcji `while`, sprawdzającej obecność spacji na końcu łańcucha, których w naszym łańcuchu nie ma. Kolejne kliknięcie *Step Into* przeniesie nas do instrukcji `return`. Kliknij przycisk *Stop Debugging* i sprawdź wyniki przejścia przez kod za pomocą debuggera w oknie IE, w którym strona jest otwarta. Wartość w polu tekstowym została z powodzeniem przycięta.

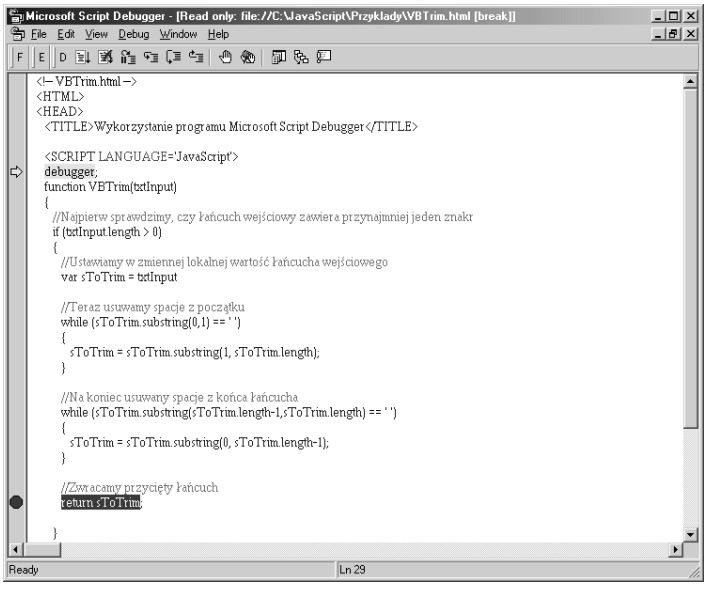
Ustawianie punktów kontrolnych

Punkty kontrolne (*breakpoint*) są skutecznym narzędziem zatrzymywania wykonywania skryptu po osiągnięciu określonego punktu. Na przykład, otwórzmy debugger z naszym dokumentem *VBTrim.html*, tym razem przez umieszczenie słowa kluczowego `debugger` bezpośrednio przed instrukcją funkcji:

```
<SCRIPT LANGUAGE='JavaScript'>
   debugger;
   function VBTrim(txtInput)
   {...
```

Otwórz teraz dokument w IE. Debugger uruchamia się od razu (z podświetloną instrukcją `debugger`), ponieważ instrukcja `debugger` ma zasięg globalny i nie jest zamknięta w funkcji. Dzięki temu jest wykonywana przy załadowaniu strony.

Ustawmy teraz punkty kontrolne. Umieść kursor w wierszu `return sToTrim` i kliknij przycisk *Toggle Breakpoint*. Okno debugera powinno teraz wyglądać tak:



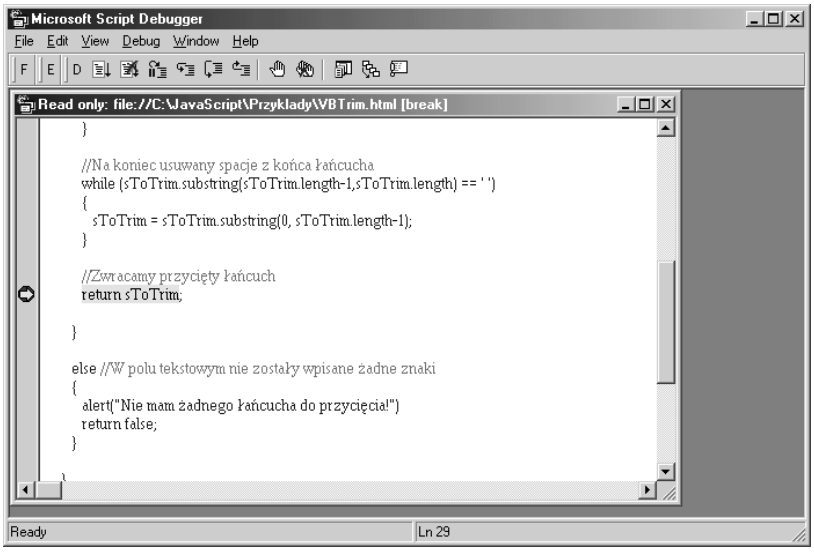
Debuger podświetla punkt kontrolny na czerwono. Następnym krokiem będzie kliknięcie przycisku *Run* na pasku zadań debugera, co zwraca sterowanie do IE. Wróć do dokumentu w IE i wpisz `4` w polu tekstowym. Kliknij przycisk *Przytnij*. Debuger uruchamia się ponownie, lecz tym razem zatrzyma się na wierszu, w którym ustawiliśmy punkt kontrolny (zobacz rysunek na następnej stronie).

