

» Idź do

- Spis treści
- Przykładowy rozdział

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

ASP.NET AJAX Server Controls. Zaawansowane programowanie w nurcie .NET Framework 3.5. Microsoft .NET Development Series

Autor: Adam Calderon, Joel Rumerman

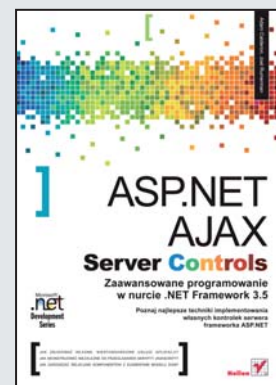
Tłumaczenie: Mikołaj Szczepaniak

ISBN: 978-83-246-2089-0

Tytuł oryginału: [Advanced ASP.NET](#)

[AJAX Server Controls For .NET Framework 3.5](#)

Format: 168x237, stron: 584



Poznaj najlepsze techniki implementowania własnych kontroltek serwera frameworka ASP.NET

- Jak skonstruować niezależne od przeglądarek skrypty JavaScript?
- Jak zbudować własne, niestandardowe usługi aplikacji?
- Jak zarządzać relacjami komponentów z elementami modelu DOM?

Kontrolki serwera pozwalają umieszczać dane dotyczące wyglądu przeglądarki i funkcjonalności serwera w spójnych obiektach wielokrotnego użytku. Można je stosować nie tylko na wielu stronach tej samej aplikacji szkieletu ASP.NET, ale także w wielu różnych aplikacjach tego frameworka. Oferuje on mnóstwo gotowych, zarówno wyjątkowo prostych, jak i złożonych kontroltek serwera. Co więcej – z jego pomocą można również tworzyć własne kontrolki, posiadające funkcjonalności, których nie zaimplementowano w kontrolkach już istniejących. Jak wykorzystać ten potencjał ASP.NET?

Książka „ASP.NET AJAX Server Controls. Zaawansowane programowanie w nurcie .NET Framework 3.5” zawiera szczegółowe wyjaśnienia i instrukcje, jak korzystać z frameworka ASP.NET AJAX w procesie tworzenia kontroltek serwera, obejmujących funkcjonalność frameworka AJAX. Dzięki temu podręcznikowi poznasz wewnętrzne mechanizmy i możliwości rozszerzania frameworka ASP.NET AJAX. Nauczysz się konstruować interaktywne kontrolki przy użyciu elementów zestawu narzędzi AJAX Control Toolkit oraz budować własne, niestandardowe usługi aplikacji.

- Programowanie w języku JavaScript
- Obsługa błędów
- Łańcuchy, zmienne i argumenty funkcji
- Programowanie biblioteki Microsoft AJAX Library
- Dziedziczenie i implementacja interfejsu
- Typy wyliczeniowe
- Kontrolki
- Obiekt Sys. Application
- Dodawanie funkcji klienckich do kontroltek serwera
- Lokalizacja we frameworku ASP.NET AJAX
- Wytwarzanie kontroltek w środowisku częściowej komunikacji zwrotnej
- Usługi aplikacji
- Architektura strony klienckiej i architektura serwera

Nie ograniczaj się – twórz i dodawaj własne funkcjonalności AJAX do kontroltek serwera!



Spis treści

<i>Słowo wstępne</i>	13
<i>Przedmowa</i>	15
<i>Podziękowania</i>	23
<i>O autorach</i>	27
I KOD KLIENTA	
1 Programowanie w języku JavaScript	31
Ogólnie o języku JavaScript	32
<i>Wprowadzenie do JavaScriptu</i>	32
<i>Atrybuty języka JavaScript</i>	32
<i>Proste typy danych</i>	34
Łańcuchy	35
<i>Obiekty</i>	36
<i>Zmienne i argumenty funkcji</i>	43
<i>Obsługa błędów</i>	51
<i>Opóźnianie wykonywania kodu za pomocą limitów i przedziałów czasowych</i>	56
Programowanie obiektowe w języku JavaScript	64
<i>Abstrakcyjne typy danych</i>	65
<i>Dziedziczenie</i>	71
Podsumowanie	75
2 Programowanie biblioteki Microsoft AJAX Library	77
Rozszerzanie wbudowanych typów języka JavaScript	78
<i>Wartości logiczne</i>	78
<i>Daty i liczby</i>	79



Łańcuchy	79
Tablice	80
Rozszerzanie biblioteki Microsoft AJAX Library	85
Klasy	85
Interfejsy	92
Typy wyliczeniowe	96
Dziedziczenie i implementacja interfejsu	101
Ważne nowe typy	111
Typ <code>Sys.EventHandlerList</code>	111
Typ <code>Sys.StringBuilder</code>	117
Obiekt <code>Sys.Debug</code>	118
Typ <code>Sys.UI.DomElement</code>	123
Typ <code>Sys.UI.DomEvent</code>	128
Zarządzanie zasięgiem	133
Delegacje	134
Wywołania zwrotne	135
Podsumowanie	137

II KONTROLKI

3 Komponenty	141
Definicja komponentów	141
<i>Komponenty, kontrolki i zachowania</i>	142
Typ <code>Sys.Component</code>	144
<i>Definiowanie nowych komponentów</i>	148
<i>Tworzenie komponentów</i>	153
<i>Podsumowanie wiedzy o komponentach</i>	168
Kontrolki	168
<i>Nowe pojęcia</i>	170
<i>Definiowanie nowej kontrolki</i>	172
<i>Tworzenie kontrolki</i>	174
<i>Podsumowanie wiedzy o kontrolkach</i>	175
Zachowania	175
<i>Definiowanie zachowania</i>	177
<i>Tworzenie zachowania</i>	178
<i>Podsumowanie wiedzy o zachowaniach</i>	183
Podsumowanie	183



4	Obiekt Sys.Application	185
	Informacje podstawowe	185
	<i>Tworzenie obiektu Sys.Application</i>	185
	<i>Informacje o typie</i>	187
	<i>Informacje o metodach</i>	188
	Menedżer komponentów	190
	<i>Dodawanie komponentu</i>	191
	<i>Odnajdywanie komponentu</i>	194
	<i>Usuwanie komponentu</i>	197
	<i>Uzyskiwanie komponentów</i>	198
	Procedura inicjalizacji	198
	<i>Proces tworzenia komponentów</i>	202
	<i>Zdarzenie load</i>	211
	Procedura zwalniania	215
	<i>Metoda Sys.Application.dispose</i>	216
	Podsumowanie	218
5	Dodawanie funkcji klienckich do kontrolek serwera	219
	Architektura generowania skryptów	220
	<i>Generowanie skryptów zachowań i kontrolek</i>	220
	<i>Zasoby skryptu</i>	225
	<i>Kontrolka ScriptManager</i>	228
	Dodawanie zachowania klienta z wykorzystaniem klasy ExtenderControl	230
	<i>Wprowadzenie do klasy ExtenderControl</i>	231
	<i>Tworzenie kontrolki rozszerzającej</i>	231
	Dodawanie funkcjonalności klienta z wykorzystaniem klasy ScriptControl	242
	<i>Przegląd klasy ScriptControl</i>	243
	<i>Tworzenie kontrolki skryptu</i>	245
	Dodawanie funkcjonalności klienta do kontrolek kompozytowych z wykorzystaniem interfejsu IScriptControl	254
	<i>Przegląd kontrolek kompozytowych</i>	254
	<i>Interfejs IScriptControl</i>	256
	<i>Tworzenie kontrolki kompozytywnej</i>	257
	Podsumowanie	261



6	Lokalizacja we frameworku ASP.NET AJAX	263
	Lokalizacja we frameworku ASP.NET	263
	<i>Określenie, które elementy wymagają lokalizacji</i>	265
	<i>Przystosowanie aplikacji do reguł określonej kultury</i>	269
	<i>Lokalizacja wyświetlanych wartości</i>	274
	Lokalizacja we frameworku ASP.NET AJAX	283
	<i>Mechanizmy lokalizacyjne języka JavaScript</i>	284
	<i>Mechanizmy lokalizacyjne ASP.NET AJAX</i>	287
	Podsumowanie	320
7	Wytwarzanie kontroltek w środowisku częściowej komunikacji zwrotnej	321
	Działanie kontrolki UpdatePanel	322
	Wpływ częściowej komunikacji zwrotnej na komponenty klienckie	327
	<i>Automatyczne zwalnianie zachowań i kontroltek</i>	332
	<i>Automatyczne zwalnianie komponentów</i>	340
	<i>Ręczne zwalnianie komponentów, kontroltek lub zachowań</i>	343
	Ładowanie wyrażeń i plików języka JavaScript	357
	<i>Metody rejestrowania skryptów w kontrolce ScriptManager</i>	357
	<i>Metoda Sys.Application.notifyScriptLoaded()</i>	363
	Zdarzenia obiektu Sys.Application	365
	<i>Zdarzenie init</i>	365
	<i>Zdarzenie load</i>	366
	Podsumowanie	368
III	KOMUNIKACJA	
8	Architektura komunikacji frameworku ASP.NET AJAX	371
	Nowy model komunikacji	372
	Architektura komunikacji frameworku ASP.NET AJAX 2.0 Extensions	374
	<i>Usługi sieciowe</i>	375
	<i>Metody stron</i>	385
	<i>Serializacja</i>	386
	<i>Komponenty frameworku stosowane po stronie serwera</i>	391
	Architektura komunikacji biblioteki Microsoft AJAX Library	398
	<i>Usługi pośredniczące</i>	398
	<i>Serializacja</i>	411



	<i>Klasa WebRequest</i>	412
	<i>Komponenty rdzenia żądania sieciowego</i>	417
	Podsumowanie	419
9	Usługi aplikacji	421
	Usługi członkostwa, ról i profilów użytkowników frameworku ASP.NET 2.0	421
	<i>Uwierzytelnianie formularzy</i>	422
	<i>ASP.NET 2.0 Provider Model</i>	425
	<i>Narzędzie Web Site Administration Tool</i>	427
	<i>Członkostwo</i>	427
	<i>Role</i>	432
	<i>Profile</i>	438
	Usługi aplikacji frameworku ASP.NET AJAX	441
	<i>Usługa uwierzytelniania</i>	441
	<i>Usługa ról</i>	446
	<i>Usługa profili</i>	448
	Niestandardowe usługi aplikacji	451
	<i>Klasa ServiceHandlerFactory protokołu HTTP i klasy pomocnicze</i>	454
	<i>Pośrednik w dostępie do usługi</i>	463
	<i>Konfiguracja</i>	467
	Podsumowanie	468
IV	ZESTAW NARZĘDZI AJAX CONTROL TOOLKIT	
10	Architektura zestawu narzędzi ASP.NET AJAX Control Toolkit	471
	Przegląd zestawu narzędzi ASP.NET AJAX Control Toolkit	471
	<i>Atrybuty jako sposób na uproszczenie programowania</i>	472
	<i>Bogaty zbiór klas platformy .NET</i>	472
	<i>Bogaty zbiór klas języka JavaScript</i>	473
	<i>Obsługa animacji</i>	473
	Struktura zestawu narzędzi ASP.NET AJAX Control Toolkit	473
	<i>Instalacja</i>	473
	<i>Układ pliku rozwiązania</i>	474
	Architektura serwera	475
	<i>Atrybuty</i>	476
	<i>Klasy bazowe dla kontroltek rozszerzających i kontroltek skryptowych</i>	480
	<i>Klasy projektowe</i>	485



Architektura strony klienckiej	488
<i>Klasa BehaviorBase</i>	489
<i>Klasa ControlBase</i>	490
Animacje	490
<i>Struktura i rodzaje animacji</i>	490
<i>Architektura klienta</i>	492
<i>Architektura serwera</i>	496
Podsumowanie	500
11 Dodawanie funkcji klienckich do kontroltek serwera z wykorzystaniem zestawu narzędzi ASP.NET AJAX Control Toolkit	501
Dodawanie zachowań strony klienckiej za pomocą klasy ExtenderControlBase	501
<i>Szablon biblioteki kontrolki rozszerzającej środowiska Visual Studio 2008</i>	502
<i>Dziedziczenie po klasie bazowej ExtenderControlBase</i>	506
<i>Tworzenie klasy AjaxControlToolkit.BehaviorBase</i>	508
<i>Dołączanie kontrolki rozszerzającej do kontrolki docelowej</i>	510
<i>Wnioski końcowe</i>	511
Dodanie mechanizmów obsługi trybu projektowania do naszej kontrolki rozszerzającej	511
<i>Domyślne funkcje trybu projektowania</i>	511
<i>Dodawanie klas projektujących i edytorów do właściwości</i>	512
Dodawanie animacji do naszej kontrolki rozszerzającej	518
<i>Animacje tworzone z wykorzystaniem interfejsu API JavaScriptu</i>	518
<i>Animacje tworzone z wykorzystaniem modelu deklaratywnego</i>	522
Podsumowanie	528
DODATKI	
A JavaScript w środowisku Visual Studio	531
Technologia IntelliSense	531
<i>Odwołania do bibliotek i usług sieciowych</i>	531
Komentarze języka XML	535
B Weryfikacja parametrów metod	539



C	Obiekty obsługujące i moduły frameworku ASP.NET	543
	Cykl życia aplikacji ASP.NET	543
	Obiekty obsługujące żądania protokołu HTTP	544
	<i>Przegląd obiektów obsługujących protokołu HTTP</i>	544
	<i>Przegląd fabryki obiektów obsługujących protokołu HTTP</i>	546
	Moduły protokołu HTTP	548
	<i>Przegląd modułów protokołu HTTP</i>	548
D	Kod obsługujący błędy klientów	553
	Klasa kliencka ErrorHandler	553
	Klasa kliencka EventArgs	555
	ErrorHandler Server Control	555
	Klasa kliencka StackTrace	556
	Usługa sieciowa ErrorDataService	557
	Strona testująca mechanizm obsługi błędów	558
	Skorowidz	559

3

Komponenty

W ROZDZIALE 2., ZATYTULOWANYM „Programowanie biblioteki Microsoft AJAX Library” rozpoczęliśmy omawianie biblioteki Microsoft AJAX Library i sposobów, w jakie jej twórcy poszerzyli wbudowane typy JavaScriptu o nowe funkcje, technik wykorzystywania modelu prototypowego do rozszerzania samej biblioteki Microsoft AJAX Library o nasze typy niestandardowe, a nawet kilku ważnych aspektów działania typów wbudowanych.

