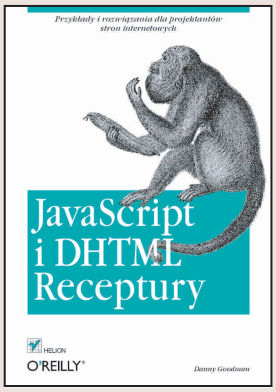
JavaScript i DHTML. Receptury



Autor: Danny Goodman
Tłumaczenie: Piotr Rajca (rozdz. 1 – 10),
Marek Pałczyński (rozdz. 11 – 15, dod. A – C)
ISBN: 83-7361-225-4
Tytuł oryginału: JavaScript & DHTML Cookbook
Format: B5, stron: 604

DHTML i JavaScript już od kilku lat stanowią ważne narzędzia każdego projektanta stron internetowych. Jednak właściwe ich zastosowanie w dalszym ciągu sprawia wiele problemów. Danny Goodman, jeden z najbardziej doświadczonych autorów książek na temat tworzenia stron internetowych, postanowił zebrać rozwiązania tych problemów i przedstawić je projektantom pragnącym ożywić swoje strony. Wszystkie prezentowane przykłady wykorzystują najnowsze standardy stworzone przez ECMA i W3C, co daje gwarancję, że będą działały w różnych przeglądarkach przez wiele następnych lat.

Książka „JavaScript i DHTML. Receptury" skupia się na praktycznych, użytecznych zastosowaniach skryptów na stronach internetowych. Autor nie marnuje miejsca na opisy fajerwerków, fruwających obrazków i zmian kolorów tła. Każde omówione rozwiązanie składa się nie tylko z gotowego do użycia kodu, ale także z gruntownego wyjaśnienia jego działania tak, byś mógł przystosować kod do swojej strony. Opisano zarówno proste skrypty (np. służące do obróbki tekstów, czy sprawdzania poprawności wprowadzonych dat), jak i złożone biblioteki służące do pozycjonowania elementów na stronie czy sortowania tabel.

Ponad 150 receptur obejmuje m.in.:

- Pracę z interaktywnymi formularzami i stylami
- Tworzenie przyjaznej dla użytkownika nawigacji po serwisie
- Efekty wizualne
- Tworzenie dynamicznej treści
- Pozycjonowanie elementów HTML
- Zarządzanie okienkami przeglądarki i ramkami

Tak jak żaden większy serwis internetowy nie może obyć się bez JavaScriptu i DHTML, tak też żaden profesjonalny twórca takich serwisów nie może obyć się bez tej książki.

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

Spis treści

4

Zmienne, funkcje i sterowanie wykonywaniem skryptów

4.0. Wprowadzenie

W tym rozdziale przedstawionych zostało wiele podstawowych zagadnień związanych z językiem JavaScript. Kilka z zamieszczonych tu receptur (ewentualnie ich zmodyfikowanych wersji) może się stać narzędziami wykorzystywanymi w codziennej pracy. Jeśli rozwiązania te nie są często stosowane, to mogą posłużyć do odświeżenia pamięci i stanowić wzorzec, z którego będzie można skorzystać w razie potrzeby.

Nawet proste zagadnienia, takie jak zmienne i funkcje w JavaScripcie, mają swoje niuanse, o których z czasem łatwo zapomnieć. Kilka rozwiązań, takich jak obsługa wyjątków oraz instrukcje `try` oraz `catch` są stosunkowo nowe i wprowadzono je w nowszych wersjach przeglądarek. Twórcy skryptów, którzy nie dysponują formalnym doświadczeniem programistycznym, zazwyczaj mają tendencje do ignorowania błędów, a niestety, postępowanie takie może się w przyszłości na nich zemścić. Z drugiej strony, implementacje niektórych aspektów obsługi wyjątków w poszczególnych przeglądarkach nie są ze sobą zgodne. Jeśli mechanizmy obsługi wyjątków nie są jeszcze wykorzystywane w skryptach (być może ze względu na konieczność zachowania zgodności z wcześniejszymi wersjami przeglądarek), to i tak należy je poznać. Wraz z upływem czasu, kiedy w przeglądarkach zostanie zaimplementowany cały model obiektów dokumentu (DOM) W3C, zasady tworzenia „bezpiecznych skryptów" będą bazować na wykorzystaniu mechanizmów obsługi wyjątków.

Na końcu niniejszego rozdziału zostały zamieszczone pewne sugestie dotyczące poprawiania efektywności działania skryptów. W większości skryptów efektywność działania nie ma kluczowego znaczenia, niemniej jednak w dużych projektach, operujących na dokumentach o złożonej strukturze i znacznych ilościach ukrytych informacji dostarczanych do przeglądarki, na kwestie związane z efektywnością należy zwracać baczną uwagę. W tym rozdziale zostaną przedstawione rozwiązania, które powinny być stosowane nawet w prostych skryptach.

4.1. Tworzenie zmiennych w JavaScripcie

NN 2 IE 3

Problem

Należy utworzyć zmienną bądź to w globalnej przestrzeni skryptu, bądź lokalnie wewnątrz funkcji.

Rozwiązanie

Aby utworzyć pierwszą kopię zmiennej, należy wykorzystać słowo kluczowe `var`, niezależnie od tego, czy zmiennej od razu zostanie przypisana jakaś wartość, czy też przypisanie nastąpi dopiero później. Wszelkie zmienne definiowane poza funkcjami należą do globalnej przestrzeni zmiennych:

```
var mojaZmienna = jakasWartosc;
```

Wszystkie instrukcje umieszczone na stronie, nawet te znajdujące się w funkcjach, mają pełne prawa do odczytu i zapisu zmiennych globalnych.

Gdy zmienna jest definiowana przy użyciu słowa kluczowego `var` wewnątrz funkcji, dostęp do niej będą mieć wyłącznie instrukcje umieszczone w tej samej funkcji:

```
function mojaFunkcja() {
    var mojaZmiennaLokalna = jakasWartosc;
    ...
}
```

Instrukcje umieszczone poza funkcją nie mają dostępu do zmiennych, których zakres ogranicza się do funkcji.

Analiza

W JavaScripcie nie istnieje żaden limit ilości danych, jakie można zapisywać w poszczególnych zmiennych. Maksymalna pojemność zmiennych jest ograniczana wyłącznie przez wielkość pamięci dostępnej dla przeglądarki, przy czym informacje o tej pamięci nie są dostępne z poziomu skryptu.

Ważnym zagadnieniem, które należy zrozumieć, jest zakres zmiennych. Zmienne globalne są dostępne nie tylko dla wszystkich instrukcji skryptu umieszczonych w bieżącym oknie przeglądarki lub ramce, lecz co więcej, dzięki odpowiednim odwołaniom, instrukcje wykonywane w innych ramkach lub oknach (prezentujących strony z tej samej domeny i serwera) także mają do nich dostęp. Na przykład instrukcja umieszczona w ramce zawierającej menu może się odwołać do zmiennej `mojaZmienna` przechowywanej w ramce `zawartosc`, wykorzystując następujące odwołanie:

```
parent.zawartosc.mojaZmienna
```

Jeśli zmienne globalne zostały zdefiniowane w innych ramkach lub oknach przeglądarki, nie trzeba się przejmować możliwością wystąpienia pomiędzy nimi kolizji nazw, gdyż odwołania do nich zawsze będą inne.

Ostrożność należy natomiast zachowywać w wypadkach definiowania, z wykorzystaniem słowa kluczowego `var`, zmiennych wewnątrz funkcji. W razie pominięcia słowa kluczowego `var` zmienna zostanie uznana za globalną. Jeśli w skrypcie została zdefiniowana zmienna globalna o tej samej nazwie, to instrukcje przypisania modyfikujące zmienną wewnątrz funkcji spowodują zmianę wartości zmiennej globalnej. Najbezpieczniejszym rozwiązaniem problemów tego typu jest konsekwentne stosowanie słowa kluczowego `var` przy tworzeniu każdej nowej zmiennej, i to niezależnie od miejsca, w którym jest definiowana. Pomimo tego, że w wypadku definiowania zmiennych globalnych słowo kluczowe `var` jest opcjonalne, to jednak jego stosowanie należy do dobrego stylu programowania. Dzięki temu bardzo łatwo można się zorientować, w którym miejscu skryptu zmienna jest używana po raz pierwszy.