Ustawianie punktów kontrolnych pozwala zatrzymać wykonywanie skryptu w dowolnym punkcie. Dzięki temu możemy łatwo znajdować miejsca występowania błędów i śledzić wartości zmiennych w każdej instrukcji.

Aby usunąć punkty kontrolne, umieść kursor w wierszu, z którego chcesz usunąć punkt kontrolny i kliknij przycisk *Toggle Breakpoint*. Aby usunąć wszystkie punkty kontrolne, użyj przycisku *Clear All Breakpoints* (w tym przypadku położenie kursora nie gra roli).

Krokowe wykonanie kodu

Microsoft Script Debugger umożliwia stosunkowo bezbolesne wykonywanie kodu krok po kroku. Przycisk *Step Into* (którego używaliśmy przed chwilą) pozwala użytkownikowi wykonać



następną względem kursora instrukcję skryptu. Aby to zilustrować, załadujmy ponownie do IE dokument *VBTrim.html* (z instrukcją `debugger` wciąż obecną na początku skryptu). Ponownie otworzy się debuger z podświetlonym wierszem `debugger;`.

Aby zacząć wykonywać skrypt krokowo, kliknij przycisk *Step Into*. Sterowanie wykonaniem zostaje zwrócone do IE, ponieważ skrypt czeka na dane wejściowe. Wpisz „ `s` ” (spacja, „s”, spacja) jako wartość i kliknij *Przytnij*. Debuger podświetlił teraz wywołanie funkcji, spowodowane zdarzeniem `onClick` przycisku w dokumencie HTML. Jeśli będziemy dalej klikać przycisk *Step Into*, skrypt wykona się i zwróci pożądane wyniki do przeglądarki.

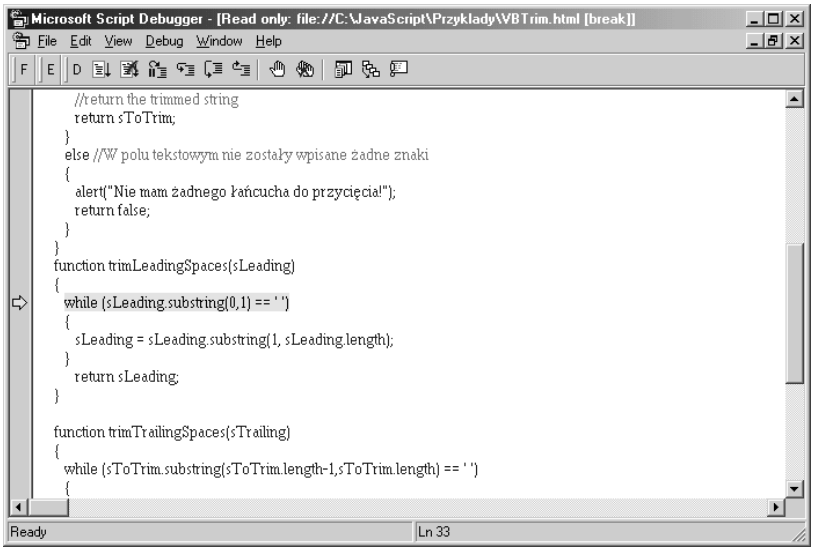
Lecz co będzie, jeśli już na początku stwierdzimy, że dana procedura nie jest powodem błędu? Czy jest jakiś sposób, żeby po prostu wykonać pozostałe instrukcje w procedurze i wrócić do następnej instrukcji? Odpowiedź brzmi „tak” — za pomocą funkcji debugera *Step Out* i *Step Over*. Aby to zademonstrować, będziemy musieli zmodyfikować dokument *VBTrim.html*. Zmienimy go tak, aby osobna funkcja obcinała spacje na początku łańcucha i osobna funkcja spacje na końcu. Wobec tego plik *VBTrim.html* będzie wyglądał następująco:

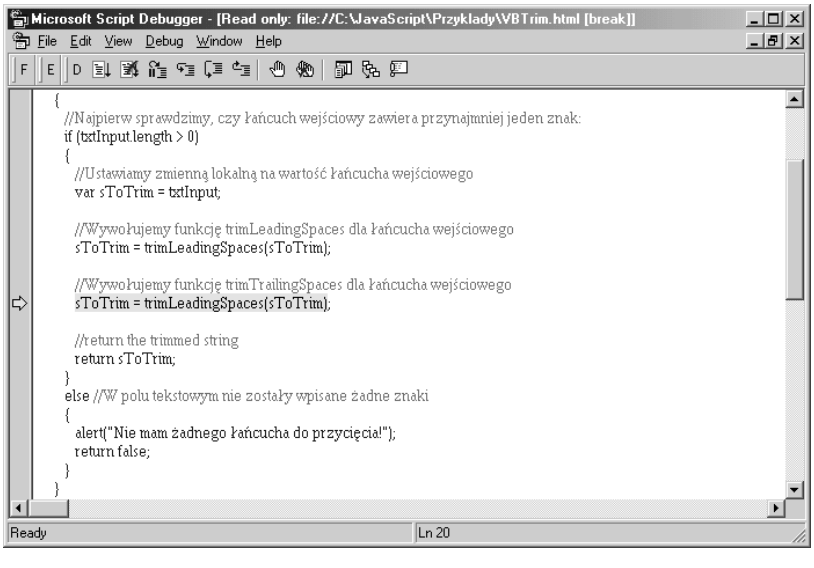
```
<!-- VBTrim.html -->
<HTML>
<HEAD>
   <TITLE>Wykorzystanie programu Microsoft Script Debugger</TITLE>

   <SCRIPT LANGUAGE="JavaScript">
   debugger;
   function VBTrim(txtInput)
   {
      //Najpierw sprawdzimy, czy łańcuch wejściowy zawiera
      //przynajmniej jeden znak:
      if (txtInput.length > 0)
      {
         //Ustawiamy zmienną lokalną na wartość łańcucha wejściowego
         var sToTrim = txtInput;

         //Wywołujemy funkcję trimLeadingSpaces dla łańcucha
```

```
            //wejściowego
            sToTrim = trimLeadingSpaces(sToTrim);

            //Wywołujemy funkcję trimTrailingSpaces dla łańcucha
            //wejściowego
            sToTrim = trimLeadingSpaces(sToTrim);

            //zwróć sktócony string
            return sToTrim;
         }
         else //W polu tekstowym nie zostały wpisane żadne znaki
         {
            alert("Nie mam żadnego łańcucha do przycięcia!");
            return false;
         }
      }
      function trimLeadingSpaces(sLeading)
      {
         while (sLeading.substring(0,1) == ' ')
         {
            sLeading = sLeading.substring(1, sLeading.length);
         }
            return sLeading;
      }

      function trimTrailingSpaces(sTrailing)
      {
         while (sToTrim.substring(sToTrim.length-1,sToTrim.length) == ' ')
         {
            sToTrim = sToTrim.substring(0, sToTrim.length-1);
         }
         return sTrailing;
      }

      </SCRIPT>
   </HEAD>
   <BODY>
      Obcinanie spacji na początku i końcu łańcucha: <br><br>
      <input type="text" name="txtValue">
      <input type="button"
      onClick="txtValue.value = VBTrim(txtValue.value)" value="Przytnij">
   </BODY>
   </HTML>
```

Otwórz zmieniony dokument w IE. Script Debugger powinien uruchomić się od razu, z instrukcją `debugger` podświetloną na żółto. Kliknij przycisk *Step Into* raz, wpisz „ `spacje` ” do pola tekstowego i kliknij przycisk *Przytnij*. Zostanie podświetlone zdarzenie `onClick`. Kliknij teraz 4 razy przycisk *Step Into*, tak by debuger przeszedł do wnętrza funkcji `trimLeadingSpaces` (zobacz pierwszy rysunek na następnej stronie).