W tym rozdziale będziemy kontynuować omawianie biblioteki Microsoft AJAX Library — tym razem jednak skoncentrujemy się na komponentach i ich typach potomnych: kontrolkach i zachowaniach. W niniejszym rozdziale przystąpimy też do tworzenia obiektów klienckich związanych z kontrolkami serwera.

Definicja komponentów

Komponentem jest każdy obiekt, którego typ kliencki dziedziczy po typie `Sys.Component`. Wspomniany typ bazowy `Sys.Component` jest bardzo ważny, ponieważ właśnie za jego pomocą będziemy rozszerzać ten framework o nowe komponenty. Tworzenie nowych komponentów z wykorzystaniem tego typu jest o tyle uzasadnione, że właśnie typ `Sys.Component` ma kilka specyficznych cech, których nie oferuje żaden inny typ biblioteki Microsoft AJAX Library.

Po pierwsze, komponenty są projektowane z myślą o zapełnianiu luki dzielącej oprogramowanie klientów i serwerów. Za pośrednictwem obiektów serwera nazywanych deskryptorami skryptów (ang. *script descriptors*) możemy wymuszać na frameworku ASP.NET AJAX automatyczne generowanie kodu JavaScriptu odpowiedzialnego za tworzenie egzemplarzy naszych typów komponentów. W ten sposób możemy wiązać mechanizmy oprogramowania klienckiego z kontrolkami naszego serwera WWW bez konieczności umieszczania jakichkolwiek skryptów języka JavaScript w klasach tych kontrollek.

■ UWAGA Tworzenie komponentów za pośrednictwem kodu serwera

Techniki tworzenia komponentów za pośrednictwem kodu serwera zostaną szczegółowo omówione w rozdziale 5., zatytułowanym „Dodawanie funkcji klienckich do kontrolki serwera”.

Po drugie, `Sys.Application`, czyli obiekt globalny występujący w roli swego środowiska wykonawczego klienta, jest tworzony w sposób umożliwiający skuteczne zarządzanie wszystkimi typami dziedziczącymi po typie `Sys.Component`. Oznacza to, że cykl życia naszych komponentów ma charakter predefiniowany. Nie tylko będziemy na bieżąco informowani o tworzeniu i niszczeniu tych komponentów, ale też będziemy mogli — w razie konieczności — wykonywać własny, niestandardowy kod przy okazji każdego z tych zdarzeń. Takie rozwiązanie gwarantuje nam pełną kontrolę i wysoki poziom bezpieczeństwa.

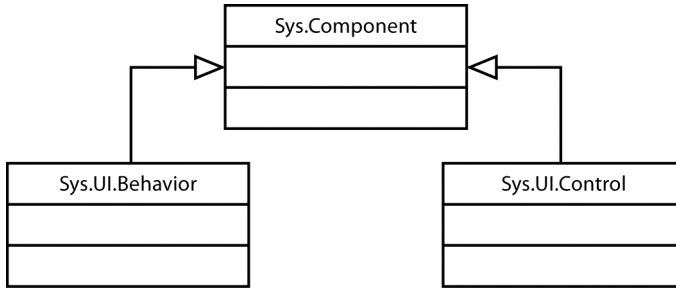
■ UWAGA Komponenty i kontrolki serwera WWW

Sposób zarządzania komponentami przez obiekt `Sys.Application` przypomina techniki zarządzania kontrolkami serwera WWW przez stronę. Ta zbieżność nie jest przypadkowa — zaprojektowano ten mechanizm w taki sposób, aby biblioteka Microsoft AJAX Library tworzyła możliwie przyjazne środowisko dla programistów ASP.NET. Obiekt `Sys.Application` zostanie szczegółowo omówiony w rozdziale 4., zatytułowanym „Obiekt `Sys.Application`”.

I wreszcie komponenty oferują wiele wbudowanych popularnych elementów funkcjonalności, których będziesz potrzebował podczas tworzenia swoich aplikacji. Dysponują na przykład egzemplarzem typu `Sys.EventHandlerList`, dzięki któremu możesz tworzyć, obsługiwać i generować niestandardowe zdarzenia. Komponenty implementują też interfejs `Sys.INotifyPropertyChanged` z metodami powiadomień o zmianach właściwości. Co więcej, komponenty implementują interfejs `Sys.INotifyDisposing`, dzięki czemu pozostałe obiekty można łatwo powiadamiać o zwalnianiu egzemplarzy komponentów.

Komponenty, kontrolki i zachowania

Jakby tego było mało (jakby te komponenty nie wystarczyły), biblioteka Microsoft AJAX Library dodatkowo oferuje dwa wyspecjalizowane typy komponentów: zachowania (ang. *behaviors*), reprezentowane przez klasę `Sys.UI.Behavior`, i kontrolki (ang. *controls*), reprezentowane przez klasę `Sys.UI.Control`. Hierarchię łączącą te trzy typy przedstawiono na rysunku 3.1.



RYСУNEK 3.1. Hierarchia klas łącząca typy `Sys.Component`, `Sys.UI.Behavior` i `Sys.UI.Control`

■ WSKAZÓWKA Komponenty zarządzane

Jak widać na rysunku 3.1, `Sys.Component` jest typem bazowym zarówno dla typu `Sys.UI.Control`, jak i dla typu `Sys.UI.Behavior`. We wcześniejszej części tego rozdziału wspomniano już, że za zarządzanie komponentami odpowiada obiekt `Sys.Application` — dzięki opisywanej relacji dziedziczenia ten sam obiekt zarządza także kontrolkami i zachowaniami. Kiedy mówimy o obiekcie `Sys.Application`, mamy na myśli strukturę zarządzającą komponentami, czyli zachowaniami, kontrolkami lub komponentami jako takimi.

W praktyce zachowania, kontrolki i komponenty są niemal identyczne. Ich podobieństwo wynika z tego, że zachowania i kontrolki dziedziczą zdecydowaną większość swojej funkcjonalności po typie bazowym, czyli po klasie komponentów, nie dodając wiele od siebie. Co więcej, funkcjonalność implementowana przez te dwa typy potomne nie wprowadza żadnych istotnych zmian.

Jedną z najważniejszych różnic dzielących klasę bazową komponentów od zachowań i kontrolki jest wbudowany w te dwa typy potomne związek z elementem modelu DOM. Twórcy tych klas zdecydowali się na dołączenie tego związku, ponieważ zachowania i kontrolki w założeniu mają mieć charakter wizualny. Z drugiej strony komponenty nie dysponują wbudowanymi związkami z elementami DOM, które nie muszą mieć postaci konstrukcji wizualnych.

Najważniejszą różnicą dzielącą zachowania i kontrolki jest to, że z pojedynczym elementem modelu DOM można związać tylko jedną kontrolkę; ale wiele zachowań.

W tabeli 3.1 podsumowano różnice dzielące wszystkie trzy typy.

Opisane w tabeli reguły są wymuszane w trakcie tworzenia komponentów, zachowań i kontrolki. Właśnie na podstawie tych cech powinniśmy podejmować decyzje o wyborze właściwego typu bazowego dla naszych nowych typów potomnych. Prosty diagram ilustrujący proces oceny tych właściwości podczas wyboru typu, po którym powinien dziedziczyć nowy typ potomny, pokazano na rysunku 3.2.

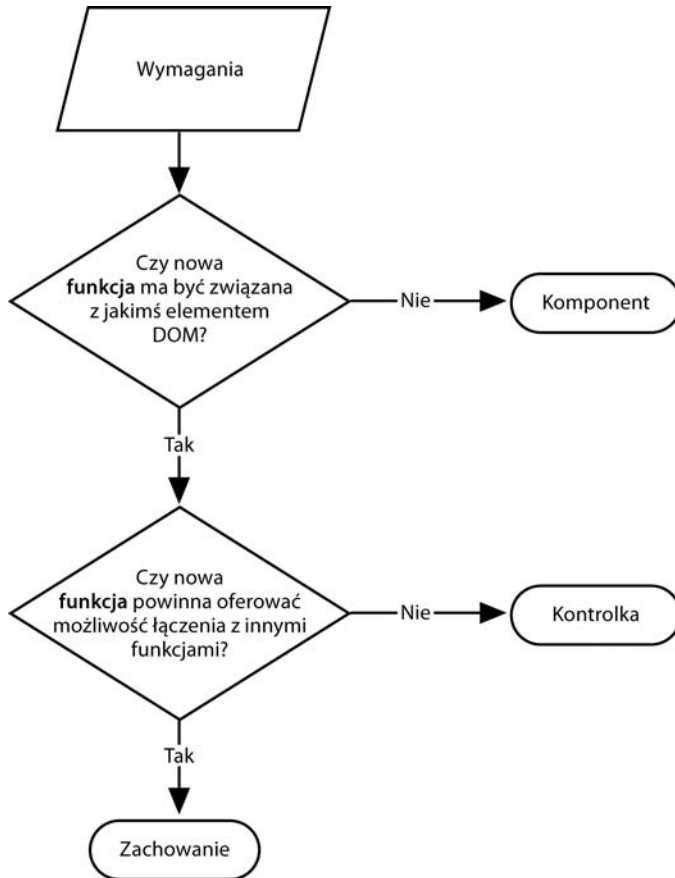
TABELA 3.1. Różnice pomiędzy komponentami, kontrolkami i zachowaniami

Typ obiektu	Możliwość związków z elementem DOM	Możliwość związków większej liczby tego rodzaju obiektów z elementem DOM	Dostępność obiektu z poziomu związanego elementu DOM
Komponent	Brak możliwości	Brak możliwości (komponent nie może być bezpośrednio związany z elementem DOM)	Brak możliwości (komponent nie może być bezpośrednio związany z elementem DOM)
Kontrolka	Musi być związany z jakimś elementem modelu DOM.	Nie, pojedynczy element DOM może być związany z tylko jedną kontrolką.	Tak, dostęp do kontrolki można uzyskać za pośrednictwem właściwości <code>expando</code> nazwanej <code>control</code> i dołączonej do elementu DOM.
Zachowanie	Musi być związany z jakimś elementem modelu DOM.	Tak, pojedynczy element DOM może być związany z jedną lub wieloma zachowaniami.	Tak, dostęp do zachowania jest możliwy za pośrednictwem właściwości <code>expando</code> nazwanej tak jak to zachowanie (pod warunkiem że to zachowanie było nazwane w czasie inicjalizacji elementu DOM). Wszystkie zachowania związane z elementem DOM są dostępne za pośrednictwem prywatnej tablicy <code>_behaviors</code> dołączonej do tego elementu.

Skoro opanowaliśmy już podstawy komponentów, kontrolerek i zachowań, możemy przystąpić do szczegółowego omówienia każdego z tych typów.

Typ `Sys.Component`

`Sys.Component` jest typem bazowym dla wszystkich komponentów i jako taki oferuje zdecydowaną większość ich elementów funkcjonalności. Typ `Sys.Component` nie dziedziczy po żadnym innym typie, ale implementuje trzy interfejsy: `Sys.IDisposable`, `Sys.INotifyPropertyChanged` oraz `Sys.INotifyDisposing`. Wszystkie te interfejsy krótko opisano w tabeli 3.2.



RYSUNEK 3.2. Proces decyzyjny dla wyboru komponentu, kontrolki lub zachowania

TABELA 3.2. Interfejsy implementowane przez typ Sys.Component

Interfejs	Zadanie	Metody
Sys.INotifyPropertyChanged	Implementuje mechanizm powiadomień o zmianach wartości właściwości.	add_propertyChanged remove_propertyChanged
Sys.INotifyDisposing	Implementuje zdarzenie zwalniania komponentu.	add_disposing remove_disposing
Sys.IDisposable	Reprezentuje obiekt przeznaczony do zwolnienia.	dispose

Klasa Sys.Component zawiera też pięć wewnętrznych składowych, które opisano w tabeli 3.3.

TABELA 3.3. Składowe typu `Sys.Component`

Nazwa składowej	Zadanie	Typ
<code>_id</code>	Reprezentuje unikatowy identyfikator danego komponentu. Składowa <code>_id</code> jest wykorzystywana do odnajdywania komponentów zarejestrowanych w obiekcie <code>Sys.Application</code> . Każdy komponent zarządzany przez ten obiekt musi mieć unikatowy identyfikator.	<code>string</code>
<code>_idSet</code>	Określa, czy prawidłowo ustawiono właściwość <code>_id</code> .	<code>boolean</code>
<code>_initializing</code>	Określa, czy dany komponent jest w trakcie procesu inicjalizacji.	<code>boolean</code>
<code>_updating</code>	Określa, czy dany komponent jest aktualizowany.	<code>boolean</code>
<code>_events</code>	Reprezentuje listę zdarzeń i metod obsługujących.	<code>Sys.Event</code> ↳ <code>HandlerList</code>

Oprócz implementowania metod wymaganych przez trzy interfejsy wymienione w tabeli 3.2, klasa `Sys.Component` udostępnia kilka metod dodatkowych, które umożliwiają nam operowanie na jej wewnętrznych składowych. W tabeli 3.4 szczegółowo opisano zarówno te metody dodatkowe, jak i metody wymagane przez wspomniane trzy interfejsy.