Choć w niektórych językach programowania operacje deklarowania zmiennej (sprowadzające się do zarezerwowania pamięci na jej wartość) oraz jej inicjalizacji (zapisania jej wartości w pamięci) są traktowane niezależnie, to jednak dynamiczne przydzielanie pamięci na wartości zmiennych wykorzystywane w JavaScripcie zwalnia programistów z konieczności zaprzątania sobie głowy zagadnieniami zarządzania pamięcią. W JavaScripcie zmienne są naprawdę „zmienne", gdyż kolejne przypisania mogą zmieniać nie tylko ich wartości, lecz także typ tych wartości (nie oznacza to wcale, że jest to przykład dobrego stylu programowania — jest to jedynie efekt uboczny luźnego typowania danych stosowanego w JavaScripcie).

Mówiąc o dobrym stylu programowania, warto zaznaczyć, że zazwyczaj zalecane jest definiowanie zmiennych globalnych na samym początku skryptu, podobnie jak zaleca się, by zmienne lokalne często wykorzystywane w funkcjach były definiowane na początku tych funkcji. Nawet jeśli w danej chwili wartość przypisywana zmiennej nie jest dostępna, można po prostu zadeklarować zmienną bez jej definiowania:

```
var mojaZmienna;
```

Jeśli należy zadeklarować więcej niż jedną zmienną, można to zrobić jednocześnie, oddzielając nazwy poszczególnych zmiennych przecinkami:

```
var mojaZmienna, licznik, franek, i, j;
```

W powyższej instrukcji można nawet zainicjalizować wartość zmiennej:

```
var mojaZmienna, licznik = 0, franek, i, j;
```

W przykładach przedstawionych w niniejszej książce zmienne są przeważnie (choć nie zawsze) definiowane i inicjalizowane na początku zakresu, w którym są dostępne. Nie jest niczym niezwykłym, że zmienne, które mają być wykorzystywane wewnątrz pętli `for`, są definiowane (przy wykorzystaniu słowa kluczowego `var`) bezpośrednio przed instrukcją pętli. Na przykład jeśli w pewnym fragmencie skryptu mają być wykorzystanie dwie zagnieżdżone pętle `for`, to bezpośrednio przed nimi można zdefiniować używane w nich liczniki:

```
var i, j;
for (i = 0; i < tablica1.length; i++) {
    for (j = 0; j < tablica2.length; j++) {
        ....
    }
}
```

Jest to oczywiście mój styl pisania skryptów. W przedstawionej sytuacji istnieje jeszcze inny powód, dla którego deklaracje zmiennych powinny znaleźć się poza pętlami. Otóż jeśli słowo kluczowe `var` zostałoby użyte w instrukcji inicjalizującej licznik pętli (np.: `var j = 0`), to pętla wewnętrzna cyklicznie wykonywałaby to słowo kluczowe podczas każdego wykonania pętli zewnętrznej. Interpreter JavaScriptu tworzy bowiem przestrzeń dla zmiennej za każdym razem, gdy odnajdzie słowo kluczowe `var`. Na szczęście interpreter określa także, która z wielokrotnie deklarowanych zmiennych jest tworzona w danej chwili, niemniej jednak operacje te niepotrzebnie wykorzystują zasoby. Zmienne należy deklarować jeden raz, a następnie, już bez żadnych ograniczeń, można inicjalizować je oraz zmieniać ich wartości. Dlatego w złożonych funkcjach, w których występują dwie lub więcej zewnętrznych pętli `for`, należy deklarować licznik pętli na samym początku funkcji, a na samym początku pętli jedynie inicjalizować jego wartość.

Jeśli zaś chodzi o wybieranie nazw dla zmiennych, to istnieją pewne jawne reguły oraz niejawne zwyczaje, którymi należy się kierować. Większe znaczenie mają oczywiście jawne reguły. Nazwy zmiennych nie mogą:

- zaczynać się od liczby;
- zawierać spacji oraz innych znaków odstępu;
- zawierać znaków przestankowych oraz innych symboli za wyjątkiem znaku podkreślenia;
- być zapisywane w cudzysłowach;
- odpowiadać zastrzeżonym słowom kluczowym języka ECMAScript (patrz dodatek C).

Stosowane przez programistów konwencje dotyczące nazw nadawanych zmiennym nie są obowiązujące, jak również nie mają żadnego wpływu na działanie skryptów. Niemniej jednak pomagają one w zachowaniu czytelności skryptu i ułatwiają jego utrzymanie, gdy po kilku miesiącach trzeba będzie sobie przypomnieć, co skrypt robi.

Nazwy zmiennych są podawane przede wszystkim po to, aby ułatwić określanie, jakiego rodzaju wartość jest przechowywana w danej zmiennej (w rzeczywistości zmienne są bardzo często nazywane identyfikatorami). Umieszczenie w skrypcie kilkunastu zmiennych o nazwach jedno- lub dwuliterowych nie ułatwi śledzenia poszczególnych wartości podczas analizy skryptu. Z drugiej strony istnieją czynniki związane z efektywnością działania (patrz receptura 4.8) przemawiające za unikaniem przesadnie długich nazw zmiennych. Im krótsza nazwa, tym lepsza, oczywiście o ile nie jest to skrót niemożliwy do odszyfrowania. Jeśli warto, musi być opisana za pomocą dwóch lub kilku słów, można je połączyć znakiem podkreślenia lub zaczynać poszczególne słowa (zaczynając od drugiego) od wielkiej litery (ta konwencja jest wykorzystywana w niniejszej książce). A zatem, obie nazwy zmiennych przedstawione w poniższym przykładzie, są dobre:

```
var czlonekZespolu = "Janek";
var czlonek_zespolu = "Janek";
```

Te reguły i pojęcia warto także stosować podczas określania nazw przypisywanych atrybutom `name` oraz `id` elementów HTML. Dzięki temu nie będzie żadnych problemów z używaniem tych nazw w odwołaniach.

W nazwach zmiennych uwzględniana jest wielkość liter. Z tego względu można (choć nie jest to zalecane) wykorzystać ten sam identyfikator zapisany literami o innej wielkości do przechowywania zupełnie nowej wartości. Jedna z konwencji, którą można wykorzystać, zaleca, by określić wszystkie zmienne, których wartości nie będą się zmieniać w całym skrypcie (czyli będą one traktowane jako stałe) i zapisać ich nazwy wielkimi literami. Przeglądarka Netscape 6 oraz jej nowsze wersje implementują przyszłe słowo kluczowe języka ECMAScript — `const` — którego można używać zamiast słowa `var` w celu definiowania stałych. Jak na razie, to słowo kluczowe nie jest obsługiwane w żadnych innych przeglądarkach, a zatem wciąż można używać zmiennych jako stałych i nie wprowadzać żadnych modyfikacji w ich wartościach.

JavaScript zapisuje wartości w zmiennych zarówno „przez odwołanie", jak i „przez wartość", przy czym użyty sposób zależy od typu danych. Jeśli zapisywana wartość jest prawdziwym obiektem (dowolnego typu — na przykład obiektem DOM, tablicą lub obiektem niestandardowym), to zmienna będzie zawierać odwołanie do danego obiektu. W takim wypadku zmiennej można używać jako zastępczego odwołania do obiektu:

```
var element = document.getElementById("mojaTablica");
var szerWyp = element.cellPadding;
```

Jeśli jednak dana jest wartością prostą (łańcuchem znaków, liczbą, wartością logiczną), to w zmiennej przechowywana jest sama wartość, bez jakiegokolwiek połączenia z oryginalnym obiektem, z którego została odczytana. Dlatego też zmienna `szerWyp` przedstawiona w powyższym przykładzie będzie zawierać wartość łańcuchową. Gdyby tej zmiennej przypisano nową wartość, to nie miałoby to żadnego wpływu na element `table`. Aby określić wartość właściwości obiektu, należy to zrobić za pośrednictwem odwołania:

```
element.cellPadding = "10";
```

Jeśli wartość właściwości obiektu sama jest obiektem, to w zmiennej zostanie zapisane odwołanie do tego obiektu bezpośrednio z nim powiązane:

```
var element = document.getElementById("mojaTablica");
var elementStyle = element.style;
elementStyle.fontSize = "18px";
```

Należy zachować dużą ostrożność, zapisując w zmiennych obiekty DOM. Może się wydawać, że zmienna jest jedynie kopią odwołania do obiektu, jednak wszelkie wprowadzane w niej modyfikacje mają wpływ na drzewo węzłów dokumentu.