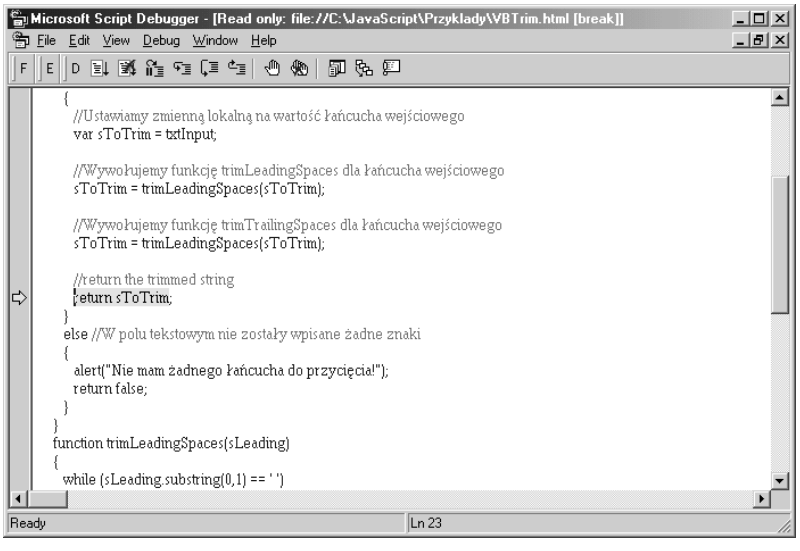
W tym punkcie debuger wyszedł z funkcji `VBTrim()` i wszedł do funkcji `trimLeadingSpaces()`. Na potrzeby dyskusji załóżmy, że ta funkcja jest w porządku. Aby wydostać się z niej, a jednak wykonać ją, użyjemy przycisku *Step Out*. Pozwoli to na dokończenie wykonywania bieżącej procedury i przejście do następnej instrukcji procedury wywołującej funkcje (`VBTrim()`). Debuger powinien teraz przejść do następnego wiersza funkcji `VBTrim()` (`trimTrailingSpaces()`) (zobacz drugi rysunek na następnej stronie).





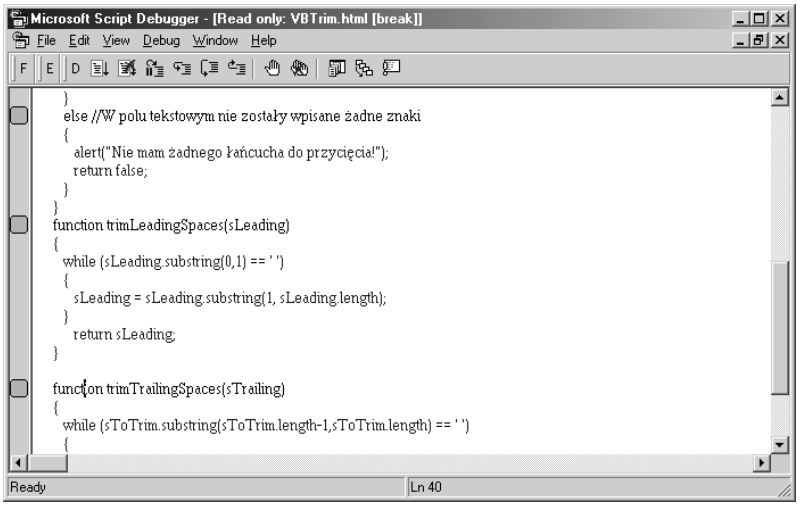
Ponownie załóżmy, że funkcja `trimTrailingSpaces` nie zawiera błędów. Wykonamy funkcję używając przycisku *Step Over*, dzięki czemu nie będzie wykonywana krokowo wiersz po wierszu. Przycisk *Step Over*, podobnie jak *Step Out*, wróci nas do następnej instrukcji procedury wywołującej (zobacz pierwszy rysunek na następnej stronie).