TABELA 3.4. Metody klasy `Sys.Component`

Nazwa metody	Opis	Składnia
<code>beginUpdate</code>	Oznacza dany komponent jako aktualizowany. Metoda <code>beginUpdate</code> jest wywoływana w czasie tworzenia komponentu.	<code>comp.beginUpdate();</code>
<code>endUpdate</code>	Oznacza dany komponent jako nieaktualizowany. Metoda <code>endUpdate</code> , która jest wywoływana w czasie tworzenia komponentu, wykonuje metody <code>initialize</code> (jeśli dany komponent nie jest inicjalizowany) i <code>updated</code> .	<code>comp.endUpdate();</code>
<code>updated</code>	Implementacja pusta.	<code>comp.updated();</code>
<code>get_isUpdating</code>	Zwraca wartość składowej <code>_updating</code> .	<code>comp.get_isUpdating();</code>
<code>initialize</code>	Oznacza dany komponent jako inicjalizowany.	<code>comp.initialize();</code>
<code>get_initialized</code>	Zwraca wartość składowej <code>_initialized</code> .	<code>comp.get_initialized();</code>

TABELA 3.4. Metody klasy Sys.Component — *ciąg dalszy*

Nazwa metody	Opis	Składnia
dispose	Wykonuje metody obsługujące zdarzenie disposing. Usuwa właściwość <code>_events</code> z danego komponentu i anuluje jego rejestrację w obiekcie <code>Sys.Application</code> .	<code>comp.dispose();</code>
get_events	Zwraca wartość składowej <code>_events</code> .	<code>comp.get_events();</code>
get_id	Zwraca wartość składowej <code>_id</code> .	<code>comp.get_id();</code>
set_id	Ustawia wartość składowej <code>_id</code> . Raz ustawionego identyfikatora nie można zmieniać (za pośrednictwem tej metody); identyfikator nie może być zmieniany także po rejestracji komponentu w obiekcie <code>Sys.Application</code> .	<code>comp.set_id(id);</code>
add_disposing	Dodaje metodę obsługującą zdarzenie disposing.	<code>comp.add_disposing(handler);</code>
remove_disposing	Usuwa metodę obsługującą zdarzenie disposing.	<code>comp.remove_disposing(handler);</code>
add_propertyChanged	Dodaje metodę obsługującą zdarzenie propertyChanged.	<code>comp.add_propertyChanged(handler);</code>
remove_propertyChanged	Usuwa metodę obsługującą zdarzenie propertyChanged.	<code>comp.remove_propertyChanged(handler);</code>
raisePropertyChanged	Wykonuje zarejestrowane metody obsługujące zdarzenie propertyChanged, przekazując na ich wejściu (w formie argumentów zdarzenia) nazwę zmodyfikowanej właściwości.	<code>comp.raisePropertyChanged(propertyName);</code>

■ WSKAZÓWKA Metody beginUpdate, endUpdate oraz initialize

Metody `beginUpdate`, `endUpdate` oraz `initialize` są wykonywane automatycznie w ramach procesu tworzenia komponentu. Wymienione metody z reguły nie są wywoływane z poziomu kodu zdefiniowanego przez użytkownika, ale można je przykryć, aby umieścić w nich jakąś niestandardową funkcjonalność.

Definiowanie nowych komponentów

Klasa `Sys.Component` jest niezwykle przydatna, jednak jej celem nie jest bezpośrednio tworzenie egzemplarzy tego typu. Przeciwnie, twórcy klasy `Sys.Component` chcieli opracować typ, który będzie wykorzystywany w roli klasy bazowej dla komponentów definiowanych przez użytkowników.

Nowy typ komponentu możemy zdefiniować, stosując model prototypowy omówiony w rozdziale 2. i rejestrując nasz komponent jako typ dziedziczący po klasie bazowej `Sys.Component`.

Komponent `ErrorHandler`

Aby zademonstrować technikę definiowania nowych komponentów, utworzymy nowy, własny komponent obsługi błędów. Komponent `ErrorHandler` będzie odpowiadał za publikowanie obsługiwanych i nieobsługiwanych błędów z wykorzystaniem specjalnej usługi danych o błędach.

Szkielet

Na początek utworzymy szkielet naszego nowego komponentu (patrz listing 3.1).

LISTING 3.1. Definiowanie komponentu `ErrorHandler`

```
/// <reference name="MicrosoftAjax.js"/>
ErrorHandler = function() {
    ErrorHandler.initializeBase(this);
};

ErrorHandler.prototype = {
    initialize: function() {
        ErrorHandler.callBaseMethod(this, 'initialize');
    },
    dispose: function() {
        ErrorHandler.callBaseMethod(this, 'dispose');
    }
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component);
```

Oprócz wywołania metody `initializeBase` w konstruktorze naszego typu i rejestracji tej klasy jako typu dziedziczącego po klasie `Sys.Component` zdecydowaliśmy się na przykrycie metod `initialize` i `dispose` klasy bazowej `Sys.Component`. Umieściliśmy te przykryte metody w szkielecie naszego komponentu, ponieważ właśnie przykrywanie `initialize` i `dispose` jest zwykle jednym z pierwszych kroków procesu tworzenia komponentu (podobną kolejność sugerujemy także Tobie).

Inicjalizacja i niszczenie komponentu

Naszą szkieletową definicję należy jeszcze uzupełnić o implementacje metod `initialize` i `dispose`.

W metodzie `initialize` konstruujemy nasz komponent. Proces konstruowania komponentu obejmuje takie kroki jak dodanie metod obsługujących zdarzenia do elementów modelu DOM, dołączenie nowego elementu DOM do drzewa i wszystkie inne operacje wymagane przez konkretny typ komponentów.

W metodzie `dispose` powinniśmy umieścić kod odpowiedzialny za zwalnianie naszego komponentu. Zwalnianie komponentu może obejmować odłączenie zdarzeń od elementu modelu DOM, zniszczenie ewentualnych utworzonych elementów DOM lub zwolnienie wszelkich innych zasobów utworzonych przez nasz komponent.

■ WSKAZÓWKA Metodę `dispose` można wywołać więcej niż raz

Warto tak implementować metodę `dispose`, aby można ją było wywoływać więcej niż raz bez ryzyka powodowania błędów czasu wykonywania. W przypadku dość złożonych aplikacji nietrudno o sytuację, w której zwalniany obiekt menedżera wywoła metodę `dispose` wszystkich swoich komponentów potomnych. Okazuje się jednak, że każdy z tych obiektów potomnych jest zarejestrowany także w obiekcie `Sys.Application` jako obiekt, który może być zwalniany (który implementuje interfejs `Sys.IDisposable`). Zwalniany obiekt `Sys.Application` automatycznie wykonuje metodę `dispose` każdego z tak zarejestrowanych obiektów — oznacza to, że metody `dispose` tych obiektów są wykonywane (co najmniej) dwukrotnie. Jeśli nie zachowamy należytej ostrożności, próby wielokrotnego wykonania metody `dispose` będą się kończyły błędami czasu wykonywania. Można jednak uniknąć tych problemów, stosując proste mechanizmy weryfikacji w formie wyrażeń warunkowych.

Na potrzeby naszego nowego komponentu `ErrorHandler` należy dodać metodę obsługującą zdarzenie `error` obiektu `window` przy okazji inicjalizacji tego komponentu i usunąć tę metodę przy okazji jego zwalniania. Proponowaną implementację tego mechanizmu przedstawiono na listingu 3.2.

LISTING 3.2. Dodawanie metody obsługującej zdarzenie `error` obiektu okna

```
/// <reference name="MicrosoftAjax.js"/>
ErrorHandler = function () {
    ErrorHandler.initializeBase(this);
};

ErrorHandler.prototype = {
    initialize: function () {
        ErrorHandler.callBaseMethod(this, 'initialize');
        window.onerror =
            Function.createDelegate(this, this._unhandledError);
    }
};
```

```
    },  
  
    dispose: function ErrorHandler$dispose() {  
        window.onerror = null;  
        ErrorHandler.callBaseMethod(this, 'dispose');  
    },  
  
    _unhandledError: function(msg, url, lineNumber) {  
        try {  
            var stackTrace = StackTrace.createStackTrace(arguments.callee);  
            ErrorDataService.PublishError  
                (stackTrace, msg, url, lineNumber);  
        }  
        catch (e) { }  
    }  
}  
ErrorHandler.registerClass('ErrorHandler', Sys.Component);
```

Jak widać na listingu 3.2, wykorzystaliśmy w naszym kodzie kilka interesujących zabiegów. Po pierwsze, w metodzie `initialize` utworzono delegację wskazującą na metodę `_unhandledError` i skojarzono tę delegację ze zdarzeniem `error` obiektu `window` (przypisując ją zdarzeniu `onerror`).

■ WSKAZÓWKA Zdarzenie `window.onerror`

W omawianym kodzie wykorzystano konstrukcję polegającą na przypisaniu delegacji zdarzeniu `onerror` (zamiast metody `$addHandler`), ponieważ z jakiegoś powodu zdarzenie `error` obiektu okna nie obsługuje dodawania zdarzeń za pomocą metod `addEventListener` ani `attachEvent` (czyli dwóch metod właściwych dla konkretnych przeglądarek i ostatecznie wywoływanych przez metodę `$addHandler`).

W metodzie `dispose` usunęliśmy metodę obsługującą zdarzenie `error` obiektu `window`. Dysponujemy więc gotowym kodem inicjalizującym i zwalnającym nasz komponent.

W metodzie `unhandledError`, która będzie wykonywana w odpowiedzi na wystąpienie nieobsługiwanych błędów, podejmujemy dwa działania. Po pierwsze, generujemy ślad stosu, korzystając z globalnego obiektu `StackTrace` przekazywanego za pośrednictwem właściwości `callee` zmiennej `arguments` naszej funkcji. Po uzyskaniu obiektu śladu stosu wykonujemy metodę `PublishError` pośrednika naszej usługi sieciowej `ErrorDataService` (za pośrednictwem tej metody przekazujemy na serwer ślad stosu, komunikat o błędzie, adres URL strony, na której ten błąd wystąpił, i numer wiersza kodu). Cały ten kod dodatkowo umieściliśmy w ramach wyrażenia `try-catch`, ponieważ nie chcemy, aby kod obsługujący błędy sam generował jakiegokolwiek błędów czasu wykonywania.

■ WSKAZÓWKA Obiekty StackTrace i ErrorDataService

Globalny obiekt StackTrace, który wykorzystano do wygenerowania naszego śladu stosu wykonywania, jest niezwykle przydatny w procesie diagnozowania kodu, a jego kompletny kod źródłowy przedstawiono w dodatku D, zatytułowanym „Kod obsługujący błędy klientów”. Podobnie, kod usługi sieciowej ErrorDataService, której użyto do odesłania informacji o błędzie na serwer, także można znaleźć we wspomnianym dodatku.

Stosowanie metod i obiektów klasy bazowej

Skoro nasz typ dziedziczy po klasie Sys.Component, z natury rzeczy przejmując wszystkie atrybuty i zachowania tego typu bazowego. Korzystając z obiektu Sys.EventHandlerList dostępnego w ramach tej klasy (i jego funkcjonalności), możemy definiować nowe zdarzenia bez konieczności samodzielnego pisania sporej ilości kodu. Na listingu 3.3 rozszerzono nasz komponent ErrorHandler o zdarzenie, które możemy zarejestrować jako generowane w odpowiedzi na wystąpienia błędów.

LISTING 3.3. Korzystanie z metod klasy bazowej

```
... // wcześniejszy kod komponentu pozostaje niezmienny

_unhandledError: function (msg, url, lineNumber) {
  try {
    var stackTrace =
      StackTrace.createStackTrace(arguments.callee);
    ErrorDataService.PublishError
      (stackTrace, msg, url, lineNumber);

    var args = new ErrorEventArgs(stackTrace, msg, url, lineNumber);
    this._raiseUnhandledErrorOccurred(args);
  }
  catch (e) { }
},

add_unhandledErrorOccurred: function(handler) {
  this.get_events().addHandler("unhandledErrorOccurred", handler);
},

remove_unhandledErrorOccurred: function(handler) {
  this.get_events().removeHandler("unhandledErrorOccurred", handler);
},

_raiseUnhandledErrorOccurred: function(args) {
  var evt = this.get_events().getHandler("unhandledErrorOccurred");
  if (evt !== null) {
    evt(this, args);
  }
}
```

```

    }
  },
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component) ;

ErrorEventArgs = function(stackTrace, message, url, lineNumber) {
  ErrorEventArgs.initializeBase(this);
  this._message = message;
  this._stackTrace = stackTrace;
  this._url = url;
  this._lineNumber = lineNumber;
}
ErrorEventArgs.registerClass("ErrorEventArgs", Sys.EventArgs);

```

Skoncentrujmy się najpierw na końcowej części listingu 3.3, gdzie zdefiniowano nowy typ `ErrorEventArgs`. Typ `ErrorEventArgs` dziedziczy po typie `Sys.EventArgs` i przekształca nasze informacje o błędzie w obiekt.

W typie `ErrorHandler` dodano trzy metody niezbędne do dodawania, usuwania i generowania zdarzenia `unhandledErrorOccurred`. W naszym kodzie wykorzystujemy listę metod obsługujących utrzymywaną przez obiekt `Sys.Component` — dostęp do listy samych zdarzeń uzyskujemy za pośrednictwem metody `this.get_events()`.

I wreszcie w metodzie `_unhandledError` dodaliśmy kod tworzący argumenty zdarzenia błędu i przekazujący je do metody, która to zdarzenie generuje.

Ostatnia zmiana, którą musimy wprowadzić do komponentu `ErrorHandler`, polega na dodaniu właściwości umożliwiającej włączanie i wyłączanie trybu publikowania błędów. Niezbędne modyfikacje przedstawiono na listingu 3.4.