Patrz także

Rozdziały 1., 2. oraz 3. omawiają zagadnienia związane z zapisywaniem w zmiennych wartości różnych typów — łańcuchów znaków, liczb, tablic oraz obiektów; receptura 4.8 przedstawia wpływ długości nazwy zmiennej na efektywność działania skryptów.

4.2. Tworzenie funkcji o konkretnej nazwie

NN 2 ***IE 3***

Problem

Chcemy zdefiniować funkcję, którą będzie można wywoływać w dowolnych instrukcjach umieszczonych na stronie.

Rozwiązanie

Aby stworzyć funkcję, która nie wymaga podawania żadnych argumentów, należy wykorzystać podstawowy format deklaracji funkcji:

```
function nazwaFunkcji() {
    // instrukcje
}
```

Jeśli funkcja ma odczytywać argumenty podawane w instrukcji wywołującej funkcję, nazwy zmiennych parametrycznych należy podać wewnątrz nawiasów umieszczonych za nazwą funkcji:

```
function nazwaFunkcji(zmParam1, zmParam2[, ..., zmParamN]) {
    // instrukcje
}
```

W definicji funkcji można umieścić dowolnie dużo unikalnych identyfikatorów zmiennych parametrycznych. Wewnątrz funkcji zmienne te staną się zmiennymi lokalnymi (przy ich tworzeniu domyślnie jest stosowane słowo kluczowe `var`). Zgodnie z konwencją luźnego typowania danych stosowaną w JavaScripcie, zmienne parametryczne mogą zawierać wartości dowolnych typów, określone w instrukcji wywołującej funkcję.

Nawiasy klamrowe otaczające instrukcje tworzące funkcję są wymagane wyłącznie wtedy, gdy w funkcji znajdują się dwie lub więcej instrukcji. Niemniej jednak stosowanie nawiasów klamrowych we wszystkich funkcjach, nawet tych, które składają się z jednej instrukcji, jest godne polecenia, gdyż poprawia czytelność kodu (konwencja ta jest wykorzystywana w niniejszej książce).

W przeważającej większości prezentowanych tu długich skryptów wykorzystywane są funkcje. Niektóre z nich mają argumenty, a inne nie. Wiele funkcji można także znaleźć w przykładach zaczerpniętych z Internetu, a w szczególności w recepturach zawierających biblioteki JavaScriptu, takie jak biblioteka DHTML API przedstawiona w recepturze 13.3.

Analiza

W JavaScripcie funkcja jest typem obiektu, a nazwa przypisywana funkcji staje się identyfikatorem danego obiektu (przy czym wielkość liter, jakimi nazwa jest zapisana, ma znaczenie). W efekcie słowa kluczowe JavaScriptu nie mogą być używane jako nazwy funkcji; podobnie nie należy nadawać tych samych nazw funkcjom oraz elementom globalnym, takim jak zmienne lub (w wypadku przeglądarki Internet Explorer) identyfikatory elementów HTML. Jeśli na jednej stronie pojawią się dwie funkcje o identycznych nazwach, to dostępna będzie jedynie ta z nich, która w kodzie źródłowym została umieszczona jako druga. JavaScript nie udostępnia możliwości przeciążania funkcji i metod, dostępnych w innych językach programowania, takich jak Java (gdzie metody o identycznych nazwach, różniące się ilością zmiennych parametrycznych, są traktowane jako niezależne metody).

W wywołaniu funkcji należy używać nawiasów:

```
mojaFunkcja();
mojaFunkcja2("witam", 42);
```

Czasami trzeba będzie zapisać odwołanie do funkcji we właściwości. Na przykład w wypadku zapisywania procedury obsługi zdarzeń we właściwości obiektu elementu (patrz rozdział 9.) instrukcja przypisania zawiera właśnie odwołanie do funkcji. Odwołanie takie składa się z samej nazwy funkcji, bez żadnych nawiasów, argumentów ani cudzysłowów:

```
document.onclick = mojaFunkcja;
```

Takie przypisanie jedynie umożliwia wywoływanie podanej funkcji w przyszłości.

W niektórych językach programowania rozróżniane są fragmenty kodu, które mogą być wykonywane samodzielnie, oraz fragmenty zwracające jakieś wartości. W JavaScripcie istnieje tylko jeden rodzaj funkcji. Jeśli funkcja zawiera instrukcję `return`, to funkcja ta będzie zwracać wartość, w przeciwnym wypadku wykonanie funkcji nie powoduje zwrócenia jakiejkolwiek wartości. Funkcje wykorzystywane w sposób, który sprawiłby, że w innych środowiskach byłyby one określane jako podprogramy, zazwyczaj zwracają wartości wyłącznie dlatego, że zostały stworzone w celu pobierania danych lub wykonywania jakichś obliczeń, a następnie zwracają uzyskany wynik do instrukcji, w której zostały wywołane. Gdy funkcja zwraca wartość, jej wywołanie jest przekształcane na

wartość, którą można bezpośrednio zapisać w zmiennej lub użyć jako argumentu wywołania innej funkcji lub metody. Możliwość tę przedstawiłem w recepturze 15.6. Jej zadaniem jest wyświetlanie informacji o porze dnia (rano, popołudnie, wieczór) w powitaniu generowanym podczas pobierania strony. Funkcja o nazwie `okreslPoreDnia()` (zdefiniowana w nagłówku strony) określa bieżący czas i na jego podstawie zwraca łańcuch znaków zawierający określenie pory dnia:

```
function okreslPoreDnia() {
    var data = new Date();
    var godzina = data.getHours();
    if (godzina < 18) {
        return "Dzień dobry";
    } else {
        return "Dobry wieczór";
    }
}
```

Powyższa funkcja jest wywoływana jako argument wywołania metody `document.write()`, umieszczającej tekst na wyświetlanej stronie WWW:

```
<script language="JavaScript" type="text/javascript">
<!--
document.write(okreslPoreDnia() + " i witaj")
//-->
</script>
<noscript>Witamy</noscript>
 w GiantCo.
```

Nie jest ważne, czy ilość argumentów przekazanych w wywołaniu funkcji będzie taka sama jak liczba zdefiniowanych w niej zmiennych parametrycznych. Na przykład jeśli funkcja jest wywoływana w dwóch różnych miejscach skryptu, a w każdym z nich w jej wywołaniu jest podawana inna ilość argumentów, to wewnątrz niej można się odwoływać do argumentów za pośrednictwem właściwości `arguments`, a nie zmiennych parametrycznych:

```
function mojaFunkcja() {
    for (var i = 0; i < mojaFunkcja.arguments.length; i++) {
        // każdy element tablicy arguments jest wartością jednego
        // z argumentów, a ich kolejność odpowiada kolejności użytej
        // w wywołaniu
    }
}
```

Typowe funkcje (za wyjątkiem funkcji zagnieżdżonych, opisanych w recepturze 4.3) umieszczane są w globalnym kontekście okna, w którym wyświetlona jest strona zawierająca definicję funkcji. Do takich funkcji globalnych, podobnie jak do zmiennych globalnych, można się odwoływać w skryptach umieszczonych w innych oknach i ramkach. Przykłady odwołań do zawartości innych ramek można znaleźć w rozdziale 7., w części „Ramki jako obiekty okien".

Wielkość funkcji zależy wyłącznie od stylu programowania. Ze względu na łatwość testowania i pielęgnacji celowe może być podzielenie długiej funkcji na fragmenty, które same są implementowane jako podprogramy zwracające wartość lub działają jako sekwencja funkcji. Jeśli zauważymy, że w dużej funkcji pewien fragment kodu jest używany kilka razy, to będzie się on doskonale nadawać do usunięcia i zaimplementowania w formie nowej funkcji, która w większej funkcji będzie kilkakrotnie wywoływana.

Kolejną decyzją związaną ze stylem programowania, którą może podejmować autor skryptów, jest miejsce zapisu nawiasów klamrowych. W tej książce zastosowano zasadę, że otwierający nawias klamrowy zaczyna się w tym samym wierszu, w którym została podana nazwa funkcji; a nawias zamykający jest tak samo wcięty jak deklaracja funkcji. Niemniej jednak otwierający nawias klamrowy można także zapisywać w wierszu poniżej nazwy funkcji i ewentualnie wyrównać do początku wiersza:

```
function mojaFunkcja()
{
    // instrukcje
}
```

Niektórzy programiści uważają, że ten sposób zapisu ułatwia synchronizację par nawiasów klamrowych. W wypadku funkcji zawierających jedną instrukcję można ją zapisywać w tym samym wierszu, w którym została umieszczona nazwa funkcji:

```
function mojaFunkcja() { // instrukcja }
```

Tworząc skrypty, można wykorzystać styl kodowania, który wydaje się nam najbardziej sensowny.