Przyciski *Step Out* i *Step Over* pozwalają nam zaoszczędzić czas, dzięki temu, że nie musimy przechodzić każdego wiersza funkcji wywołanej przez główną procedurę. Jeśli mamy pewność, że określona funkcja nie zawiera błędu, przeskoczymy wywołanie funkcji. Jeśli chcemy przetestować wartość wewnątrz funkcji, lecz nie całą funkcję, wejdziemy do niej aż do instrukcji, w której wartość jest ustawiana, sprawdzimy wartość za pomocą okna poleceń i wyjdziemy z funkcji.



Zakładki

Zakładki pozwalają nam łatwo wracać do określonych wierszy kodu. Gdy dojdziemy do wiersza, który chcemy oznaczyć zakładką, wystarczy nacisnąć *Ctrl+F2*. Debuger oznaczy taki wiersz jak poniżej:



Aby przejść do następnej zakładki, naciśnij *F2*. Aby przejść do poprzedniej, naciśnij *Shift+F2*. Aby usunąć zakładkę, naciśnij *Ctrl+F2*.

Ćwiczenie

Omówiliśmy główne funkcje programu Microsoft Script Debugger, więc pora zastosować tę wiedzę w praktyce. Weźmy na przykład dokument *VBTrim.html* (wersję z trzema funkcjami).

Nasi fikcyjni użytkownicy zgłosili, że strona *VBTrim.html* umieszcza w polu tekstowym słowo `false` po każdym naciśnięciu przycisku. Pierwsze, co zrobimy jako programiści, to załadujemy stronę do debugera. Następnie za pomocą okna *Command Window* wprowadzimy do funkcji wartość, która przetestuje większość funkcji, na przykład łańcuch ze spacją na początku, następnie kilkoma innymi znakami i spacją na końcu. W ten sposób przetestujemy wszystkie instrukcje poza `else`.

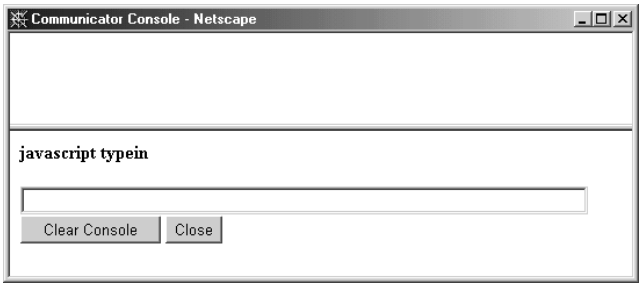
Ponieważ nie znaleźliśmy żadnej anomalii za pomocą tego łańcucha wejściowego, musimy wygenerować dane wejściowe, które pozwolą przetestować instrukcję `else` — pusty łańcuch (`txtInput.length = 0`). Widzimy teraz, że instrukcja `return` zwraca łańcuch `false` do pola tekstowego. Aby naprawić błąd, powinniśmy usunąć instrukcję `return` lub zwrócić łańcuch wejściowy.

Netscape JavaScript Console

Netscape udostępnia JavaScript Console jako narzędzie wspomagające uruchamianie JavaScriptu. Konsola wyświetla listę wszystkich błędów, wygenerowanych przez aktualnie otwarty dokument. *JavaScript Console* działa podobnie jak *Command Window* ze Script Debuggera Microsoftu: pozwala wprowadzać polecenia JavaScriptu i wyświetla wyniki.

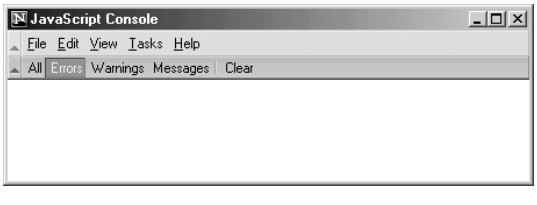
Jak otworzyć konsolę

Wpisanie `javascript:` w pasku adresu Netscape Navigatora 4.x otwiera konsolę JavaScriptu, która wygląda następująco:



Aby otworzyć konsolę JavaScriptu w Netscape 6, z menu *Tasks* wybierz *Narzędzia*/*JavaScript Console* (lub wpisz `javascript:` w pasku adresu, jak poprzednio).

