

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

JavaScript dla webmasterów. Zaawansowane programowanie

Autor: Nicholas C. Zakas

Tłumaczenie: Jarosław Dobrzański (wstęp, rozdz. 1 – 8),

Krzysztof Czupryński (rozdz. 9), Daniel Kaczmarek

(rozdz. 10 – 20)

ISBN: 83-246-0280-1

Tytuł oryginału: [Professional JavaScript for Web Developers](#)

Format: B5, stron: 660



Kompendium wiedzy na temat języka JavaScript

- Model DOM i programowanie obiektowe
- Tworzenie dynamicznych interfejsów użytkownika
- Mechanizmy komunikacji klient-serwer

JavaScript to język programowania interpretowany po stronie przeglądarki i wykorzystywany do tworzenia elementów stron WWW. Opracowany w firmie Netscape, początkowo służył wyłącznie do weryfikowania poprawności danych wprowadzanych w formularzach. Dziś ma znacznie szersze zastosowania. Przede wszystkim pozwala wzbogacić stronę WWW o elementy niedostępne w „czystym” HTML, a jego najnowsze wersje umożliwiają korzystanie z dokumentów XML oraz komunikację z usługami sieciowymi. Z tego względu JavaScript jest niemal nieodłącznym elementem nowoczesnej witryny internetowej.

Książka „JavaScript dla webmasterów. Zaawansowane programowanie” to podręcznik opisujący wszystkie możliwości języka JavaScript. Przedstawia jego historię i pokazuje, jak rozwiązywać problemy, przed którymi często stają twórcy witryn i aplikacji WWW. W książce opisano kluczowe elementy języka, takie jak zdarzenia, wyrażenia regularne oraz metody identyfikacji przeglądarki WWW i interakcji z nią, umożliwiające tworzenie dynamicznych interfejsów użytkownika. Scharakteryzowano sposoby rozszerzania języka JavaScript oraz techniki budowania mechanizmów komunikacji między klientem i serwerem bez używania elementów pośredniczących.

- Podstawowe elementy ECMAScript
- Zasady programowania obiektowego
- Osadzanie elementów JavaScript w kodzie strony WWW
- Hierarchia modelu DOM
- Korzystanie z wyrażeń regularnych
- Detekcja typu przeglądarki i systemu operacyjnego
- Obsługa zdarzeń
- Kontrola poprawności danych z formularzy
- Wykorzystywanie elementów języka XML
- Komunikacja między przeglądarką i serwerem oraz usługi sieciowe
- Bezpieczeństwo aplikacji JavaScript

**Jeśli chcesz, aby Twoje aplikacje WWW działały szybciej,
skorzystaj z możliwości JavaScript**

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



Spis treści

O autorze	15
Wstęp	17
Rozdział 1. Czym jest JavaScript?	23
Krótka historia	24
Implementacje JavaScriptu	25
ECMAScript	25
Model DOM	28
Model BOM	31
Podsumowanie	32
Rozdział 2. Podstawy ECMAScriptu	33
Składnia	33
Zmienne	34
Słowa kluczowe	37
Słowa zarezerwowane	37
Wartości proste i referencje	37
Typy proste	38
Operator typeof	39
Typ Undefined	39
Typ Null	40
Typ Boolean	40
Typ Number	41
Typ String	43
Konwersje	44
Konwersja na ciąg znakowy	44
Konwersja na liczbę	45
Rzutowanie typów	46
Typy referencyjne	48
Klasa Object	48
Klasa Boolean	49
Klasa Number	50
Klasa String	51
Operator instanceof	55
Operatory	55
Operatory jednoargumentowe	55
Operatory bitowe	59
Operatory logiczne	65
Operatory mnożeniowe	69
Operatory addytywne	70

Operatory porównujące	72
Operatory równości	73
Operator warunkowy	75
Operatory przypisania	75
Przecinek	76
Instrukcje	76
Instrukcja if	76
Instrukcje iteracyjne	77
Etykietowanie instrukcji	79
Instrukcje break i continue	79
Instrukcja with	80
Instrukcja switch	81
Funkcje	82
Nie przeładowywać!	84
Obiekt arguments	84
Klasa Function	85
Zamknięcia	87
Podsumowanie	88