Patrz także

Receptura 4.1 zawiera informacje o zmiennych przekazywanych „przez odwołanie" oraz „przez wartość" — informacje te dotyczą także argumentów funkcji; receptura 4.3 zawiera informacje na temat zagnieżdżania funkcji.

4.3. Zagnieżdżanie funkcji

NN 4 ***IE 4***

Problem

Chcemy stworzyć funkcję należącą wyłącznie do jednej, innej funkcji.

Rozwiązanie

Poczynając od przeglądarek IE 4 oraz NN4, istnieje możliwość zagnieżdżania jednej funkcji wewnątrz drugiej, wykorzystując przy tym składnię przedstawioną na poniższym przykładzie:

```
function mojaFunkcjaA() {
    // instrukcje
    function mojaFunkcjaB() {
        // kolejne instrukcje
    }
}
```

W tym wypadku funkcja zagnieżdżona może być używana wyłącznie w instrukcjach umieszczonych wewnątrz funkcji zewnętrznej (należy jednak przeczytać informacje dotyczące przeglądarki Netscape 6 oraz jej nowszych wersji podane w części „Analiza"). Instrukcje umieszczone w funkcji zagnieżdżonej mają dostęp zarówno do zmiennych deklarowanych w funkcji zewnętrznej, jak i do zmiennych globalnych. Z kolei instrukcje umieszczone w funkcji zewnętrznej nie mają dostępu do zmiennych utworzonych w funkcji zagnieżdżonej.

Analiza

Podstawowym celem tworzenia funkcji zagnieżdżonych jest możliwość hermetyzacji czynności związanych z funkcją zewnętrzną poprzez tworzenie podprogramów prywatnych dla tej funkcji zewnętrznej. Ponieważ funkcja zagnieżdżona nie jest bezpośrednio dostępna w przestrzeni globalnej, jej nazwę można wykorzystać zarówno w tej przestrzeni globalnej, jak i wewnątrz dowolnej innej funkcji.

Netscape 6 (oraz nowsze wersje tej przeglądarki) w logiczny i wygodny sposób rozszerzają środowisko funkcji zagnieżdżonych — otóż funkcja zagnieżdżona staje się metodą obiektu funkcji, wewnątrz której została zdefiniowana. Istnieją pewne interesujące efekty uboczne związane ze sposobem implementacji tej możliwości. Przeanalizujmy przykład poniższych funkcji testowych:

```
function mojaFunkcjaA() {
    var wartoscA = "A";
    mojaFunkcjaB();
    function mojaFunkcjaB() {
        var wartoscB = "B";
        alert(wartoscB);
        alert(wartoscA);
    }
}
```

Wywołanie funkcji `mojaFunkcjaA()` z głównego poziomu skryptu spowoduje, że zostanie wywołana funkcja zagnieżdżona, która posłusznie wyświetli wartości dwóch zmiennych, z których jedna została zdefiniowana w funkcji zewnętrznej, a druga — w wewnętrznej. Przeglądarka Internet Explorer działa tak samo i zgodnie z naszymi oczekiwaniami. Jednak w Netscape 6 istnieje możliwość wywołania funkcji zagnieżdżonej z głównego poziomu skryptu, wykorzystując przy tym funkcję zewnętrzną:

```
mojaFunkcjaA.mojaFunkcjaB();
```

Ponieważ w tym wypadku funkcja zewnętrzna nie jest wykonywana, zmienna `wartoscA` nie zostaje zainicjalizowana. W momencie wykonania funkcji `mojaFunkcjaB()` wartość zmiennej `wartoscB` jest wyświetlana poprawnie, lecz jest niezdefiniowana (`undefined`). Oznacza to, że dostęp do zmiennych zdefiniowanych w funkcji zewnętrznej jest możliwy wyłącznie w tym wypadkach, gdy funkcja zagnieżdżona jest wywoływana przez tę funkcję zewnętrzną.

Patrz także

Receptura 4.1 zawiera informacje o zakresie zmiennych.

4.4. Tworzenie funkcji anonimowych

NN 4 IE 4

Problem

Chcemy zdefiniować funkcję w formie wyrażenia, które można by na przykład przekazać jako argument wywołania konstruktora obiektu lub przypisać wybranej metodzie.

Rozwiązanie

W takich wypadkach można skorzystać z alternatywnego sposobu definiowania funkcji, który nie wiąże się z koniecznością tworzenia funkcji o konkretnych nazwach (przedstawionych w recepturze 4.2). Rozwiązanie to tworzy tak zwane *funkcje anonimowe*, które od zwyczajnych funkcji różnią się jedynie brakiem identyfikatora. Oto składnia służąca do tworzenia funkcji anonimowych:

```
var jakiesOdwolanie = function() { instrukcje };
```

Instrukcje umieszczone wewnątrz nawiasów klamrowych to zwyczajna sekwencja instrukcji JavaScriptu oddzielonych od siebie średnikami. W razie potrzeby można także zdefiniować zmienne parametryczne:

```
var jakiesOdwolanie = function(zmParam1[, ... zmParamN]) { instrukcje }
```

Funkcje anonimowe należy wywoływać za pomocą odwołania:

```
jakiesOdwolanie();
```

Dyskusja

W wyniku utworzenia funkcji anonimowej zwracany jest obiekt typu `function`. Z tego względu prawą stronę instrukcji można przypisać do dowolnej instrukcji przypisania, w której oczekiwane jest odwołanie do funkcji (nazwa funkcji bez pary nawiasów). Aby zademonstrować te zagadnienia, stworzymy nową wersję uproszczonego konstruktora obiektów z receptury 3.8. Przykład zaczyna się od definicji zwyczajnej funkcji wywoływanej jako metoda czterech obiektów tworzonych w skrócony sposób:

```
function pokazDane() {
    alert("Pracownik " + this.imie + " ma " + this.wiek + " lat.");
}
var pracownicyBD = [{imie:"Alicja", wiek:23, pokaz:pokazDane},
                    {imie:"Franek", wiek:32, pokaz:pokazDane},
                    {imie:"Janka", wiek:28, pokaz:pokazDane},
                    {imie:"Stefan", wiek:24, pokaz:pokazDane}];
```

Warto zwrócić uwagę na to, w jaki sposób w konstruktorze obiektów odwołanie do metody `pokazDane()` jest przypisywane do metody o nazwie `pokaz`. Wywołanie tej metody dla dowolnego z obiektów jest realizowane w następujący sposób:

```
pracownicyBD[2].pokaz();
```

Na potrzeby naszego przykładu do pierwszego z obiektów przypiszemy funkcję anonimową. Funkcja anonimowa jest dostosowana do potrzeb pierwszego obiektu i zastępuje odwołanie do funkcji `pokazDane()`:

```
var pracownicyBD = [{imie:"Alicja", wiek:23,
                     pokaz:function() {alert("Dane o Alicji nie są dostępne dla
                     ↳ogółu.")}},
                    {imie:"Franek", wiek:32, pokaz:pokazDane},
                    {imie:"Janka", wiek:28, pokaz:pokazDane},
                    {imie:"Stefan", wiek:24, pokaz:pokazDane}];
```

Teraz wywołanie metody `pracownicyBD[0].pokaz()` spowoduje wyświetlenie specjalnego okienka dialogowego. Wynika to z faktu, iż funkcja `pokazDane()` została zastąpiona przez funkcję anonimową. W ten sposób uniknęliśmy konieczności tworzenia zewnętrznej funkcji mającej własny identyfikator, który działałby wyłącznie jako pośrednik pomiędzy nazwą metody `pokaz` oraz instrukcjami wykonywanymi w momencie wywołania tej metody.

Patrz także

Receptura 4.2 pokazuje sposób tworzenia tradycyjnych funkcji o określonej nazwie.

4.5. Opóźnianie wywołania funkcji

NN 2 ***IE 3***

Problem

Chcemy wykonać funkcję w niedługiej przyszłości, po upłynięciu zadanego czasu.