LISTING 3.4. Dodanie właściwości wyłączającej tryb publikacji błędów

```

/// <reference name="MicrosoftAjax.js"/>
ErrorHandler = function () {
  ErrorHandler.initializeBase(this);
  this._disableErrorPublication = false;
};

ErrorHandler.prototype = {
  ...

  get_disableErrorPublication: function() {
    return this._disableErrorPublication;
  },

  set_disableErrorPublication: function(value) {
    if (!this.get_updating()) {
      this.raisePropertyChanged("disableErrorPublication");
    }
    this._disableErrorPublication = value;
  },
};

```



```
_unhandledError: function(msg, url, lineNumber) {
  try {
    var stackTrace = StackTrace.createStackTrace(arguments.callee);
    if (!this._disableErrorPublication) {
      ErrorDataService.PublishError
        (stackTrace, msg, url, lineNumber);
    }
    var args =
      new ErrorEventArgs(stackTrace, msg, url, lineNumber);
    this._raiseUnhandledErrorOccured(args);
  }
  catch (e) { }
},
...
}
ErrorHandler.registerClass('ErrorHandler', Sys.Component);
...
```

Ostatnia zmiana czyni z naszego typu przydatny komponent, który można z powodzeniem wykorzystywać do wysyłania na serwer informacji o błędach występujących po stronie klienta i — tym samym — powiadamiania programistów o ewentualnych problemach.

Tworzenie komponentów

Na podstawie materiału zaprezentowanego w tym rozdziale do tego momentu można by pomyśleć, że tworzenie nowych komponentów polega na konstruowaniu ich obiektów i przypisywaniu im odpowiednim zmiennym (patrz listing 3.5).

LISTING 3.5. Tworzenie egzemplarza komponentu za pomocą słowa `new`

```
var errorHandler = new ErrorHandler();
errorHandler.set_disableErrorPublication (false);
```

Chociaż w kodzie w tej formie nie ma niczego nieprawidłowego (w końcu egzemplarz komponentu jest właśnie obiektem JavaScriptu), komponenty należy tworzyć raczej za pośrednictwem metody `Sys.Component.create`. Składnię wywołań metody `create` przedstawiono na listingu 3.6.

LISTING 3.6. Tworzenie egzemplarza komponentu za pośrednictwem metody `Sys.Component.create`

```
var newComponent =
  Sys.Component.create(
    type,
    properties,
    events,
    references,
    element);
```

Działanie metody `Sys.Component.create`, która jest dostępna także za pośrednictwem zmiennej globalnej `$create`, nie ogranicza się tylko do tworzenia nowego egzemplarza określonego typu. Oprócz tworzenia nowego egzemplarza wskazanego typu metoda `Sys.Component.create` rejestruje ten egzemplarz w obiekcie `Sys.Application` jako komponent zarządzany, po czym automatycznie wywołuje metody `beginUpdate`, `endUpdate`, `updating` i `initialize` tego komponentu. Wszystkie te działania są podejmowane automatycznie w zależności od parametrów użytych w jej wywołaniu. Co więcej, metoda `$create` może przypisywać wartości początkowe właściwości, dodawać metody obsługujące zdarzenia, przypisywać inne komponenty w formie referencji i wiązać z danym komponentem element modelu DOM. I wreszcie metoda `$create` zwraca wskaźnik do nowo utworzonego egzemplarza. Jak widać, metoda `$create` robi dużo więcej niż tylko tworzenie nowego egzemplarza naszego typu.

■ UWAGA Wykonywanie metody `initialize`

Metoda `initialize` zawsze jest wykonywana po ustawieniu wszystkich właściwości, zdarzeń i referencji.

Co ważne, metoda `$create` tworzy nie tylko egzemplarze typów, które bezpośrednio dziedziczą po klasie `Sys.Component`, ale też egzemplarze typów na wielu poziomach dziedziczenia pomiędzy wskazanym typem a typem `Sys.Component`. Zbiór tych egzemplarzy obejmuje kontrolki dziedziczące po typie `Sys.UI.Control` i zachowania dziedziczące po typie `Sys.UI.Behavior`. Metoda `$create` działa inaczej w przypadku zachowań i inaczej w przypadku kontrolki — te nieznaczne różnice zostaną omówione przy okazji analizy tych typów w dalszej części tego rozdziału.

■ UWAGA Parametry metody `$create`

Jedynym parametrem wymaganym przez metodę `$create` jest `type`. Pozostałe parametry — `properties`, `events`, `references` i `element` — mają opcjonalny charakter. Jeśli nie chcemy z nich korzystać, powinniśmy w ich miejsce przekazać wartość `null`.

Przekazanie wartości innej niż `null` za pośrednictwem parametru `element` jest prawidłowe, pod warunkiem że `type`, którego egzemplarz tworzymy, dziedziczy po klasie `Sys.UI.Control` lub `Sys.UI.Behavior`. Jeśli prześlemy ten parametr w sytuacji, gdy tworzony egzemplarz komponentu nie dziedziczy po żadnym z wymienionych typów, zostanie wygenerowany błąd.

Podobnie, jeśli nie prześlemy wartości innej niż `null` za pośrednictwem parametru `element` podczas tworzenia egzemplarza typu dziedziczącego po klasie `Sys.UI.Control` lub `Sys.UI.Behavior`, zostanie wygenerowany błąd.

Aby zademonstrować sposób użycia metody `$create`, przeanalizujemy szereg jej wywołań, zmieniając parametry przekazywane na wejściu tej metody.

■ UWAGA Obiekt `Sys.Application` jest zainicjalizowany

W poniższym materiale poświęconym działaniu metody `$create` zakładamy, że obiekt `Sys.Application` został już zainicjalizowany. Mimo że komponenty nie zawsze są tworzone w takich okolicznościach, decydujemy się na to założenie na potrzeby początkowej analizy metody `$create`, aby uprościć nasze dalsze rozważania.

Okazuje się jednak, że istnieje kilka istotnych różnic dzielących proces tworzenia komponentów po inicjalizacji obiektu `Sys.Application` od procesu tworzenia komponentów przed pełną inicjalizacją tego obiektu — zmiany działania metody `$create` wskutek niepełnej inicjalizacji obiektu `Sys.Application` zostaną omówione przy okazji analizy samego procesu inicjalizacji tego obiektu w rozdziale 4.

Stosowanie parametru `type`

Przyjrzyjmy się najpierw podstawowemu wywołaniu metody `$create` polegającemu na przekazaniu samego typu obiektu, który chcemy utworzyć, oraz wartości `null` w miejsce pozostałych parametrów. Przykład takiego wywołania przedstawiono na listingu 3.7.

LISTING 3.7. Parametr `type`

```
var errorHandler =
    Sys.Component.create(
        ErrorHandler,
        null,
        null,
        null,
        null);
```

type

Opis: Typ tworzonego komponentu

Oczekiwany typ: `type`

Wymagany: Tak

Pozostałe wymagania: Obiekt przekazany za pośrednictwem tego parametru musi być egzemplarzem typu dziedziczącego po klasie `Sys.Component`.

Uwagi: Parametr nie jest otoczony cudzysłowami, ponieważ ma postać obiektu typu `Type`, nie łańcucha. Obiekt klasy `Type` jest egzemplarzem typu `Function` zarejestrowanym w bibliotece Microsoft AJAX Library z wykorzystaniem metody `registerClass`, `registerInterface` lub `registerEnum` (tak jak nasz komponent `ErrorHandler`).

W przedstawionym przykładzie pierwszym działaniem metody `Sys.Component.create` jest sprawdzenie, czy za pośrednictwem parametru `type` przekazano obiekt (w tym przypadku `ErrorHandler`) typu `Type`, który dziedziczy po klasie `Sys.Component`. Po sprawdzeniu tego parametru metoda `Sys.Component.create` tworzy nowy egzemplarz typu `ErrorHandler` i przypisuje go pewnej zmiennej lokalnej.

■ UWAGA Rejestrowanie obiektów implementujących interfejs `IDisposable`

Podczas tworzenia egzemplarza komponentu, nowy obiekt jest rejestrowany w obiekcie `Sys.Application` jako egzemplarz typu implementującego interfejs `IDisposable`. Takie rozwiązanie gwarantuje możliwość wywołania metody `dispose` tego egzemplarza podczas zwalniania samego obiektu `Sys.Application`. Omówimy to zagadnienie bardziej szczegółowo w rozdziale 4.

Bezpośrednio potem następuje wywołanie metody `beginUpdate` już utworzonego egzemplarza komponentu. Działanie metody `beginUpdate` domyślnie ogranicza się do przypisania wartości `true` wewnętrznej fladze aktualizacji, można jednak tę metodę przykryć w ramach implementacji nowego komponentu, aby podejmowała ewentualne dodatkowe działania.

Następnie jest wywoływana metoda `endUpdate` nowego egzemplarza — metoda `endUpdate` przywraca wartość `false` wewnętrznej flagi `_updating`, po czym wykonuje przykrytą przez nas metodę `initialize` (która w naszej wersji dołącza do zdarzenia `error` obiektu `Window` funkcję obsługującą). Po wykonaniu metody `initialize` następuje wywołanie metody `updated`. Jeśli nie przykryliśmy metody `updated`, jej oryginalna wersja nie podejmuje żadnych działań. Do zwróconego komponentu możemy uzyskiwać dostęp za pośrednictwem zmiennej, której przypisano wywołanie metody `Sys.Component.create`.

■ WSKAZÓWKA Test składowej `_initialized`

Metoda `endUpdate` zawiera mechanizm sprawdzający (przed wywołaniem metody `initialize`), czy wewnętrzna składowa `_initialized` reprezentuje wartość `false`. O ile w przypadku metody `$create` w chwili wywoływania `endUpdate` składowa `_initialized` zawsze ma wartość `false`, o tyle w razie wywołania metody `endUpdate` na dalszych etapach cyklu życia danego komponentu składowa `_initialized` będzie miała wartość `true`, co uniemożliwi ponowne wykonanie metody `initialize`. Takie rozwiązanie umożliwia nam wywoływanie metod `beginUpdate` i `endUpdate` bez ryzyka ponownej inicjalizacji naszego komponentu.

Ten prosty przykład dobrze pokazuje, jak ważne są zdania niewypowiedziane. Do tej pory ani razu nie wspomniano w tym punkcie o dodawaniu naszego komponentu do zbioru obiektów zarządzanych przez obiekt `Sys.Application` (jak wiemy, za tę operację odpowiada metoda `Sys.Component.create`). W tym przypadku naszego komponentu nie dodano do tego zbioru, ponieważ nigdy nie ustawiono jego identyfikatora — tylko kom-

ponenty z ustawionymi identyfikatorami są automatycznie dodawane do zbioru komponentów zarządzanych obiektu `Sys.Application`. Można to zmienić, ręcznie ustawiając identyfikator komponentu i dodając go do listy komponentów zarządzanych obiektu `Sys.Application` (patrz listing 3.8).

LISTING 3.8. Ręczne ustawienie identyfikatora komponentu i wywołanie metody `addComponent`

```
var errorHandler =
    Sys.Component.create(
        ErrorHandler,
        null,
        null,
        null,
        null);

errorHandler.set_id("ApplicationErrorHandler");
Sys.Application.addComponent(errorHandler);
```

■ UWAGA Wywołanie metody `addComponent`

Gdybyśmy spróbowali ręcznie dodać dany komponent do obiektu `Sys.Application` bez uprzedniego ustawienia identyfikatora tego komponentu, otrzymalibyśmy błąd czasu wykonywania.

Alternatywnym sposobem rozwiązania tego problemu jest początkowe ustawienie identyfikatora komponentu. Jeśli ustawimy ten identyfikator za pośrednictwem parametru `properties`, nasz komponent zostanie automatycznie dodany do zbioru komponentów zarządzanych przez obiekt `Sys.Application` bezpośrednio po przetworzeniu parametru `events`. Niezbędne zmiany przedstawiono na listingu 3.9.

LISTING 3.9. Ustawienie identyfikatora komponentu w ramach wywołania metody `Sys.Component.create`

```
var errorHandler =
    Sys.Component.create(
        ErrorHandler,
        {id: "ApplicationErrorHandler"},
        null,
        null,
        null);
```

Ponieważ określenie identyfikatora komponentu jest warunkiem jego rejestracji jako komponentu zarządzanego, niemal zawsze powinniśmy tę właściwość definiować przy okazji wywołania metody `$create`. Istnieją jednak specjalne przypadki, w których możemy świadomie zrezygnować z ustawiania identyfikatora lub odkładać tę operację na przyszłość — zdarza się to jednak dość rzadko.

Warto też pamiętać, że identyfikatory komponentów zarządzanych przez obiekt `Sys.Application` muszą być unikatowe. Gdybyśmy spróbowali dodać do zbioru komponentów zarządzanych przez ten obiekt dwa komponenty z tym samym identyfikatorem (niezależnie od tego, czy zrobilibyśmy to za pośrednictwem wyrażenia `$create`, czy ręcznie wywołując metodę `addComponent`), otrzymalibyśmy błąd.

■ UWAGA Korzystanie ze zwróconej zmiennej

Metoda `$create` umożliwia nam dostęp do utworzonego komponentu za pośrednictwem zwróconego przez siebie wskaźnika. Okazuje się jednak, że wspomniany znacznik jest przydatny tylko w pewnych sytuacjach. Ponieważ nasz komponent jest rejestrowany w obiekcie `Sys.Application`, możemy w przyszłości uzyskać dostęp do tego komponentu albo odnajdując go w zbiorze komponentów zarządzanych przez ten obiekt, albo korzystając z jego unikatowego identyfikatora.

Stosowanie parametru *properties*

W tym przykładzie spróbujemy przekazać na wejściu metody `Sys.Component.create` kilka początkowych wartości właściwości. Wywołanie tej metody przedstawiono na listingu 3.10.

LISTING 3.10. Przekazywanie początkowych wartości właściwości

```
var errorHandler =
  Sys.Component.create(
    ErrorHandler,
    {
      id: "ApplicationErrorHandler",
      disableErrorPublication: true
    },
    null,
    null,
    null);
```

properties

Oczekiwany typ: `Object`

Wymagany: Nie

Opis: Obiekt zawierający pary klucz-wartość, gdzie klucz reprezentuje nazwę ustawianej właściwości komponentu, a wartość reprezentuje to, co tej właściwości ma zostać przypisane.