Rozdział 3. Podstawy programowania obiektowego 91

Terminologia obiektowa	91
Wymogi języków obiektowych	92
Składniki obiektu	92
Posługiwanie się obiektami	92
Deklaracja i tworzenie egzemplarzy	93
Referencje do obiektu	93
Usuwanie referencji do obiektu	93
Wiązanie wczesne a wiązanie późne	94
Typy obiektów	94
Obiekty własne	94
Obiekty wewnętrzne	105
Obiekty hosta	111
Zakres	112
Publiczny, prywatny i chroniony	112
Statyczny nie jest statyczny	112
Słowo kluczowe this	113
Definiowanie klas i obiektów	114
Wzorzec fabryki	114
Wzorzec konstruktora	116
Wzorzec prototypu	117
Hybrydowy wzorzec konstruktor-prototyp	118
Metoda dynamicznego prototypu	119
Hybrydowy wzorzec fabryki	120
Którego wzorca używać?	121
Praktyczny przykład	121
Modyfikowanie obiektów	123
Tworzenie nowej metody	124
Redefiniowanie istniejących metod	125
Bardzo późne wiązanie	126
Podsumowanie	126

Rozdział 4. Dziedziczenie	129
Dziedziczenie w praktyce	129
Implementacja dziedziczenia	130
Sposoby dziedziczenia	131
Bardziej praktyczny przykład	137
Alternatywne wzorce dziedziczenia	142
zInherit	142
xbObject	146
Podsumowanie	150
Rozdział 5. JavaScript w przeglądarce	151
JavaScript w kodzie HTML	151
Znacznik <script/>	151
Format plików zewnętrznych	152
Kod osadzony a pliki zewnętrzne	153
Umieszczenie znaczników	154
Ukrywać albo nie ukrywać	155
Znacznik <noscript/>	156
Zmiany w XHTML	157
JavaScript w SVG	159
Podstawy SVG	159
Znacznik <script/> w SVG	161
Umieszczenie znaczników <script/> w SVG	161
Obiektowy model przeglądarki	162
Obiekt window	162
Obiekt document	174
Obiekt location	178
Obiekt navigator	180
Obiekt screen	182
Podsumowanie	182
Rozdział 6. Podstawy modelu DOM	183
Co to jest DOM?	183
Wprowadzenie do XML	183
Interfejs API dla XML	187
Hierarchia węzłów	187
Modele DOM w konkretnych językach	190
Obsługa modelu DOM	191
Korzystanie z modelu DOM	191
Dostęp do węzłów dokumentu	191
Sprawdzanie typu węzła	193
Postępowanie z atrybutami	193
Dostęp do konkretnych węzłów	195
Tworzenie węzłów i manipulowanie nimi	197
Elementy funkcjonalne HTML w modelu DOM	202
Atrybuty jako właściwości	203
Metody do pracy z tabelami	203
Przemierzanie w modelu DOM	206
Obiekt NodeIterator	206
TreeWalker	211

Wykrywanie zgodności z modelem DOM	213
Poziom 3 modelu DOM	215
Podsumowanie	215
Rozdział 7. Wyrażenia regularne	217
Obsługa wyrażeń regularnych	217
Korzystanie z obiektu RegExp	218
Wyrażenia regularne w standardowych metodach typu String	219
Proste wzorce	221
Metaznaki	221
Używanie znaków specjalnych	221
Klasy znaków	222
Kwantyfikatory	225
Złożone wzorce	229
Grupowanie	229
Referencje wsteczne	230
Przemienność	231
Grupy nieprzechwyujące	233
Wyprzedzenia	234
Granice	235
Tryb wielowierszowy	236
Istota obiektu RegExp	237
Właściwości egzemplarza	237
Właściwości statyczne	238
Typowe wzorce	240
Kontrola poprawności dat	240
Kontrola poprawności danych karty kredytowej	242
Kontrola poprawności adresu e-mail	246
Podsumowanie	247
Rozdział 8. Wykrywanie przeglądarki i systemu operacyjnego	249
Obiekt navigator	249
Metody wykrywania przeglądarki	250
Wykrywanie obiektów/możliwości	250
Wykrywanie na podstawie ciągu User-Agent	251
(Niezbyt) krótka historia ciągu User-Agent	251
Netscape Navigator 3.0 i Internet Explorer 3.0	252
Netscape Communicator 4.0 i Internet Explorer 4.0	253
Internet Explorer 5.0 i nowsze wersje	254
Mozilla	254
Opera	256
Safari	257
Epilog	258
Skrypt wykrywający przeglądarkę	258
Metodyka	258
Pierwsze kroki	259
Wykrywanie przeglądarki Opera	261
Wykrywanie przeglądarek Konqueror i Safari	263
Wykrywanie przeglądarki Internet Explorer	266
Wykrywanie przeglądarki Mozilla	267
Wykrywanie przeglądarki Netscape Communicator 4.x	268