Rozwiązanie

Aby jednokrotnie wywołać funkcję po upłynięciu zadanej ilości milisekund, należy użyć metody `window.setTimeout()`. Metoda ta ustawia licznik czasu, który uruchomi wskazaną funkcję. Funkcja, którą należy wykonać jest określana za pomocą łańcucha znaków zawierającego jej nazwę oraz parę nawiasów, jak pokazano na poniższym przykładzie:

```
var idOdliczania = setTimeout("mojaFunkcja()", 5000);
```

Metoda `setTimeout()` zwraca identyfikator „operacji odliczania", który należy zapisać w zmiennej globalnej. Jeśli w dowolnej chwili przed uruchomieniem funkcji konieczne będzie przerwanie operacji, będzie to można zrobić, wywołując metodę `clearTimeout()` i przekazując do niej zapamiętany wcześniej identyfikator:

```
clearTimeout(idOdliczania);
```

Po uruchomieniu odliczania dalsze operacje związane z realizacją skryptu mogą być normalnie wykonywane, dlatego też dobrym rozwiązaniem jest umieszczanie wywołania metody `setTimeout()` jako ostatniej instrukcji w funkcji.

Analiza

Jest bardzo ważne, aby zrozumieć, czego metoda `setTimeout()` nie robi — otóż nie wstrzymuje ona realizacji całego skryptu na podany czas.

Zamiast tego metoda ta ustawia wewnętrzny licznik czasu, który powoduje wykonanie funkcji o podanej nazwie po upłynięciu zadanej ilości milisekund. Na przykład jeśli tworzymy pokaz slajdów, w którym należy przejść na inną stronę, jeśli w ciągu 15 sekund użytkownik nie wykona żadnej operacji, to początkowo odliczanie można rozpoczynać w procedurze obsługi zdarzeń `onload` oraz w funkcji `rozpocznijOdliczanie()`:

```
var idOdliczania;
function przejdzNaStrone() {
    location.href = "slajd3.html";
}
function rozpocznijOdliczanie() {
    clearTimeout(idOdliczania);
    idOdliczania = setTimeout("przejdzNaStrone()", 15000);
}
```

W takim wypadku zapewne chcielibyśmy także określić, na przykład procedurę obsługi zdarzeń `onmousemove`, aby każda operacja wykonana przez użytkownika za pomocą myszy powodowała rozpoczęcie odliczania od początku:

```
window.onmousemove = rozpocznijOdliczanie;
```

Funkcja `rozpocznijOdliczanie()` automatycznie usuwa wcześniejszą operację zanim funkcja `przejdzNaStrone()` zostanie wywołana, a następnie rozpoczyna nową operację odliczania.

Jeśli funkcja wywoływana z opóźnieniem wymaga podania argumentów, to można wygenerować łańcuch znaków zawierający niezbędne wartości, i to nawet w wypadku, gdy wewnątrz funkcji wartości te są przechowywane w zmiennych. Jednak wartości te nie mogą być odwołaniami do obiektów — to bardzo ważne. Argumenty funkcji muszą mieć postać, która pozwoli na zapisanie ich w formie łańcucha znaków, gdyż pierwszym argumentem wywołania metody `setTimeout()` musi być właśnie łańcuch. Receptura 8.4 przedstawia sposób przekazywania nazw obiektów DOM związanych z formularzami, stanowiący rozwiązanie umożliwiające przekazywanie odwołań do obiektów. Najtrudniejszym zadaniem jest odpowiednie zapisanie cudzysłowów:

```
function czyToAdresEmail(element) {
    var str = element.value;
    var wr = /^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/;
    if (!str.match(wr)) {
        alert("Sprawdź format zapisu adresu poczty elektronicznej.");
        setTimeout("fokusDoElementu('" + element.form.name + "', '" +
                   ↳element.name + "')", 0);
```

```
            return false;
        } else {
            return true;
        }
    }
```

W powyższym przykładzie funkcja `fokusDoElementu()` wymaga przekazania dwóch argumentów koniecznych do określenia poprawnego odwołania zarówno do obiektu `form`, jak i `input`. Ponieważ pierwszym argumentem metody `setTimeout()` musi być łańcuch znaków, konieczne jest wymuszenie, by argumenty wywołania metody `fokusDoElementu()` również były traktowane jako łańcuchy znaków. Do tego celu służą apostrofy umieszczane w rozbudowanej instrukcji konkatenacji. (Zerowa wartość czasu podana w wywołaniu metody `setTimeout()` jest poprawnym rozwiązaniem dla danej aplikacji. Więcej informacji na ten temat można znaleźć w recepturze 8.4.)

Przy próbie znalezienia możliwości stworzenia prawdziwego opóźnienia pomiędzy instrukcjami wykonywanymi wewnątrz funkcji lub pomiędzy fragmentami funkcji okazuje się, że JavaScript nie udostępnia żadnych takich rozwiązań porównywalnych z poleceniami stosowanymi w innych językach programowania. Ten sam efekt można jednak uzyskać, dzieląc oryginalną funkcję na kilka mniejszych, przy czym każda z nich będzie odpowiadać fragmentowi funkcji, który należy wykonać z opóźnieniem. Funkcje należy następnie połączyć ze sobą, umieszczając na końcu pierwszej metodę `setTimeout()` wywołującą kolejną z funkcji po upłynięciu zadanego czasu:

```
function funkcjaGlowna() {
    // instrukcje poczatkowe
    setTimeout("funkcjaCzesc2()", 10000);
}
function funkcjaCzesc2() {
    // instrukcje początkowe
    setTimeout("funkcjaKoniec()", 5000);
}
function funkcjaKoniec() {
    // instrukcje kończące przetwarzanie
}
```

W kodzie skryptu powiązane ze sobą funkcje nie muszą być umieszczone bezpośrednio jedna za drugą. Jeśli wszystkie te funkcje muszą operować na tym samym zbiorze wartości, to wartości te można przekazywać z jednej funkcji do drugiej jako argumenty (zakładając, że nie będą one obiektami) bądź zapisać w zmiennych globalnych. Jeśli wartości są ze sobą powiązane, może to przemawiać za zdefiniowaniem niestandardowego obiektu i zapisaniem tych wartości w jego właściwościach opatrzonych odpowiednimi nazwami. Dzięki temu łatwiej będzie już na pierwszy rzut oka zorientować się, w jaki sposób poszczególne segmenty funkcji wykorzystują wartości lub jak je modyfikują.

Kolejna metoda JavaScriptu — `setInterval()` — działa podobnie do metody `setTimeout()`, lecz cyklicznie wywołuje wskazaną funkcję, aż jawnie każemy jej przestać (przy użyciu metody `clearInterval()`). Drugi argument wywołania tej metody (liczba całkowita określająca ilość milisekund) określa, co jaki czas będzie wywoływana wskazana funkcja.

Patrz także

Receptura 8.4 przedstawia sposób wykorzystania metody `setTimeout()` w celu synchronizacji wykonywania skryptu; receptura 10.11 przedstawia sekwencję trzech funkcji inicjalizacyjnych połączonych ze sobą za pomocą wywołań metody `setTimeout()`, zapewniającej odpowiednią synchronizację ich wywołań; receptura 12.3 przedstawia przykład wykorzystania niezależnej zmiennej licznikowej, zapewniającej, że funkcja zostanie wywołana ściśle określoną ilość razy; receptury 13.9 oraz 13.10 pokazują sposoby wykorzystania metody `setInterval()` do tworzenia animacji.

4.6. Wykonywanie różnych kodów w zależności od warunku

NN 4 IE 4

Problem

Chcemy, aby skrypt wykonał fragment kodu w zależności od wartości zewnętrznych, takich jak wartość logiczna, dane wpisane przez użytkownika w polu tekstowym lub wartość wybrana z elementu `select`.