W przedstawionym przykładzie początkowe kroki metody `$create` są takie same jak w poprzednim przykładzie. Metoda `$create` weryfikuje wartość parametru `type`, tworzy nasz komponent i wykonuje metodę `beginUpdate`.

W kolejnym kroku metoda `$create` przypisuje przekazane wartości właściwościom komponentu. Właściwości i ich wartości są przekazywane na wejściu tej metody z wykorzystaniem składni łańcuch-obiekt (wyróżnionej pogrubieniem na listingu 3.10). Zamiast korzystać ze wspomnianej składni, można by użyć przedstawionego na listingu 3.11 kodu tworzącego obiekt, jednak właśnie składnia łańcuch-obiekt jest w tej sytuacji krótsza i bardziej czytelna.

LISTING 3.11. Tworzenie obiektu właściwości z wykorzystaniem zmiennych

```
var initialProperties = new Object;  
initialProperties.id = "ApplicationErrorHandler";  
initialProperties.disableErrorPublication = true;  
  
var errorHandler =  
    $create(  
        ErrorHandler,  
        initialProperties,  
        null,  
        null,  
        null);
```

Niezależnie od stosowanej składni nasz kod wyraża dążenie do ustawienia dwóch właściwości: `id` oraz `disableErrorPublication`.

Aby ustawienie tych właściwości było możliwe, metoda `$create` deleguje sterowanie do innej metody, nazwanej `Sys$Component_setProperties`. Ta globalna metoda dostępna w ramach biblioteki Microsoft AJAX Library została stworzona właśnie z myślą o ustawianiu właściwości komponentów. Metoda `Sys$Component_setProperties` otrzymuje na wejściu dwa parametry: obiekt `target` i obiekt `properties`.

W ramach tej metody każda z naszych właściwości `expando` dołączona do parametru `properties` jest odczytywana i przetwarzana według zbioru ściśle określonych reguł.

Pierwsza reguła określa, czy dla danej właściwości istnieje metoda zwracająca jej wartość. Nazwę tej metody określa się, poprzedzając nazwę samej właściwości (w tym przypadku `id`) przedrostkiem `get_`. W naszym przykładzie metoda `get_id` istnieje w klasie bazowej `Sys.Component`, zatem warunek tej reguły jest spełniony.

Po stwierdzeniu istnienia metody zwracającej metoda `Sys$Component_setProperties` przystępuje do poszukiwania metody ustawiającej. Nazwa tej metody powinna się składać z nazwy samej właściwości poprzedzonej przedrostkiem `set_`. Ponownie okazuje się, że odpowiednia metoda (w tym przypadku `set_id`) istnieje w klasie bazowej `Sys.Component`.

Po stwierdzeniu istnienia metody ustawiającej metoda `Sys$Component_setProperties` wywołuje ją, przekazując na jej wejściu wartość bieżącej właściwości. W prezentowanym przykładzie metoda ustawiająca otrzymuje wartość `ApplicationErrorHandler`.

Opisywany proces jest powtarzany aż do momentu skutecznego zastosowania wszystkich właściwości danego komponentu lub wystąpienia jakiegoś błędu (związanego na przykład z brakiem metody zwracającej, niewłaściwą liczbą parametrów obsługiwanych przez metodę ustawiającą lub wieloma innymi możliwymi problemami). W naszym przykładzie właściwość `disableErrorPublication` jest inicjalizowana z wartością `true`.

■ UWAGA Iteracyjne przeszukiwanie właściwości

Dostęp do właściwości `expando` dołączonych do parametru `properties` uzyskujemy, korzystając z pętli `for...in`. Jak wiemy z rozdziału 1., zatytułowanego „Programowanie w języku JavaScript”, pętla `for...in` iteracyjnie przeszukuje właściwości obiektu i umieszcza we wskazanej zmiennej bieżącą właściwość.

Po umieszczeniu nazwy bieżącej właściwości w zmiennej, dostęp do wartości skojarzonej z tą nazwą można uzyskać według reguł obowiązujących w świecie tablic asocjacyjnych opisanych w rozdziale 1.

Wywoływanie metod ustawiających

Jak już wspomniano, metoda ustawiająca właściwość jest wykonywana w trakcie tworzenia obiektu komponentu. Podobnie jak w językach platformy .NET, metoda ustawiająca może zawierać dowolny kod niezbędny do ustawienia odpowiedniej właściwości. Jeśli działanie tej metody będzie procesem długotrwałym, tworzenie komponentu będzie wstrzymane do czasu zakończenia tego procesu.

W tej sytuacji musimy zachować należytą ostrożność, aby metoda ustawiająca była możliwie efektywna i — tym samym — aby proces tworzenia komponentu trwał jak najkrócej.

Jeśli dysponujemy jakimś dodatkowym kodem, który jednak nie powinien być wykonywany w ramach procesu tworzenia komponentu, możemy tego wykonywania uniknąć, uzależniając je od wartości składowej prywatnej `_updating`:

```
...
set_disableErrorPublication: function(value) {
  if (!this.get_updating()) {
    this.raisePropertyChanged
      ("disableErrorPublication");
  }
  this._disableErrorPublication = value;
},
...
```

W powyższym przykładzie kodu źródłowego przed wygenerowaniem zdarzenia `propertyChanged` upewniamy się, że dany komponent nie jest aktualizowany. Sprawdzenie flagi `_updating` jest nieporównanie mniej kosztowne niż przechodzenie przez cały proces generowania zdarzenia.

Ostrzeżenie: Przedstawiony kod ma wyłącznie charakter testowy. Przed jego użyciem powinniśmy dokładnie sprawdzić, czy w naszym przypadku zdarzenie `propertyChanged` rzeczywiście nie powinno być generowane w czasie, gdy dany komponent jest aktualizowany.

Ustawianie właściwości złożonych

Przykłady właściwości `id` i `disableErrorPublication` naszego egzemplarza komponentu `ErrorHandler` ustawianych za pośrednictwem metody `$create` są dość proste. Okazuje się jednak, że istnieją cztery bardziej zaawansowane scenariusze ustawiania właściwości, które warto wykorzystać podczas tworzenia złożonych komponentów w ramach pojedynczych wyrażeń:

1. Ustawianie wartości pozbawionej metody ustawiającej lub zwracającej, na przykład atrybutu elementu DOM lub właściwości dołączonej do prototypu.
2. Dopisywanie elementów do tablicy.
3. Ustawianie właściwości podkomponentu, czyli komponentu zawartego w innym komponencie.
4. Dodawanie właściwości do istniejącego obiektu.

Każdy z tych scenariuszy zilustrowano w kodzie poniższego, dość mało realistycznego komponentu `MyComplexComponent`:

```
MyComplexComponent = function() {
  MyComplexComponent.initializeBase(this);
  this.city = null;
  this._areaCodes = [];
  this._myObject = { firstName: "Harry" };
  this.subComponent =
    $create(ErrorHandler,
      null,
      null,
      null,
      null);
};

MyComplexComponent.prototype = {
  someExpandoProperty: null,
  get_address: function() {
    return this._address;
  },
  set_address: function(value) {
    this._address = value;
  },
  get_areaCodes: function() {
    return this._areaCodes;
  },
  get_myObject: function() {
    return this._myObject;
  }
};

MyComplexComponent.registerClass(
  "MyComplexComponent",
  Sys.Component);
```

```
var newComponent =
  $create(
    MyComplexComponent,
    {
      id: "MyNewComplexComponent",
      city: "Sanok",
      areaCodes: [619, 858, 760],
      someExpandoProperty: "Moja wartość właściwości expando",
      subComponent:
        {
          id: "ApplicationErrorHandler",
          disableErrorPublication: "true"
        },
      myObject: { lastName: "Nowak" }
    },
    null,
    null,
    null);
```

1. Ustawianie wartości, dla której nie istnieje metoda zwracająca lub ustawiająca.
Ten scenariusz zilustrowano na przykładzie ustawiania właściwości `city` i `someExpandoProperty`. Tego rodzaju właściwości można ustawiać, ponieważ stanowią istniejące pola swojego obiektu. Gdyby nie istniały, wyrażenie `setProperties` by ich nie dodało.
2. Dołączanie elementów do tablicy.
Drugi zaawansowany scenariusz zilustrowano na przykładzie właściwości `areaCodes`. W tym przypadku zdefiniowano nową tablicę złożoną z trzech elementów (619, 858 i 760) i przypisano ją właściwości `areaCodes`. Dodawanie elementów do istniejącej tablicy wymaga metody zwracającej, ale nie wymaga metody ustawiającej wartość tej właściwości. W razie istnienia metody ustawiającej istnieje możliwość jej użycia zamiast kodu metody zwracającej — wówczas to kod metody ustawiającej będzie odpowiadał za dołączenie elementów do tablicy. Warto też pamiętać o konieczności uprzedniego utworzenia egzemplarza danej tablicy. Jeśli okaże się, że dana zmienna wskazuje na `null` lub `undefined`, zostanie wygenerowany błąd.
3. Ustawianie właściwości podkomponentu.
Trzeci zaawansowany scenariusz przedstawiono na przykładzie właściwości `subComponent`. W tym przypadku zdefiniowano podobiekt zawierający właściwości `id` oraz `disableErrorPublication`, które zdefiniowano wcześniej w ramach komponentu `ErrorHandler`. Kiedy metoda `setProperties` odkrywa te właściwości, uzyskuje dostęp do podkomponentu, po czym rekurencyjnie wywołuje metodę `setProperties`, stosując ten podkomponent w roli parametru `target` oraz podobiekt zawierający te właściwości w roli parametru `properties`. Tego rodzaju wywołania rekurencyjne mogą być wykorzystywane na dowolnej liczbie poziomów (jeśli ustawiono parametr `properties` w odpowiedni sposób).

Równie dobrze moglibyśmy tutaj zdefiniować metodę zwracającą i otrzymać ten sam efekt, gdybyśmy jednak dodatkowo zdefiniowali metodę ustawiającą, proces ustawiania właściwości podkomponentu nie działałby zgodnie z naszymi oczekiwaniami.

Metoda `setProperties` wywołwana rekurencyjnie z wykorzystaniem komponentu w roli parametru `target` sama wywołuje metodę `beginUpdate` tego komponentu przed wejściem w pętlę `for...in` i metodę `endUpdate` po opuszczeniu tej pętli. Warto o tym pamiętać, jeśli w naszym kodzie korzystamy z metody `get_updating`.

4. Ustawianie właściwości zwykłego obiektu JavaScriptu.

Czwarty, ostatni zaawansowany scenariusz przedstawiono na przykładzie właściwości `myObject`. Właściwość `myObject` definiuje prosty obiekt zawierający właściwość `lastName`, która reprezentuje wartość "Nowak". Kiedy metoda `setProperties` odkrywa tę właściwość, stosuje rekurencyjne wywołanie metody `setProperties`, aby zastosować tę nową właściwość dla składowej `myObject`. Tym razem zamiast przekazywać komponent za pośrednictwem parametru `target`, przekazujemy w roli tego parametru właściwość `myObject`, a obiekt nowej właściwości jest przekazywany za pośrednictwem właściwości `properties`.

Jak widać, parametr `properties` metody `$create` może z powodzeniem obsługiwać kilka zaawansowanych scenariuszy. Jeśli będziesz o tej możliwości pamiętać, być może znajdziesz dla tych rozwiązań zastosowania w swoim kodzie.

Stosowanie parametru `events`

W tym przykładzie spróbujemy wykorzystać parametr `events` do skojarzenia metody obsługującej z dostępnym, zainicjalizowanym zdarzeniem. Możliwy sposób realizacji tego zadania pokazano na listingu 3.12.

LISTING 3.12. Przekazywanie metod obsługujących na wejściu metody `$create`

```
$create(  
  ErrorHandler,  
  {  
    id:"ApplicationErrorHandler",  
    disableErrorPublication: true  
  },  
  {  
    unhandledErrorOccurred:  
      function(sender, args) {  
        alert(args._stackTrace);  
      }  
  },  
  null,  
  null);
```

events

Oczekiwany typ: Object

Wymagany: Nie

Opis: Obiekt zawierający pary klucz-wartość, gdzie klucz reprezentuje nazwę zdarzenia danego komponentu, a wartość zawiera metodę obsługującą, która ma zostać skojarzona z tym zdarzeniem.

W przedstawionym przykładzie początkowe kroki podejmowane przez metodę `$create` są takie same jak w przykładzie ilustrującym techniki stosowania parametru `properties`. Metoda `$create` weryfikuje otrzymany typ, tworzy komponent, wykonuje metodę `beginUpdate`, po czym ustawia właściwości tego komponentu.

Po ustawieniu właściwości następuje przetworzenie parametru `events`. Podobnie jak w przypadku parametru `properties`, parametr `events` jest obiektem zawierającym zbiór par klucz-wartość. Elementy obiektu `events` są iteracyjnie przeszukiwane, a każda odnaleziona para klucz-wartość dodaje metodę obsługującą do odpowiedniego zdarzenia aż do wyczerpania całego zbioru lub wystąpienia jakiegoś błędu. Następnie — także podobnie jak w przypadku parametru `properties` — przekazane w ten sposób metody obsługujące są dodawane do zdarzeń za pomocą odpowiedniej metody. W tym przypadku klucz, czyli `unhandledErrorOccurred`, jest automatycznie poprzedzany przedrostkiem `add_` — otrzymujemy więc metodę `add_unhandledErrorOccurred`. Wspomniany łańcuch jest następnie traktowany jako funkcja należąca do definicji danego komponentu. Jeśli metodę `add_unhandledErrorOccurred` rzeczywiście uda się odnaleźć w tym komponencie i jeśli wartość tej pary klucz-wartość jest obiektem typu `Function`, obiekt ten jest przekazywany na wejściu metody `add_unhandledErrorOccurred` jako jej parametr (w ten sposób wskazana metoda jest dodawana do zbioru metod obsługujących dane zdarzenie).