Konsola JavaScriptu w Netscape 6 wygląda następująco:

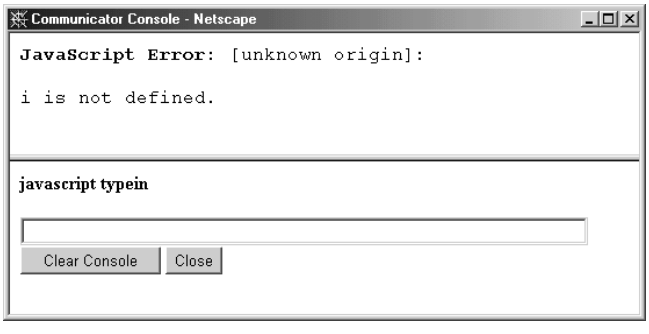


Obliczanie wartości wyrażeń za pomocą konsoli

Wprawdzie Netscape JavaScript Console przypomina okno poleceń Microsoft Script Debuggera pod tym względem, że pozwala nam obliczać pojedyncze wyrażenia JavaScriptu, lecz różni się tym, że nie pozwala na interakcję z kodem załadowanym do przeglądarki. Na przykład, załaduj poniższy kod do przeglądarki NN 4.x:

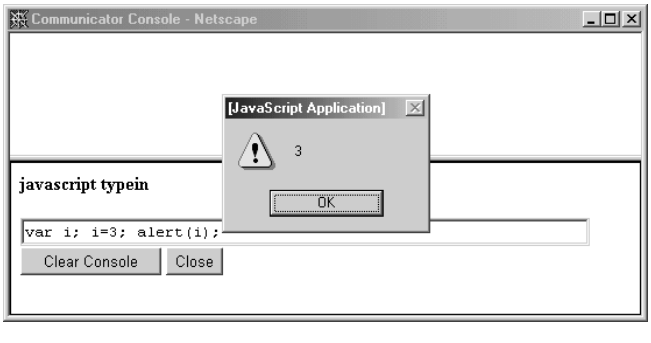
```
<!-- alerti.html -->
<HTML>
<HEAD>
   <SCRIPT>
      var i = 3;
   </SCRIPT>
</HEAD>
</HTML>
```

Następnie otwórz konsolę JavaScriptu i wpisz `alert(i)` w polu tekstowym *javascript typein*. Wygeneruje to błąd:



Powoduje go fakt, że konsola w ogóle nie daje interakcji z kodem na stronie. Aby oddziaływać wzajemnie z tym kodem, możemy użyć URL `javascript:` w pasku adresu przeglądarki, na przykład `javascript:alert(i)`.

Aby wygenerować właściwe wyniki za pomocą konsoli, musimy odtworzyć cały skrypt lub jego sekcję, niezbędną, aby otrzymać pożądany efekt.



Wywołanie konsoli przy błędach

W przypadku przeglądarek Netscape 4.06 i nowszych (z wyjątkiem wersji 6+) możemy osiągnąć za pomocą skryptów taki efekt, że JavaScript Console będzie pojawiać się za każdym razem, gdy strona natknie się na błąd. Aby zaprogramować tę funkcjonalność, musimy zmienić plik preferencji użytkownika *prefs.js*, położony w katalogu instalacyjnym Netscape w *Users*, a następnie *User Name* (lub *default*), na przykład:

```
C:\Program Files\Netscape\Users\default\prefs.js
```

Otwórz plik *prefs.js* w dowolnym edytorze tekstowym. Trzeba dodać następujący wiersz na końcu pliku:

```
user_pref("javascript.console.open_on_error",true);
```

Aby zmiany odniosły skutek, plik należy zapisać i zamknąć, a następnie ponownie uruchomić Netscape.

Funkcjonalność automatycznego otwierania konsoli w razie błędu obecnie nie jest zaimplementowana w Netscape 6.