Rozwiązanie

Należy użyć konstrukcji sterujących `if`, `if-else` lub `switch`, aby określić ścieżkę wykonywania przebiegającą przez wybrany fragment skryptu. Jeśli należy wykonać fragment skryptu w wypadku gdy będzie sprawdzony tylko jeden warunek, można wykorzystać prostą instrukcję `if`, której wyrażenie warunkowe testuje zadany warunek:

```
if (warunek) {
    // instrukcje wykonywane, gdy warunek jest spełniony (ma wartość true)
}
```

Aby wykonać wybrany fragment skryptu, gdy będzie spełniony określony warunek, a inny fragment we wszystkich pozostałych sytuacjach, należy użyć instrukcji `if-else`:

```
if (warunek) {
    // instrukcje wykonywane, gdy warunek jest spełniony (ma wartość true)
} else {
    // instrukcje wykonywane, gdy warunek nie jest spełniony
}
```

Można także bardziej precyzyjnie określić, kiedy klauzula `else` ma być wykonana, sprawdzając w niej dodatkowe warunki:

```
if (warunekA) {
    // instrukcje wykonywane, gdy warunekA jest spełniony (ma wartość true)
} else if (warunekB) {
    // instrukcje wykonywane, gdy warunekB jest spełniony
```

```
} else {
    // instrukcje wykonywane, gdy nie jest spełniony ani warunekA, ani warunekB
}
```

W wypadku istnienia wielu warunków można rozważyć wykorzystanie instrukcji `switch`, o ile testy bazują na wartości łańcuchowej lub liczbowej:

```
switch (wyrazenie) {
    case wartoscA:
        // instrukcje wykonywane, gdy wyrazenie przyjmie wartoscA
        break;
    case wartoscB:
        // instrukcje wykonywane, gdy wyrazenie przyjmie wartoscB
        break;
    ...
    default:
        // instrukcje wykonywane, gdy wyrazenie przyjmie wartość, która
        // nie została podana w żadnej klauzuli case
}
```

Instrukcja `break` umieszczana na końcu każdej z klauzul `case` zapewnia, że nie zostanie wykona żadna z klauzul `case` umieszczonych w dalszej części instrukcji ani (opcjonalna) klauzula `default`.

Analiza

Wyrażenie warunkowe umieszczane w instrukcjach `if` oraz `if-else` są wyrażeniami zwracającymi wartości logiczne (Boolean) — `true` lub `false`. Zazwyczaj w wyrażeniach tego typu wykorzystywane są operatory porównania (==, ===, !=, !==, <, <=, > oraz >=) określające relacje pomiędzy dwiema wartościami.

W przeważającej większości wypadków porównywana jest zmienna z pewną stałą lub wartością znaną:

```
var miesiac = mojaData.getMonth();
if (miesiac == 1) {
    // wartość 1 liczona od zera oznacza drugi miesiąc - czyli luty
    dlugoscMies = okreslDlugoscMiesPrzestepnego(mojaData);
} else {
    dlugoscMies = okreslDlugoscMiesiaca(miesiac);
}
```

JavaScript udostępnia pewien skrócony sposób obliczania wartości wyrażeń warunkowych. Jest on przydatny w sytuacjach, gdy chcemy wykonać pewien fragment kodu w zależności od istnienia określonej właściwości lub obiektu. Tabela 4.1 zawiera warunki, które automatycznie przyjmują wartość `true` lub `false`, jeśli zostaną umieszczone wewnątrz nawiasów wyrażenia warunkowego. Na przykład dzięki temu, że wyrażenie warunkowe przyjmuje wartość `true`, jeśli istnieje podany obiekt, można utworzyć następującą instrukcję:

```
if (mojObiekt) {
    // mojObiekt istnieje, można go więc wykorzystać
}
```

Tabela 4.1. Odpowiedniki wyrażeń warunkowych

Prawda (ang.: true)	**Fałsz (ang.: false)**
W łańcuchu znaków występuje jeden lub więcej znaków	Łańcuch znaków jest pusty
Liczba jest różna od 0	`0`
Wartość jest różna od `null`	`null`
Istnieje obiekt, do którego odwołuje się wyrażenie	Obiekt, do którego odwołuje się wyrażenie, nie istnieje
Właściwość obiektu istnieje, a jej wartość jest łańcuchem znaków zawierającym co najmniej jedną literę lub liczbę różną od 0	Właściwość obiektu nie istnieje lub jej wartość jest pustym łańcuchem znaków bądź jest równa 0

Sprawdzając istnienie właściwości obiektu (w tym także właściwości globalnego obiektu `window`), zawsze należy rozpoczynać wyrażenie od odwołania do obiektu, jak w poniższym przykładzie:

```
if (window.innerHeight) { ... }
```

Dużą uwagę należy także zachować w wypadkach testowania istnienia właściwości, jeśli jej wartość może być pustym łańcuchem znaków lub zerem. W takich wypadkach wyrażenie warunkowe także przyjmuje wartość `false`, nawet jeśli właściwość istnieje. Dlatego lepszym rozwiązaniem jest sprawdzanie typu wartości zapisanej w testowanej właściwości. Jeśli nie mamy pewności, jakiego typu danych ma być ta wartość, to można ją porównywać ze stałą `undefined`:

```
if (typeof mojObiekt.mojaWlasciwosc != "undefined") {
    // mojaWlasciwosc istnieje i została jej przypisana jakaś wartość
}
```

Jeśli istnieje prawdopodobieństwo, że nie będzie istnieć ani obiekt, ani jego właściwość, to należy zgrupować wyrażenia warunkowe testujące istnienie obu tych elementów. Najpierw należy sprawdzać istnienie obiektu, a potem właściwości. Jeśli okaże się, że obiekt nie istnieje, to test sprawdzający istnienie właściwości nie zostanie już wykonany.

```
if (mojObiekt && typeof mojObiekt.mojaWlasciwosc != "undefined") {
    // mojObiekt istnieje, podobnie jak mojaWlasciwosc
}
```

Gdyby w pierwszej kolejności sprawdzane było istnienie właściwości, to cały test zakończyłby się niepowodzeniem, a dodatkowo skrypt zgłosiłby błąd, gdyż wyrażenie odwołujące się do obiektu byłoby błędne.

JavaScript udostępnia także skróconą formę zapisu, umożliwiającą pominięcie nawiasów klamrowych w prostych instrukcjach przypisania, które zachowują się różnie w zależności od warunku. Składnia ta została przedstawiona poniżej:

```
var mojaZmienna = (warunek) ? wartosc1 : wartosc2;
```

Jeśli warunek przyjmie wartość `true`, to całe wyrażenie zapisane po prawej stronie operatora przypisania zwróci pierwszą wartość (*`wartosc1`*); w przeciwnym wypadku zwrócona zostanie druga wartość. Na przykład:

```
var kolorTla = (temperatura > 100) ? "red" : "blue";
```

W kilku recepturach prezentowanych w kolejnych rozdziałach ten skrótowy zapis jest często wykorzystywany, i to nawet w konstrukcjach zagnieżdżonych. Na przykład:

```
var kolorTla = (temperatura > 100) ? "red" : ((temperatura < 80) ? "blue" :
↳"yellow");
```

Powyższe wyrażenie jest odpowiednikiem znacznie dłuższej, bardziej czytelnej lecz mniej eleganckiej złożonej instrukcji warunkowej:

```
var kolorTla ;
if (temperatura > 100) {
    kolorTla = "red";
} else if (temperatura < 80) {
    kolorTla = "blue";
} else {
    kolorTla = "yellow";
}
```

W sytuacjach gdy istnieje wiele możliwych ścieżek wykonywania skryptu, a to, która z nich zostanie wykonana, zależy nie od wyrażenia warunkowego lecz od wartości pewnego wyrażenia, optymalnym rozwiązaniem jest użycie instrukcji `switch`. W poniższym przykładzie zakładamy, że formularz zawiera element `select` pozwalający użytkownikowi na wybór wielkości produktu. Po dokonaniu wyboru procedura obsługi zdarzeń `onchange` zdefiniowana w elemencie `select` wywołuje funkcję, która wyświetla w polu tekstowym odpowiednią cenę:

```
function okreslCene(formularz) {
    var listaWielkosci = formularz.wyborWielkosci;
    var wybranyElem =
    ↳listaWielkosci.options[listaWielkosci.selectedIndex].value;
    switch (wybranyElem) {
        case "mala" :
            formularz.cena.value = "44.95";
            break;
        case "srednia" :
            formularz.cena.value = "54.97";
            break;
        case "duza" :
            formularz.cena.value = "64.99";
            break;
        default:
            formularz.cena.value = "Wybierz wielkość";
    }
}
```

Jeśli wyrażenie użyte w instrukcji `switch` zawsze przyjmuje jedną z wartości podanych w klauzulach `case`, to można pominąć klauzulę `default`, jednak podczas tworzenia skryptu warto ją stosować jako rodzaj „zaworu bezpieczeństwa” ostrzegającego przed możliwymi błędami spowodowanymi pojawieniem się nieoczekiwanej wartości.

Patrz także

Większość receptur zamieszczonych w rozdziale 15., wykorzystuje skrócone instrukcje warunkowe do obsługi różnych modeli obsługi zdarzeń.