W naszym przykładzie użycia wyrażenia `$create` zdefiniowano metodę obsługującą zdarzenie w ramach samego wywołania. Okazuje się, że tak zdefiniowana metoda może być z powodzeniem skojarzona ze zdarzeniem `unhandledErrorOccurred`. Alternatywnym sposobem przekazania tej metody jest predefiniowanie funkcji obsługującej nasze zdarzenie (patrz listing 3.13).

LISTING 3.13. Predefiniowanie metody obsługującej zdarzenie

```
function unhandledErrorHandler(sender, args) {
    alert(args._stackTrace);
}

$create(
    MyComponent,
    {address: "ul. Fałszywa 123" },
    {
        unhandledErrorOccurred:
            unhandledErrorHandler
    }
),
null,
null);
```

Predefiniowanie metod obsługujących zdarzenia umożliwia ich wielokrotne wykorzystywanie także w innych komponentach lub wywoływanie proceduralne.

Co więcej, gdybyśmy chcieli obsługiwać zdarzenie za pomocą metody zawartej w naszym komponencie (zamiast — jak w kodzie z listingu 3.13 — korzystać z funkcji globalnej), powinniśmy wykonać dodatkowy krok polegający na utworzeniu delegacji obejmującej naszą metodę obsługującą, aby kontekst wskazywał na odpowiedni komponent (patrz listing 3.14).

LISTING 3.14. Opakowanie metody obsługującej w ramach delegacji

```
MyOtherComponent = function() {
  MyOtherComponent.initializeBase(this);
  this._subComponent = null;
};

MyOtherComponent.prototype = {
  _unhandledErrorOccurred: function(sender, args) {
    var stackTrace = args._stackTrace;
    if (typeof(stackTrace) != "undefined") {
      alert ("Ślad stosu tego błędu: " + stackTrace);
    }
  },

  initialize: function() {
    MyOtherComponent.callBaseMethod(this, "initialize");

    this._errorHandler =
      $create(
        ErrorHandler,
        {
          id:"ApplicationErrorHandler",
          disableErrorPublishing: true
        },
        {
          unhandledErrorOccurred:
            Function.createDelegate (
              this,
              this._unhandledErrorOccurred
            )
        },
        null,
        null);

    // powoduje wygenerowanie błędu
    var nullObj = null;
    nullObj.causeError;
  }
};

MyOtherComponent.registerClass("MyOtherComponent", Sys.Component);
```

We fragmencie kodu metody `initialize` typu `MyOtherComponent` wyróżnionym pogrubieniem utworzyliśmy egzemplarz komponentu `ErrorHandler`. Metodę obsługującą przypisywaną do zdarzenia `unhandledErrorOccurred` umieszczamy w delegacji, aby umożliwić wykonywanie metody `_unhandledErrorOccurred` z wykorzystaniem właściwego kontekstu.

■ UWAGA Przedrostki funkcji

Przy okazji omawiania technik ustawiania właściwości i dodawania metod obsługujących zdarzenia za pomocą metody `$create` mogliśmy obserwować sposoby poprzedzania właściwości przedrostkami `get_` i `set_` oraz metod obsługujących przedrostkiem `add_`. Okazuje się, że wymienione przedrostki nie tylko mają charakter estetyczny, ale także znaczenie funkcjonalne.

Stosowanie parametru *references*

Za pośrednictwem parametru `references` możemy przypisać jeden komponent właściwości innego komponentu i — tym samym — łączyć ze sobą różne komponenty. Być może zastanawiasz się, po co mielibyśmy wykorzystywać do tego celu odrębny parametr, skoro równie dobrze można by osiągnąć ten cel za pomocą parametru `properties`. Dodatkowy parametr jest niezbędny, ponieważ podczas tworzenia egzemplarzy komponentów klienckich z wykorzystaniem kodu serwera nie znamy kolejności tworzenia tych komponentów. Jeśli korzystamy z odrębnego parametru, w ramach procesu inicjalizacji realizowanego przez obiekt `Sys.Application` referencje do komponentów są traktowane w specjalny sposób, a ich przypisywanie następuje dopiero po utworzeniu wszystkich komponentów. W ten sposób udało się wyeliminować problemy związane z próbami uzyskiwania przez komponenty dostępu do innych, jeszcze nieistniejących komponentów.

Aby zilustrować znaczenie parametru `references`, przekazujemy jeden komponent w formie referencji do innego komponentu za pośrednictwem wspomnianego parametru wyrażenia `$create`. W tym celu musimy najpierw utworzyć komponent, który będzie można wykorzystać w roli referencji do naszego drugiego komponentu. Na listingu 3.15 przedstawiono dwa przykładowe wyrażenia `$create`. Dla jasności w prezentowanym scenariuszu posłużyliśmy się dwoma fikcyjnymi komponentami.

LISTING 3.15. Przypisywanie referencji

```
// tworzy pierwszy komponent
$create(
    MyComponent,
    {
        id: "MyFirstComponent"
    },
    null,
    null,
    null
);
```

```
// tworzy drugi komponent i przypisuje go właściwości nazwanej
// subComponent i należącej do pierwszego komponentu
$create(
    MyComponent,
    {
        id: "MySecondComponent"
    },
    null,
    {
        subComponent: "MyFirstComponent"
    },
    null
);
```

references

Oczekiwany typ: Object

Wymagany: Nie

Opis: Obiekt zawierający pary klucz-wartość, gdzie klucz reprezentuje właściwość komponentu, a wartość reprezentuje komponent, który tej właściwości ma zostać przypisany. Wartość powinna zawierać identyfikator tego komponentu.

Po wykonaniu kodu odpowiedzialnego za przypisywanie zdarzeń metoda `$create` przystępuje do przetwarzania parametru `references`. Podobnie jak parametry `properties` i `events`, parametr `references` ma postać obiektu z parami klucz-wartość, gdzie klucz reprezentuje właściwość, której chcemy przypisać komponent, a wartość reprezentuje identyfikator komponentu, który ma zostać przypisany tej właściwości.

Na listingu 3.15 obiekt przekazywany za pośrednictwem parametru `references` wyróżniono pogrubieniem. Jak widać, chcemy przypisać właściwości `subComponent` tworzonego komponentu komponent z identyfikatorem `MyFirstComponent`. Podobnie jak omówiona wcześniej metoda `setProperty`, metoda `setReferences` szuka metody ustawiającej, której nazwa odpowiada nazwie metody danej właściwości poprzedzonej przedrostkiem `set_`. W tym przypadku będzie to metoda nazwana `set_subComponent`. Po odnalezieniu tej metody wyrażenie `$create` szuka w zbiorze komponentów zarządzanych przez obiekt `Sys.Application` komponentu z identyfikatorem `MyFirstComponent`. Jeśli uda się odnaleźć ten komponent, zostanie wywołana metoda ustawiająca `set_subComponent`, której parametr będzie reprezentował znaleziony komponent.

■ UWAGA Odnajdywanie komponentów zarządzanych

Za pośrednictwem metody `Sys.Application.find` możemy odnajdywać zarejestrowane komponenty według identyfikatorów. Przy okazji szczegółowego omawiania klasy `Sys.Application` (w rozdziale 4.) przyjrzymy się uważnie między innymi jej metodzie `find`.

■ WSKAZÓWKA Kolejność tworzenia komponentów

Jak już wspomniano, warunkiem prawidłowego działania tego kodu jest dostępność egzemplarza komponentu `MyFirstComponent` przed wykonaniem drugiego wyrażenia `$create`. Referencje do nieistniejących komponentów mogą być stosowane tylko wtedy, gdy obiekt `Sys.Application` znajduje się w fazie inicjalizacji. Omówimy to zagadnienie bardziej szczegółowo w rozdziale 4.

Stosowanie parametru `element`

Ostatni parametr metody `$create`, czyli `element`, reprezentuje wskaźnik do elementu modelu DOM. Ponieważ parametr `element` może być stosowany tylko podczas tworzenia nowego zachowania lub nowej kontrolki, omówimy go przy okazji analizy procesów definiowania i tworzenia wymienionych typów.

Podsumowanie wiedzy o komponentach

Komponent definiuje się nie tylko jako obiekt dziedziczący po klasie `Sys.Component`, ale też jako obiekt zarządzany przez obiekt `Sys.Application`. Oznacza to, że samo utworzenie egzemplarza typu dziedziczącego po klasie `Sys.Component` za pomocą słowa kluczowego `new` nie powoduje automatycznej rejestracji tego egzemplarza w obiekcie `Sys.Application`. Aby dokonać takiej automatycznej rejestracji, musimy użyć metody `$create`. Metoda `$create` dodatkowo realizuje takie zadania jak ustawianie właściwości, kojarzenie metod obsługujących ze zdarzeniami, przypisywanie referencji do innych komponentów i wiązanie tworzonych komponentów z elementem DOM (o czym przekonamy się za chwilę, przy okazji omawiania kontrolki i zachowań). Co więcej, ta sama metoda automatycznie wywołuje metodę `initialize` tworzonych komponentów — w ramach tej metody możemy umieścić dowolny kod, który powinien być wykonywany po ustawieniu wszystkich właściwości, dodaniu metod obsługujących zdarzenia i przypisaniu referencji do komponentów.

Kontrolki

Kontrolka jest specjalnym rodzajem komponentu bezpośrednio związanym z elementem modelu DOM. Pojedynczy element DOM może być związany z tylko **jedną** kontrolką, a wspomniana kontrolka **musi** być związana tylko z tym konkretnym elementem.

Ponieważ z pojedynczym elementem DOM można związać tylko jedną kontrolkę, w praktyce możliwe zastosowania tego rodzaju komponentów ograniczają się do sytuacji, w których chcemy dysponować pełną kontrolą nad tym elementem modelu DOM. W sytuacji, gdy nie jesteśmy pewni swoich oczekiwań odnośnie danego elementu DOM, powinniśmy raczej użyć zachowania, po czym — w razie konieczności — zastąpić je kontrolką. W praktyce przechodzenie od kontrolki do zachowań (i odwrotnie) nie jest zbyt trudne i nie wymaga zbyt wielu zmian w kodzie źródłowym.

Ponieważ kontrolka jest bezpośrednio związana z elementem modelu DOM, zawiera metody stworzone specjalnie z myślą o uzyskiwaniu dostępu i wykonywaniu operacji na skojarzonym z nią elementem DOM. W tabeli 3.5 szczegółowo omówiono metody kontrolki odpowiedzialne za dostęp i modyfikacje powiązanego elementu DOM.

TABELA 3.5. Metody klasy Sys.UI.Control

Nazwa metody	Opis	Składnia
set_id	Przykrywa metodę set_id komponentu. Generuje błąd, ponieważ identyfikator kontrolki zawsze jest skojarzony z identyfikatorem elementu DOM.	brak prawidłowych zastosowań
get_id	Przykrywa metodę get_id komponentu. Zwraca identyfikator elementu DOM związanego z daną kontrolką.	return ctrl.get_id();
get_visible	Zwraca wartość zwróconą przez wywołanie metody Sys.UI.DomElement.getVisible elementu DOM związanego z daną kontrolką.	return ctrl.get_visible();
set_visible	Wywołuje metodę Sys.UI.DomElement.setVisible, korzystając z elementu DOM związanego z daną kontrolką i wartości logicznej przekazanej na wejściu metody set_visible.	ctrl.set_visible(↳(visibility));
get_visibility ↳Mode	Wywołuje metodę Sys.UI.DomElement.getVisibilityMode, korzystając z elementu DOM związanego z daną kontrolką.	return ctrl.get_visibility ↳Mode();
set_visibility ↳Mode	Wywołuje metodę Sys.UI.DomElement.setVisibilityMode, korzystając z elementu DOM związanego z daną kontrolką i parametru typu Sys.UI.VisibilityMode przekazanego na wejściu metody set_visibilityMode.	ctrl.set_visibilityMode(Sys.UI.VisibilityMode);
get_element	Zwraca element modelu DOM związany z daną kontrolką.	return ctrl.get_element();

TABELA 3.5. Metody klasy `Sys.UI.Control` — *ciąg dalszy*

Nazwa metody	Opis	Składnia
<code>addClass</code>	Wywołuje metodę <code>Sys.UI.DomElement.addClass</code> , korzystając z elementu DOM związanego z daną kontrolką i nazwy dodawanej klasy CSS.	<code>ctrl.addClass</code> ↳ <code>(cssClassName)</code>
<code>removeClass</code>	Wywołuje metodę <code>Sys.UI.DomElement.removeClass</code> , korzystając z elementu DOM związanego z daną kontrolką i nazwy usuwanej klasy CSS.	<code>ctrl.removeClass</code> ↳ <code>(cssClassName);</code>
<code>toggleClass</code>	Wywołuje metodę <code>Sys.UI.DomElement.toggleClass</code> , korzystając z elementu DOM związanego z daną kontrolką i nazwy włączanej klasy CSS.	<code>ctrl.toggleClass</code> ↳ <code>(cssClassName);</code>
<code>dispose</code>	Przykrywa metodę <code>dispose</code> klasy <code>Sys.Component</code> . Wywołuje metodę <code>dispose</code> klasy bazowej, przypisuje wartość <code>undefined</code> właściwości <code>expandElement</code> i usuwa z danego komponentu referencję do elementu DOM.	<code>ctrl.dispose();</code>

Nowe pojęcia

Oprócz metod, które uzyskują dostęp do elementu DOM skojarzonego z daną kontrolką i operują na tym elemencie, twórcy klasy `Sys.UI.Control` wprowadzili dwa nowe pojęcia: rodzic kontrolki (ang. *control's parent*) i znana z frameworku ASP.NET propagacja zdarzeń (ang. *event bubbling*).

Rodzic kontrolki

Właściwość `parent` kontrolki zawiera wskaźnik do innej kontrolki. Wartość tej właściwości jest wyznaczana na dwa sposoby. Jeśli właściwość `parent` ustawiono wprost za pomocą metody `set_parent`, wartość przekazana za pośrednictwem parametru tej metody jest rodzicem danej kontrolki. Jeśli rodzica nie określono wprost, w jego poszukiwaniu jest wykorzystywany wskaźnik `parentNode` elementu DOM aż do odnalezienia elementu skojarzonego z jakąś kontrolką — właśnie ta kontrolka jest traktowana jako rodzic naszej kontrolki.