Ustawianie punktów kontrolnych

Wprawdzie JavaScript Console nie pozwala interaktywnie przejść przez skrypt krok po kroku, lecz udostępnia **punkty kontrolne** (*watch point*), które pomagają śledzić wartości zmiennych w skrypcie. Punkty kontrolne używają do śledzenia własności obiektów metod `watch()` i `unwatch()`.

Metoda `watch()` przyjmuje dwa parametry: nazwę obserwowanej własności i nazwę funkcji, która ma zostać wykonana przy jej zmianie. Gdy podana własność ulega zmianie, wówczas metoda `watch()` przekazuje nazwę własności oraz wartości poprzednią i bieżącą do podanej funkcji.

Proszę zwrócić uwagę, że funkcja musi zwrócić bieżącą wartość własności, w przeciwnym razie własność stanie się niezdefiniowana.

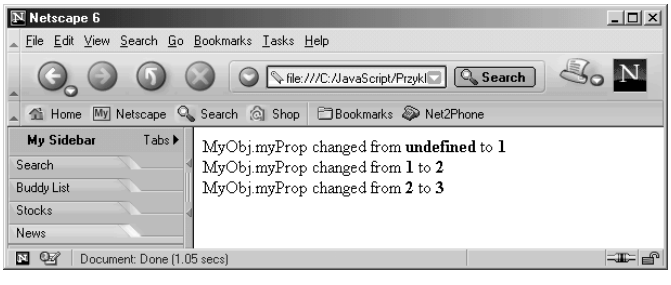
Metoda `unwatch()` przyjmuje nazwę własności jako parametr i po prostu kończy śledzenie zmian tej własności.

Poniższy skrypt demonstruje użycie metod `watch()` i `unwatch()`. Skrypt ten obserwuje wartość własności `myObj.myProp`, dopóki licznik nie osiągnie wartości `3`.

```
<!-- watchpoint.html -->
<HTML>
<HEAD>
      <SCRIPT>
   function showChange(propName,wasVal,isVal)
   {
      document.write('MyObj.' + propName + ' changed from <b>' + wasVal +
         '</b> to <b>' + isVal + '</b><br>');
      return isVal;
```

```
    }
       var myObj = new Object;

       myObj.watch('myProp',showChange);

       for (var i=1; i<=5; i++)
       {
          myObj.myProp = i
          if (i == 3)
          {
             myObj.unwatch('myProp')
          }
       }
    </SCRIPT>
  </HEAD>
</HTML>
```

Wynik działania skryptu jest następujący:



Podsumowanie

Ponieważ JavaScript jest językiem interpretowanym o luźnej kontroli typów, błędy są często zgłaszane bez szczegółowych objaśnień, co poszło nieprawidłowo. Istnieje mnóstwo sposobów sondowania zachowania danego skryptu, od prostych interaktywnych zapytań aż po pełne graficzne środowiska uruchomieniowe. Proste techniki są często wystarczające, aby wyszukać najpowszechniejsze przyczyny problemów. W przypadku pozostałych musimy zaakceptować fakt, że wygoda języka interpretowanego z jednej strony daje nam elastyczność, lecz z drugiej nakłada na autora skryptów obowiązek szczególnej staranności.

W niniejszym rozdziale omówiliśmy:

- Trzy różne typy błędów JavaScriptu: składniowe, wykonania i logiczne.
- Część powszechnie spotykanych błędów JavaScriptu.
- Korzystanie z wyjątków.
- Kilka podstawowych technik uruchomieniowych.
- Microsoft Script Debugger.
- Netscape JavaScript Console.

Po ukończeniu lektury i zastosowaniu technik omówionych w tym rozdziale Czytelnik powinien móc całkiem efektywnie usuwać błędy ze swoich skryptów. Postępowanie zgodnie ze wskazówkami omówionymi w tym i poprzednim rozdziale powinno prowadzić do popełniania mniejszej liczby błędów. Nawet jeśli nie wszystkie informacje w tym rozdziale były dla Ciebie nowością, mamy nadzieję, że posłużyły jako dobre repetytorium tematu.