4.7. Elegancka obsługa błędów w skryptach

NN 6 IE 5

Problem

Chcemy obsługiwać wszystkie błędy pojawiające się w skryptach w taki sposób, aby użytkownicy ich nie zauważali, czyli chcemy sprawić, by przeglądarka nie wyświetlała komunikatów o błędach.

Rozwiązanie

Istnieje pewien szybki, lecz niezbyt elegancki sposób uniknięcia generacji komunikatów o błędach pojawiających się w skryptach, działający także w starszych wersjach przeglądarek. Polega on na dodaniu poniższego fragmentu kodu do skryptu umieszczonego w nagłówku strony:

```
function nicNieRob() {return true;}
window.onerror = nicNieRob;
```

Rozwiązanie to nie zapobiegnie generowaniu błędów pojawiających się podczas kompilacji skryptów (na przykład błędów syntaktycznych wykrywanych przez interpreter podczas ładowania strony). Co więcej, nie udostępni ono programistom informacji o tym, w których miejscach kodu pojawiły się błędy. Można je stosować jedynie w wypadkach, gdy konieczne jest opublikowanie strony przed jej dokładnym przetestowaniem; podczas testowania strony powyższy fragment kodu należy usunąć.

W przeglądarkach IE 5 oraz Netscape 6 (oraz ich nowszych wersjach) można stosować bardziej formalny sposób obsługi błędów (wyjątków). Aby starsze wersje przeglądarek nie generowały żadnych komunikatów o błędach po napotkaniu specjalnych instrukcji wykorzystywanych do obsługi wyjątków, instrukcje te należy umieszczać w bloku `<script>`, którego atrybut `language` informuje o wykorzystaniu języka JavaScript 1.5 (`language="JavaScript1.5"`).

Instrukcje, które mogą powodować (zgłaszać) wyjątki, należy umieszczać wewnątrz bloków `try-catch`. Wykonywane instrukcje zapisywane są w bloku `try`, natomiast blok `catch` przetwarza wszelkie zgłaszane wyjątki:

```
<script type="text/javascript" language="JavaScript1.5">
function mojaFunkcja() {
    try {
        // instrukcja (instrukcje), które w wypadku spełnienia pewnych warunków
```

```
            // mogą zgłaszać wyjątki
        }
        catch(e) {
            // instrukcje obsługujące wyjątki (obiekt błędu jest zapisywany
               ↳w zmiennej e)
        }
    }
    </script>
```

Nawet jeśli w bloku `catch` nie są wykonywane żadne operacje, to wyjątek zgłoszony w bloku `try` nie spowoduje żadnych fatalnych konsekwencji. Kolejne instrukcje umieszczone w funkcji, jeśli takie w ogóle są, zostaną wykonane w standardowy sposób — oczywiście pod warunkiem, że nie zależą one od wartości tworzonych w bloku `try`. Ewentualnie można pominąć dalszy kod funkcji i elegancko zakończyć jej wykonywanie, umieszczając wewnątrz bloku `catch` instrukcję `return`.

Analiza

Każdy zgłoszony wyjątek generuje kopię obiektu `Error`. Odwołanie do tego obiektu jest przekazywane do bloku `try-catch` jako argument klauzuli `catch`. Fragment skryptu umieszczony wewnątrz klauzuli catch może sprawdzić właściwości tego obiektu, aby poznać przyczyny błędu. Jak na razie, w oficjalnym standardzie języka ECMAScript obiekt `Error` ma jedynie kilka właściwości, jednak w niektórych przeglądarkach zostały zaimplementowane właściwości niestandardowe, zawierające te same informacje, które można zobaczyć w komunikatach o błędach. W tabeli 4.2 przedstawione zostały informacyjne właściwości obiektu `Error` oraz przeglądarki, w których właściwości te są dostępne.

Tabela 4.2. Właściwości obiektu Error

Właściwość	IE/Win	IE/Mac	NN	Opis
`description`	5	5	brak	Słowny opis błędu
`fileName`	brak	brak	6	Adres URL pliku zawierającego skrypt, który spowodował zgłoszenie błędu
`lineNumber`	brak	brak	6	Numer wiersza kodu źródłowego skryptu, w którym powstał błąd
`message`	5.5	brak	6	Słowny opis błędu (ECMA)
`name`	5.5	brak	6	Typ błędu (ECMA)
`number`	5	5	brak	Numer błędu w przeglądarkach firmy Microsoft

Z założenia, komunikaty o błędach nigdy nie mają być prezentowane użytkownikom. Właściwości `description` oraz `message` obiektu błędu można wykorzystać we własnych skryptach, aby określić, w jaki sposób obsłużyć wyjątek. Niestety, zdarza się, że w różnych przeglądarkach komunikaty dotyczące tego samego błędu nie są identyczne. Na przykład próba odwołania się do niezdefiniowanego obiektu spowoduje, że Internet Explorer wyświetli komunikat:

```
Brak definicji 'mojObiekt'
```

Z kolei w przeglądarce Netscape 6 oraz jej nowszych wersjach komunikat dotyczący tego samego błędu będzie mieć postać:

```
mojObiekt is not defined
```

Ze względu na te różnice tworzenie skryptów obsługujących błędy, które mogłyby działać w różnych przeglądarkach, jest trudnym zadaniem. W powyższym przykładzie można by spróbować rozwiązać problem, sprawdzając, czy łańcuchy znaków (przy użyciu wyrażeń regularnych) zapisane we właściwościach obiektu zawierają łańcuch znaków `"defin"`:

```
try {
    window.onmouseover = sledzPolozenie;
}
catch(e) {
    var kmk = (e.message) ? e.message : e.description;
    if (/sledzPolozenie/.exec(kmk) && /defined/.exec(kmk)) {
        // funkcja sledzPolozenie nie jest dostępna na stronie
    }
}
```

Można także samodzielnie zgłaszać wyjątki, traktując to rozwiązanie jako sposób „wbudowania" obsługi wyjątków we własne skrypty. Poniższa funkcja jest zmodyfikowaną wersją funkcji weryfikującej informacje podawane w formularzu, sprawdzającą, czy pole zawiera wyłącznie liczbę. Kod umieszczony w klauzuli `try` sprawdza, czy podana wartość nie jest poprawna. Jeśli faktycznie wartość będzie błędna, to skrypt utworzy własną kopię obiektu `Error`, a następnie zgłosi wyjątek za pomocą instrukcji `throw`. Oczywiście zgłoszony wyjątek jest od razu przechwytywany przez klauzulę `catch`, która wyświetla komunikat informujący o błędzie, po czym przenosi fokus do sprawdzanego pola:

```
function przetworzLiczbe(poleForm) {
    try {
        var wartosc = parseInt(poleForm.value, 10);
        if (isNaN(wartosc)) {
            var kmk = "Można wpisywać wyłącznie liczby.";
            var err = new Error(kmk);
            if (!err.message) {
                err.message = kmk;
            }
            throw err;
        }
        // tu można bezpiecznie przetwarzać podaną liczbę
    }
    catch (e) {
        alert(e.message);
        poleForm.focus();
        poleForm.select();
    }
}
```

Funkcje sprawdzające tego typu są wywoływane zarówno w procedurach obsługi zdarzeń `onchange` pól tekstowych, jak również w funkcjach weryfikujących poprawność całego formularza, przedstawionych w rozdziale 8.

4.8. Poprawianie efektywności działania skryptów

NN 2 IE 3

Problem

Chcemy przyspieszyć działanie powolnego skryptu.

Rozwiązanie

W wypadku przetwarzania niewielkich fragmentów kodu interpreter JavaScriptu zazwyczaj realizuje ten kod szybko. Jeśli jednak na stronie zostanie umieszczony bardzo duży, zagnieżdżony kod, to będzie można zauważyć pewne opóźnienia w jego działaniu, nawet jeśli wszystkie dane zostaną przez przeglądarkę pobrane.

Poniżej przedstawiono kilka przydatnych porad, które mogą pomóc w rozwiązaniu problemów związanych z „wąskimi gardłami" w skryptach:

- Należy unikać stosowania funkcji `eval()`.
- Należy unikać stosowania instrukcji `with`.
- Należy unikać wielokrotnego przetwarzania wyrażeń.
- Warto używać symulowanych tablic mieszających do odnajdywania elementów w dużych tablicach obiektów.
- Należy unikać częstego korzystania z metody `document.write()`.