Metody stworzone z myślą o operowaniu na wskaźniku do rodzica kontrolki opisano w tabeli 3.6.

TABELA 3.6. Metody typu `Sys.UI.Control` związane z rodzicem kontrolki

Nazwa metody	Opis	Składnia
<code>get_parent</code>	Zwraca wprost ustawionego rodzica lub pierwszą kontrolkę napotkaną podczas przetwarzania wskaźnika <code>parentNode</code> elementu DOM.	<pre>var parent = ↪ctrl.get_parent();</pre>
<code>set_parent</code>	Wprost ustawia rodzica danej kontrolki.	<pre>ctrl.set_parent(<i>otherCtrl</i>);</pre>

Propagacja zdarzeń

Propagacja zdarzeń jest techniką polegającą na przekazywaniu zdarzeń w górę hierarchii za pośrednictwem wskaźnika do rodzica i — tym samym — umożliwianiu kontrolkom rodziców obsługi tych zdarzeń.

Mechanizm propagowania zdarzeń zaimplementowany w bibliotece Microsoft AJAX Library przypomina technikę propagowania zdarzeń z wykorzystaniem kontrolki ASP.NET. Kontrolka rozpoczyna ten proces, wywołując metodę `raiseBubbleEvent` i przekazując na jej wejściu źródło i argumenty zdarzenia. W ciele metody `raiseBubbleEvent` jest uzyskiwany rodzic danej kontrolki za pośrednictwem dołączonej do niej metody `get_parent`, po czym następuje wywołanie metody `onBubbleEvent` dla tego rodzica. Domyślna implementacja metody `onBubbleEvent` klasy `Sys.UI.Control` zwraca wartość `false`, oznaczającą, że dana kontrolka nie obsłużyła tego zdarzenia i że proces propagacji powinien trwać dalej (zdarzenie powinno być przekazywane w górę hierarchii).

Jeśli dana kontrolka jest zainteresowana obsługą tak propagowanego zdarzenia, może to zrobić, przykrywając domyślną implementację metody `onBubbleEvent`. W przykrytej metodzie należy określić, czy proces propagowania zdarzenia powinien być kontynuowany, czy zatrzymany na poziomie tej kontrolki. Jeśli propagacja zdarzenia zatrzymuje się na rodzicu danej kontrolki, metoda `onBubbleEvent` zwraca wartość `true`. Jeśli jednak dane zdarzenie ma być propagowane dalej, w górę drzewa rodziców kontrolki, metoda `onBubbleEvent` zwraca wartość `false`.

Metody stworzone z myślą o technice propagowania zdarzeń szczegółowo opisano w tabeli 3.7.

UWAGA Dodatkowe metody

Ponieważ klasa `Sys.UI.Control` dziedziczy po klasie bazowej `Sys.Component`, wszystkie metody dostępne dla obiektów klasy `Sys.Component` są dostępne także dla egzemplarzy klasy `Sys.UI.Control`.

TABELA 3.7. Metody typu Sys.UI.Control związane z propagowaniem zdarzeń

Nazwa metody	Opis	Składnia
onBubbleEvent	Metoda onBubbleEvent jest częścią frameworku propagowania zdarzeń. Aby zapewnić jakąś funkcjonalność tej metody, należy ją przykryć. Metoda onBubbleEvent domyślnie zwraca wartość false.	Metoda onBubbleEvent jest automatycznie wywoływana przez metodę raiseBubbleEvent.
raiseBubbleEvent	Metoda raiseBubbleEvent jest częścią frameworku propagowania zdarzeń. Przeszukuje listę rodziców kontrolki i uruchamia metodę onBubbleEvent każdego z tych obiektów.	ctrl.raiseBubbleEvent(source, args);

Definiowanie nowej kontrolki

Podobnie jak w przypadku nowych komponentów, podczas definiowania nowych kontrolk będziemy stosowali model prototypowy opisany w rozdziale 2. Aby zilustrować sposób definiowania nowej kontrolki, utworzymy nową kontrolkę skojarzoną z polem tekstowym (typu textbox), która umożliwi nam tylko wpisywanie wartości liczbowych. Na listingu 3.16 przedstawiono kod niezbędny do zdefiniowania nowej kontrolki NumberOnlyTextBox.

LISTING 3.16. Definiowanie nowego typu kontrolki

```

/// <reference name="MicrosoftAjax.js"/>
NumberOnlyTextBox = function(element) {
    NumberOnlyTextBox.initializeBase(this, [element]);
    this._keyDownDelegate = null;
};

NumberOnlyTextBox.prototype = {
    initialize: function() {
        NumberOnlyTextBox.callBaseMethod(this, 'initialize');
        this._keyDownDelegate =
            Function.createDelegate(this, this._keyDownHandler);
        $addHandler(this.get_element(), "keydown", this._keyDownDelegate);
    },

    dispose: function() {
        $removeHandler
            (this.get_element(), "keydown", this._keyDownDelegate);
        this._keyDownDelegate = null;
        NumberOnlyTextBox.callBaseMethod(this, 'dispose');
    },
};

```



```
_keyDownHandler: function(e) {  
    return ((e.keyCode >= 48 && e.keyCode <= 57) || (e.keyCode == 8));  
}  
};  
  
NumberOnlyTextBox.registerClass("NumberOnlyTextBox", Sys.UI.Control);
```

Jak widać na listingu 3.16, istnieją dwie ważne różnice dzielące kontrolkę `NumberOnlyTextBox` od zadeklarowanego wcześniej komponentu `ErrorHandler`.

Po pierwsze, funkcję klasy bazowej naszej kontrolki `NumberOnlyTextBox` pełni typ `Sys.UI.Control`, nie — jak wcześniej — typ `Sys.Component`.

Po drugie, konstruktor naszego nowego typu otrzymuje parametr `element` i przekazuje go do konstruktora swojej klasy bazowej za pośrednictwem metody `initializeBase`. Parametr `element` reprezentuje element modelu DOM, który ma zostać skojarzony z daną kontrolką.

W odpowiedzi na przekazanie wspomnianego elementu na wejściu konstruktora klasy `Sys.UI.Control` są podejmowane trzy działania. Po pierwsze, tak przekazany element DOM jest weryfikowany pod kątem ewentualnego związku z inną kontrolką. Jeśli okaże się, że taki związek istnieje, konstruktor klasy `Sys.UI.Control` generuje błąd i cały proces tworzenia kontrolki kończy się niepowodzeniem. W przeciwnym razie konstruktor przystępuje do drugiego kroku, polegającego na przypisaniu tego elementu DOM wewnętrznej składowej `_element`. I wreszcie kontrolka sama jest przypisywana temu elementowi modelu DOM z wykorzystaniem właściwości `expando` nazwanej `control`. Gdybyśmy utworzyli referencję do elementu związanego z naszą kontrolką, moglibyśmy uzyskać dostęp do związanej z nim kontrolki za pomocą następującego wywołania:

```
$get("TextBox1").control;
```

Na listingu 3.17 przedstawiono kod demonstrujący omówione powyżej reguły na przykładzie naszego nowo utworzonego typu kontrolki `NumberOnlyTextBox`.

LISTING 3.17. Tworzenie egzemplarza typu `NumberOnlyTextBox` z wykorzystaniem słowa kluczowego `new`

```
<html xmlns="http://www.w3.org/1999/xhtml">  
<head runat="server">  
    <title>Control Testing!</title>  
</head>  
  
<body>  
    <form id="form1" runat="server">  
        <asp:ScriptManager ID="SM1" runat="server" />  
  
        // dla uproszczenia pominięto definicję typu NumberOnlyTextBox  
  
        <asp:TextBox ID="txtBox1" runat="server" Width="150px" />  
  
        <script type="text/javascript">  
  
            var numberOnlyTextBox =
```

```

    new NumberOnlyTextBox($get("txtBox1"));

    // wyświetla „txtBox1”
    alert ("Identyfikator elementu związanego z kontrolką numberOnlyTextBox: " +
          numberOnlyTextBox.get_element().id);

    // wyświetla „txtBox1”
    alert ("Identyfikator elementu związanego z kontrolką numberOnlyTextBox: " +
          $get("txtBox1").control.get_id());

    // generuje błąd JavaScriptu, ponieważ dana kontrolka
    // jest już związana z polem txtBox1
    var numberOnlyTextBox2 =
        new NumberOnlyTextBox ($get("txtBox1"));
</script>

</form>
</body>
</html>

```

Tworzenie kontrolki

Na listingu 3.17 do utworzenia nowego egzemplarza naszego typu `NumberOnlyTextBox` wykorzystaliśmy słowo kluczowe `new`. Z podrozdziału poświęconego naszemu komponentowi wiemy już, że metoda `$create` wykonuje cały pakiet zadań i nie ogranicza się tylko do utworzenia nowego egzemplarza wskazanego typu, a ponieważ nasz nowy typ dziedziczy po klasie `Sys.UI.Control`, która z kolei dziedziczy po klasie `Sys.Component`, możemy użyć wyrażenia `$create` w taki sam sposób jak w przypadku komponentu `ErrorHandler`.

Zamiast tracić czas na ponowne omawianie metody `$create`, ograniczymy się tylko do analizy zastosowań parametru `element`, ponieważ tylko w jego przypadku mamy do czynienia z istotnymi różnicami. Podstawą naszych analiz będzie kod z listingu 3.17, który zmodyfikujemy przez zastosowanie metody `$create`. Kod powstały w wyniku tych zmian wprowadzono na listingu 3.18.

LISTING 3.18. Tworzenie egzemplarza typu `NumberOnlyTextBox` z wykorzystaniem metody `$create`

```

<html>
<head runat="server">
  <title>Test kontrolki!</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="SM1" runat="server" />

    // dla uproszczenia pominięto definicję typu NumberOnlyTextBox

    <asp:TextBox ID="txtBox1" runat="server" Width="150px" />

```

```
<script type="text/javascript">
    $create(
        NumberOnlyTextBox,
        null,
        null,
        null,
        $get("txtBox1")
    );
</script>

</form>
</body>
</html>
```

Kod wyróżniony pogrubieniem zawiera wywołanie metody `$create`. Warto zwrócić szczególną uwagę na sposób przekazania wywołania `$get("txtBox1")` za pośrednictwem parametru `element` metody `$create`. W czasie tworzenia nowego egzemplarza klasy `NumberOnlyTextBox` metoda `$create` sprawdza, czy typ `NumberOnlyTextBox` dziedziczy po typie `Sys.UI.Control` lub `Sys.UI.Behavior`. Jeśli tak (a tak jest w tym przypadku), metoda `$create` wykorzystuje parametr `element` w roli argumentu wywoływanego konstruktora (podobne rozwiązanie zastosowaliśmy na listingu 3.17 przed użyciem metody `$create` do utworzenia naszej nowej kontrolki).

UWAGA Ustawianie identyfikatora kontrolki

Inaczej niż w przypadku komponentów i zachowań, bezpośrednie ustawianie identyfikatorów kontrolki nie jest możliwe. Identyfikator kontrolki zawsze jest tożsamy z identyfikatorem powiązanego z nią elementu modelu DOM.

Podsumowanie wiedzy o kontrolkach

Kontrolki w niewielkim stopniu różnią się od komponentów (klasa komponentów jest typem bazowym klas kontrolki). Najważniejsza różnica polega na konieczności związania każdej kontrolki z pojedynczym elementem modelu DOM (w przypadku komponentów istnienie tego rodzaju związków nie jest konieczne).

Zachowania

Zachowanie to kolejny specjalny rodzaj komponentu związany z elementami modelu DOM. Podobnie jak kontrolki, zachowania muszą być skojarzone z elementami DOM. Okazuje się jednak, że w inaczej niż w przypadku kontrolki, pojedynczy element DOM może być związany z więcej niż jednym zachowaniem.

W wymiarze praktycznym zachowania definiują oczekiwany sposób zachowywania się elementu DOM. Możemy na przykład zdecydować, że dany element DOM powinien **być związany** do pojedynczego wiersza, **plywać** na stronie bądź **wypełniać** całą dostępną przestrzeń na ekranie. Wszystkie te zachowania możemy dołączyć do elementu modelu DOM właśnie za pomocą egzemplarzy typów zachowań.

Aby ułatwić nam definiowanie nowych typów zachowań i korzystanie z egzemplarzy tych typów, typ bazowy `Sys.Behavior` definiuje kilka dodatkowych metod, uzupełniających zbiór metod oferowanych przez jego typ bazowy `Sys.Component`. Metody klasy `Sys.Behavior` szczegółowo omówiono w tabeli 3.8.