Powyższych „winowajców" należy się wystrzegać w szczególności w pętlach, gdzie opóźnienia mogą się pomnażać.

Analiza

Jedną z najbardziej nieefektywnych funkcji języka JavaScript jest `eval()`. Funkcja ta konwertuje obiekt zapisany w formie łańcucha znaków, zamieniając go na prawdziwe odwołanie do obiektu. Jest ona często wykorzystywanym rozwiązaniem, gdy dysponujemy łańcuchem znaków zawierającym nazwę obiektu lub jego identyfikator i na jego podstawie musimy stworzyć odwołanie do rzeczywistego obiektu. Na przykład dysponując sekwencją podmienianych obrazków tworzących menu, o nazwach `menuImg1`, `menuImg2` i tak dalej, można ulec pokusie stworzenia funkcji odtwarzającej oryginalną postać obrazków przy wykorzystaniu następującego fragmentu kodu:

```
for (var i = 0; i < 6; i++) {
    var obiImg = eval("document.menuImg" + i);
    obiImg.src = "images/menuImg" + i + "_normalny.jpg";
}
```

W tym przykładzie pokusa ta pojawia się, gdyż konkatenacja łańcuchów znaków jest używana także do tworzenia adresów URL obrazków. Niestety, użycie funkcji `eval()` wewnątrz pętli powoduje duże straty czasu.

Jeśli chodzi o odwoływanie się do obiektów elementów HTML, niemal zawsze istnieje sposób na uzyskanie zwyczajnego odwołania na podstawie łańcucha znaków, bez konieczności stosowania funkcji `eval()`. W przedstawionym przykładzie operującym na obrazkach rozwiązaniem jest kolekcja (tablica) `document.images`. Poniżej przedstawiona została zmodyfikowana, ulepszona wersja pętli:

```
for (var i = 0; i < 6; i++) {
    var obiImg = document.images["menuImg" + i];
    obiImg.src = "images/menuImg" + i + "_normalny.jpg";
}
```

Jeśli obiekt elementu ma nazwę lub identyfikator, to można uzyskać do niego dostęp za pośrednictwem jednej z kolekcji przechowujących obiekty elementów. W razie korzystania z przeglądarek obsługujących DOM W3C, w wypadkach gdy dysponujemy łańcuchem znaków zawierającym nazwę lub identyfikator elementu, naturalnym rozwiązaniem będzie wykorzystanie metody `document.getElementById()`. Jednak nawet starszy kod, dysponujący nazwami takich obiektów, jak obrazki lub pola formularzy, może korzystać z kolekcji, na przykład: document.images, bądź kolekcja elements obiektu form (`document.mojFormularz.elements["nazwaElementu"]`). W wypadku wykorzystania obiektów dodatkowe informacje można znaleźć w dyskusji poświęconej symulowanym tablicom mieszającym, zamieszczonej w dalszej części „Analizy".

Kolejnym czynnikiem pogarszającym efektywności skryptów jest instrukcja `with`. Jej celem jest pomoc w zawężeniu zakresu instrukcji umieszczonych wewnątrz bloku kodu. Na przykład jeśli pewna sekwencja instrukcji operuje głównie na właściwościach jednego obiektu lub wywołuje jego metody, to można ograniczyć zakres tego bloku kodu w taki sposób, iż instrukcje będą zakładać, że właściwości i metody należą do konkretnego obiektu. W poniższym fragmencie skryptu instrukcje wewnątrz bloku wywołują metodę `sort()` obiektu tablicy, a następnie odczytują wartość jego właściwości `length`:

```
with mojaTablica {
    sort();
    var iloscElem = length;
}
```

O tak, może się wydawać, że jest to rozwiązanie efektywne, jednak zanim interpreter przetworzy wyrażenia umieszczone wewnątrz bloku, będzie musiał wykonać wiele operacji niezbędnych do uzupełnienia odwołań do obiektu.

Przetworzenie każdego wyrażenia oraz odwołania wymaga czasu procesora. Im więcej jest „kropek" w odwołaniu, tym dłużej potrwa jego przetworzenie. Dlatego warto unikać powtarzania długich odwołań do obiektów lub wyrażeń, jeśli nie są one konieczne. Poniższy fragment kodu może przypominać funkcję znaną Czytelnikowi z własnych doświadczeń programistycznych:

```
function mojaFunkcja(idElementu) {
    for (i = 0; i < document.getElementById(idElementu).childNodes.length; i++) {
        if (document.getElementById(idElementu).childNodes[i].nodeType == 1) {
            // tutaj przetwarzamy węzły elementu
        }
    }
}
```

Podczas wykonywania tej funkcji ilość przetworzeń wyrażenia `document.getElement-ById()` odpowiada podwojonej ilości węzłów podrzędnych elementu, którego identyfikator został przekazany do funkcji. Na początku realizacji każdego przebiegu pętli `for` metoda jest wykonywana w ramach wyrażenia warunkowego, a później, wewnątrz pętli, jest ona wywoływana powtórnie podczas wyznaczania wartości warunku umieszczonego w instrukcji `if`. Jest więcej niż prawdopodobne, że dodatkowe instrukcje wewnątrz pętli także będą przetwarzać to wyrażenie, aby uzyskać dostęp do węzłów podrzędnych wskazanego elementu. Postępowanie takie jest marnowaniem czasu procesora.

Zamiast niego, kosztem utworzenia jednej dodatkowej zmiennej lokalnej, można wyeliminować wszystkie te powtarzające się wyrażenia. Wystarczy jeden raz określić wartość niezmiennej części wyrażenia, a później zamiast niej używać odwołania zapisanego w zmiennej:

```
function mojaFunkcja(idElementu) {
    var element = document.getElementById(idElementu);
    for (i = 0; i < element.childNodes.length; i++) {
        if (element.childNodes[i].nodeType == 1) {
            // tutaj przetwarzamy węzły elementu
        }
    }
}
```

Jeśli wszystkie operacje wykonywane wewnątrz pętli operują wyłącznie na węzłach podrzędnych elementu wyznaczonego poza pętlą, to przetwarzanie wyrażenia można by skrócić jeszcze bardziej:

```
function mojaFunkcja(idElementu) {
    var wezlyElementu = document.getElementById(idElementu).childNodes;
    for (i = 0; i < wezlyElementu.length; i++) {
        if (wezlyElementu[i].nodeType == 1) {
            // tutaj przetwarzamy węzły elementu
        }
    }
}
```

Dodatkową zaletą dokonanych modyfikacji jest skrócenie kodu źródłowego. Jeśli zatem uda się znaleźć w kodzie powtarzające się wyrażenia, których wartości nie ulegają zmianie w obrębie danego fragmentu skryptu, warto rozważyć wcześniejsze zapisanie ich wartości w zmiennych lokalnych.

Kolejnym usprawnieniem jest eliminacja czasochłonnego przeglądania tablic, w szczególności tablic wielowymiarowych lub tablic obiektów. Jeśli skrypt operuje na dużej tablicy (powiedzmy, zawierającej ponad 100 elementów), nawet średni czas wyszukiwania może być zauważalny. W takich wypadkach zamiast zwyczajnego przeglądania tablicy można wykorzystać rozwiązania przedstawione w recepturze 3.9, polegające na jednorazowym

wygenerowaniu symulowanej tablicy mieszającej. Tablice mieszającą można wygenerować podczas ładowania strony, dzięki czemu opóźnienia związane z jej tworzeniem dodadzą się do łącznego czasu wyświetlania strony. Od tej chwili odwoływanie się do elementów tablicy będzie niemal natychmiastowe, nawet jeśli poszukiwany element jest ostatni w kilkusetelementowej tablicy.

Ostatnia rada jest związana z wykorzystaniem metody `document.write()` do generacji zawartości podczas ładowania strony. Metodę tę należy traktować, jak gdyby z założenia była wolną operacją typu wejścia-wyjścia. Metodę `document.write()` powinno się wywoływać jak najrzadziej. Jeśli strona wymaga wygenerowania dużej ilości kodu HTML, to należy go w całości umieścić w jednej dużej zmiennej łańcuchowej, a następnie zapisać na stronie przy użyciu jednego wywołania `document.write()`.

Patrz także

Receptura 3.9 zawiera szczegółowe informacje o tworzeniu symulowanych tablic mieszających; receptura 3.13 przedstawia nieliczne sytuacje, w których wykorzystanie funkcji `eval()` jest konieczne.