TABELA 3.8. Metody klasy `Sys.UI.Behavior`

Nazwa metody	Opis	Składnia
<code>get_element</code>	Zwraca element modelu dom związany z danym zachowaniem.	<code>return behavior. ↳get_element();</code>
<code>get_name</code>	Zwraca nazwę danego zachowania. Jeśli wprost określono nazwę danego zachowania, zwracana jest właśnie ta nazwa. W przeciwnym razie metoda <code>get_name</code> zwraca skróconą nazwę typu tego zachowania.	<code>return behavior. ↳get_name();</code>
<code>set_name</code>	Ustawia nazwę danego zachowania. Nazwa ustawiana w ten sposób musi być unikatowa. Nazw zachowań nie można ustawiać po ich inicjalizacji. Nazwa zachowania nie może się rozpoczynać od znaku białego, nie może się kończyć znakiem białym, nie może też być łańcuchem pustym.	<code>behavior.set_name ↳("HiddenElm");</code>
<code>initialize</code>	Wywołuje metodę <code>initialize</code> klasy bazowej. Kojarzy dane zachowanie z odpowiednim elementem DOM przez dodanie do tego elementu właściwości <code>expando</code> z nazwą danego zachowania.	<code>behavior.initialize();</code>

TABELA 3.8. Metody klasy Sys.UI.Behavior — *ciąg dalszy*

Nazwa metody	Opis	Składnia
dispose	Przykrywa metodę dispose klasy Component. Wywołuje metodę dispose klasy bazowej, usuwa z elementu DOM właściwość expando reprezentującą nazwę danego zachowania i usuwa referencję do tego elementu z samego zachowania.	<pre>Sys. behavior.dispose();</pre>
get_id	Zwraca wartość właściwości id danego komponentu (jeśli tę wartość ustawiono). Jeśli wartość tej właściwości nie została ustawiona, metoda get_id zwraca identyfikator elementu DOM powiązanego z tym zachowaniem i nazwę samego zachowania.	<pre>return behavior. ↳get_id();</pre>
Sys.UI.Behavior. ↳getBehaviorsByType	Zwraca wszystkie zachowania dołączone do elementu DOM określonego typu.	<pre>return Sys.UI.Behavior. ↳getBehaviorsByType ↳(element, typeName)</pre>
Sys.UI.Behavior. ↳getBehaviorByName	Zwraca zachowanie dołączone do wskazanego elementu modelu DOM (jeśli takie zachowania istnieją).	<pre>return Sys.UI.Behavior. ↳getBehaviorByName ↳(element, behaviorName)</pre>
Sys.UI.Behavior. ↳getBehaviors	Zwraca kopię zachowań dołączonych do wskazanego elementu DOM. Jeśli określony element modelu DOM nie jest związany z żadnymi zachowaniami, metoda getBehaviors zwraca tablicę pustą.	<pre>return Sys.UI.Behavior. ↳getBehaviors(element);</pre>

Definiowanie zachowania

Podobnie jak podczas definiowania nowych komponentów i nowych kontroltek, definiując nowe zachowania posługujemy się modelem prototypowym omówionym w rozdziale 2. Zamiast tworzyć od podstaw nowy przykład, odpowiednio zmodyfikujemy przedstawiony w poprzednim podrozdziale kod definiujący kontrolkę NumberOnlyTextBox. Na listingu 3.19 przedstawiono kod niezbędny do zdefiniowania zachowania NumberOnlyTextBox.

LISTING 3.19. Definiowanie typu zachowania

```

/// <reference name="MicrosoftAjax.js"/>
NumberOnlyTextBox = function(element) {
    NumberOnlyTextBox.initializeBase(this, [element]);
    this._keyDownDelegate = null;
};

NumberOnlyTextBox.prototype = {
    initialize: function() {
        NumberOnlyTextBox.callBaseMethod(this, 'initialize');
        this._keyDownDelegate =
            Function.createDelegate(this, this._keyDownHandler);
        $addHandler(this.get_element(), "keydown", this._keyDownDelegate);
    },

    dispose: function() {
        $removeHandler
            (this.get_element(), "keydown", this._keyDownDelegate);
        this._keyDownDelegate = null;
        NumberOnlyTextBox.callBaseMethod(this, 'dispose');
    },

    _keyDownHandler: function(e) {
        return ((e.keyCode >= 48 && e.keyCode <= 57) || (e.keyCode == 8));
    }
};

NumberOnlyTextBox.registerClass("NumberOnlyTextBox", Sys.UI.Behavior);

```

Jak widać, kod definiujący nasze zachowanie `NumberOnlyTextBox` niemal niczym nie różni się od kodu definiującego wcześniej tak samo nazwaną kontrolkę. Różnice sprowadzają się do rezygnacji z typu bazowego `Sys.UI.Control` na rzecz typu `Sys.UI.Behavior`.

Podobnie jak konstruktor klasy `Sys.UI.Control`, konstruktor typu `Sys.UI.Behavior` otrzymuje na wejściu parametr reprezentujący element modelu DOM. W ciele tego konstruktora wartość wspomnianego parametru jest przypisywana wewnętrznej składowej `_element`, co jest równoznaczne ze skojarzeniem odpowiedniego elementu DOM z tworzonym zachowaniem. Następnie nasze zachowanie jest dodawane do właściwości `expando` nazwanej `_behaviors`. Właściwość `expando _behaviors` pod pewnymi względami przypomina właściwość `control` stosowaną dla kontroltek, ale ma postać tablicy, która może reprezentować więcej niż jedno zachowanie związane z danym elementem DOM.

Tworzenie zachowania

Z podrozdziałów poświęconych komponentom i kontrolkom wiemy, że najlepszym sposobem konstruowania nowych egzemplarzy typów dziedziczących po klasie `Sys.Component` jest korzystanie z metody `$create` — nie inaczej jest w przypadku zachowań.

W praktyce proces tworzenia zachowania przebiega dokładnie tak samo jak proces tworzenia kontrolki, zatem kod przedstawiony już na listingu 3.18 w zupełności wystarczy jako przykład tego rozwiązania.

Okazuje się jednak, że inaczej niż w przypadku kontroltek, podczas tworzenia zachowań mogą występować dwa poważne problemy ściśle związane z unikatowością zachowań.

Problemy związane z unikatowością zachowań

Z pierwszym problemem mamy do czynienia w sytuacji, gdy identyfikator (właściwość `id`) zachowania nie jest ustawiona i gdy jest generowany automatycznie na podstawie skojarzonego z tym zachowaniem elementu DOM i nazwy samego zachowania. Ponieważ wspomniana wartość jest generowana automatycznie, jest wielce prawdopodobne, że ten sam identyfikator zostanie przypisany więcej niż jednemu zachowaniu. Jeśli ten sam identyfikator zostanie wygenerowany dla wielu zachowań, próba rejestracji drugiego i każdego kolejnego zachowania z tego zbioru w obiekcie `Sys.Application` zakończy się niepowodzeniem, ponieważ komponenty zarządzane przez ten obiekt muszą mieć unikatowe identyfikatory. Opisany problem zademonstrowano na listingu 3.20.

LISTING 3.20. Tworzenie zachowań z tym samym identyfikatorem

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Test zachowań!</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="SM1" runat="server" />

    // dla uproszczenia pominięto definicję typu NumberOnlyTextBox

    <asp:TextBox ID="txtBox1" runat="server" Width="150px" />

    <script type="text/javascript">
      $create(
        NumberOnlyTextBox,
        null,
        null,
        null,
        $get("txtBox1")
      );

      // to wywołanie spowoduje błąd, ponieważ identyfikator tego komponentu
      // będzie taki sam jak w poprzednio zarejestrowanym zachowaniu
      $create(
        NumberOnlyTextBox,
        null,
        null,
        null,
```

```
        $get("txtBox1")
    );
</script>

</form>
</body>
</html>
```

Ponieważ w przedstawionym przykładzie nie ustawiliśmy wprost nazw ani identyfikatorów tworzonych zachowań, właściwości `id` obu tych zachowań będą miały wartość `txtBox1$NumberOnlyTextBox`. Automatycznie generowany identyfikator zachowania składa się z identyfikatora powiązanego elementu DOM (w tym przypadku `txtBox1`), symbolu `$` oraz nazwy samego zachowania, która w razie braku nazwy określonej wprost jest zastępowana nazwą typu zachowania (bez ewentualnych przestrzeni nazw).

■ UWAGA Nazwa `NumberOnlyTextBox`

W naszym przykładzie automatycznie generowana nazwa zachowania jest identyczna z nazwą typu, czyli `NumberOnlyTextBox`.

Gdyby typ naszego zachowania należał do jakichś przestrzeni nazw, na przykład `MyProject.Behaviors.NumberOnlyTextBox`, automatycznie generowana nazwa tego zachowania i tak miałyby postać `NumberOnlyTextBox`.

Kiedy drugie zachowanie spróbuje się zarejestrować w obiekcie `Sys.Application`, zostanie wygenerowany błąd w związku z istnieniem już zarejestrowanego komponentu z takim samym identyfikatorem.

Aby rozwiązać ten problem, należy wprost ustawić albo nazwę, albo identyfikator naszego zachowania. Niezależnie od tego, którą właściwość zdecydujemy się ustawić, użyta wartość musi być unikatowa. Jeśli ustawimy właściwość `id`, użyty identyfikator powinien być unikatowy w skali wszystkich komponentów. Jeśli ustawimy właściwość `name`, użyta nazwa powinna być unikatowa w skali wszystkich zachowań skojarzonych z tym samym elementem modelu DOM. Na listingu 3.21 przedstawiono kod, który prawidłowo tworzy dwa zachowania związane z tym samym polem typu `textbox`.

LISTING 3.21. Ustawianie identyfikatora zachowania

```
<script type="text/javascript">
    $create(
        NumberOnlyTextBox,
        {id: "Behavior1" },
        null,
        null,
        $get("txtBox1")
    );
```



```
$create(  
    NumberOnlyTextBox,  
    {id: "Behavior2" },  
    null,  
    null,  
    $get("txtBox1")  
);  
</script>
```

Drugi poważny problem związany z tworzeniem zachowań może występować w sytuacji, gdy dołączamy wiele egzemplarzy tego samego zachowania do jednego elementu modelu DOM i gdy nazwy tych zachowań nie zostały wprost ustawione. Ponieważ automatycznie generowane nazwy egzemplarzy tego samego zachowania będą takie same (na przykład `NumberOnlyTextBox`), nie będziemy mogli ich odnaleźć za pośrednictwem metody `Sys.UI.getBehaviorByName`.

Dołączanie wielu egzemplarzy tego samego zachowania do jednego elementu modelu DOM zdarza się dość rzadko, ale nie można takiej sytuacji wykluczyć. Na listingu 3.22 przedstawiono kod ilustrujący możliwość odnajdywania tylko jednego z zachowań typu `NumberOnlyTextBox` skojarzonych z naszym polem tekstowym.

LISTING 3.22. Problemy z odnajdywaniem zachowań według nazw

```
<html>  
<head runat="server">  
    <title>Test zachowań!</title>  
</head>  
<body>  
    <form id="form1" runat="server">  
        <asp:ScriptManager ID="SM1" runat="server" />  
  
        // dla uproszczenia pominięto definicję typu NumberOnlyTextBox  
  
        <asp:TextBox ID="txtBox1" runat="server" Width="150px" />  
  
        <script type="text/javascript">  
            $create(  
                NumberOnlyTextBox,  
                {id: "Behavior1"},  
                null,  
                null,  
                $get("txtBox1")  
            );  
  
            $create(  
                NumberOnlyTextBox,  
                {id: "Behavior2"},  
                null,  
                null,
```

```

    $get("txtBox1")
  );

  var beh = Sys.UI.Behavior.getBehaviorByName
    ($get("txtBox1"), " NumberOnlyTextBox ");

  alert (beh.get_name());

  var behaviorsAssignedToDom =
    Sys.UI.Behavior.getBehaviors($get("txtBox1"));

  var behaviors = '';

  for (var i=0; i<behaviorsAssignedToDom.length; i++) {
    behaviors += behaviorsAssignedToDom[i].get_name() + " ";
  }

  // wyświetla „NumberOnlyTextBox NumberOnlyTextBox”, ponieważ
  // zdefiniowano dwa tak samo nazwane zachowania
  alert (behaviors);

</script>

</form>
</body>
</html>

```

Rozwiązanie tego problemu wymaga ustawienia wprost nazw wszystkich tworzonych zachowań.

Na zakończenie tego podpunktu poświęconego problemom związanym z tworzeniem zachowań warto podkreślić, że konsekwentne ustawianie wprost identyfikatorów i (lub) nazw może nam oszczędzić wielu problemów, mimo że brak wyrażeń ustawiających te właściwości nie zawsze powoduje generowanie błędów. Sugerujemy więc ustawianie właściwości id oraz name dla każdego tworzonych egzemplarza zachowania niezależnie od sytuacji. Przykład zastosowania tego wzorca przedstawiono na listingu 3.23.

LISTING 3.23. Przypisywanie identyfikatorów i nazw tworzonym egzemplarzom zachowań

```

<script type="text/javascript">
  $create(
    NumberOnlyTextBox,
    {id: "Behavior1",
      name: "Behavior1"},
    null,
    null,
    $get("txtBox1")
  );

```



```
$create(  
    NumberOnlyTextBox,  
    {id: "Behavior2",  
      name: "Behavior2"},  
    null,  
    null,  
    $get("txtBox1")  
);  
</script>
```

Podsumowanie wiedzy o zachowaniach

Zachowania nie różnią się zbyt wiele od egzemplarzy swojego typu bazowego, czyli komponentów. Najważniejsza różnica ma związek z koniecznością kojarzenia zachowań z elementami modelu DOM (komponenty nie mogą być kojarzone z tego rodzaju elementami). Najważniejszą różnicą dzielącą zachowania od kontrolki jest możliwość dołączania do pojedynczego elementu DOM tylko jednej kontrolki (liczba dołączanych zachowań jest nieograniczona).

Podsumowanie

W tym rozdziale omówiono komponenty, kontrolki i zachowania. Przyjrzelśmy się typowi bazowemu komponentów pod kątem jego popularnych obiektów oraz sposobom rozszerzania tego typu przez kontrolki i zachowania, czyli wyspecjalizowane komponenty zawierające referencje do elementów modelu DOM. Omówiliśmy też techniki samodzielnego konstruowania egzemplarzy wszystkich trzech typów oraz sposoby realizacji tych zadań z wykorzystaniem funkcji `$create`.

W następnym rozdziale skoncentrujemy się na obiekcie `Sys.Application`, który pełni funkcję menedżera wszystkich komponentów, kontrolki i zachowań. Po omówieniu wspomnianego obiektu przystąpimy do analizy technik łączenia elementów biblioteki Microsoft AJAX Library z kodem frameworku ASP.NET AJAX stosowanym po stronie serwera (w tym technik tworzenia komponentów, kontrolki i zachowań za pośrednictwem kodu .NET). I wreszcie, aby ostatecznie wyczerpać temat komponentów, kontrolki i zachowań, zajmiemy się tematem ich lokalizowania i sposobami reagowania na ich umieszczenie w kontrolce `UpdatePanel`.