Skrypt wykrywający platformę i system operacyjny	269
Metodyka	269
Pierwsze kroki	269
Wykrywanie systemów operacyjnych Windows	270
Wykrywanie systemów operacyjnych dla platformy Macintosh	272
Wykrywanie uniksowych systemów operacyjnych	273
Pełny skrypt	274
Przykład — strona logowania	277
Podsumowanie	282
Rozdział 9. Wszystko o zdarzeniach	285
Zdarzenia dzisiaj	285
Przeływ zdarzenia	286
Rozprzestrzenianie się zdarzeń	286
Przechwytywanie zdarzeń	288
Przeływ zdarzenia w modelu DOM	289
Procedury obsługi zdarzeń i słuchacze zdarzeń	290
Internet Explorer	291
DOM	292
Obiekt Event	294
Lokalizacja	294
Właściwości i metody	295
Podobieństwa	298
Różnice	301
Typy zdarzeń	304
Zdarzenia myszki	304
Zdarzenia klawiatury	309
Zdarzenia HTML	311
Zdarzenia mutacji	317
Zdarzenia wspólne dla wielu przeglądarek	317
Obiekt EventUtil	317
Dodawanie/usuwanie procedur obsługi błędów	318
Formatowanie obiektu event	320
Pobieranie obiektu zdarzenia	324
Przykład	325
Podsumowanie	326
Rozdział 10. Zaawansowane techniki DOM	329
Skrypty definiujące style	329
Metody modelu DOM przetwarzające style	331
Własne podpowiedzi	333
Sekcje rozwijalne	334
Dostęp do arkuszy stylów	335
Style obliczane	339
innerText i innerHTML	341
outerText i outerHTML	342
Zakresy	344
Zakresy w modelu DOM	344
Zakresy w Internet Explorerze	355
Czy zakresy są praktyczne?	359
Podsumowanie	360

Rozdział 11. Formularze i integralność danych	361
Podstawowe informacje na temat formularzy	361
Oprogramowywanie elementu <form/>	363
Odczytywanie odwołań do formularza	363
Dostęp do pól formularza	364
Wspólne cechy pól formularzy	364
Ustawienie aktywności na pierwszym polu	365
Zatwierdzanie formularzy	366
Jednokrotne zatwierdzanie formularza	368
Resetowanie formularzy	368
Pola tekstowe	369
Odczytywanie i zmiana wartości pola tekstowego	369
Zaznaczanie tekstu	371
Zdarzenia pól tekstowych	372
Automatyczne zaznaczanie tekstu	372
Automatyczne przechodzenie do następnego pola	373
Ograniczanie liczby znaków w polu wielowierszowym	374
Umożliwianie i blokowanie wpisywania znaków w polach tekstowych	376
Liczbowe pola tekstowe reagujące na klawisze strzałek w górę i w dół	382
Pola list i listy rozwijane	384
Uzyskiwanie dostępu do opcji	385
Odczytywanie i zmiana wybranych opcji	385
Dodawanie opcji	386
Usuwanie opcji	388
Przenoszenie opcji	388
Zmiana kolejności opcji	389
Tworzenie pola tekstowego z automatyczną podpowiedzią	390
Dopasowywanie	390
Główny mechanizm	391
Podsumowanie	393
Rozdział 12. Sortowanie tabel	395
Punkt wyjścia: tablice	395
Metoda reverse()	397
Sortowanie tabeli zawierającej jedną kolumnę	397
Funkcja porównania	399
Funkcja sortTable()	399
Sortowanie tabel zawierających więcej niż jedną kolumnę	401
Generator funkcji porównania	402
Zmodyfikowana funkcja sortTable()	403
Sortowanie w porządku malejącym	404
Sortowanie danych innych typów	406
Sortowanie zaawansowane	410
Podsumowanie	414
Rozdział 13. Technika „przeciągnij i upuść”	415
Systemowa technika „przeciągnij i upuść”	415
Zdarzenia techniki „przeciągnij i upuść”	416
Obiekt dataTransfer	422
Metoda dragDrop()	426
Zalety i wady	428

Symulacja techniki „przeciągnij i upuść”	429
Kod	430
Tworzenie docelowych obiektów przeciągania	432
Zalety i wady	434
zDragDrop	435
Tworzenie elementu, który można przeciągać	435
Tworzenie docelowego obiektu przeciągania	436
Zdarzenia	436
Przykład	437
Podsumowanie	439
Rozdział 14. Obsługa błędów	441
Znaczenie obsługi błędów	441
Błędy i wyjątki	442
Raportowanie błędów	443
Internet Explorer (Windows)	443
Internet Explorer (Mac OS)	445
Mozilla (wszystkie platformy)	446
Safari (Mac OS X)	446
Opera 7 (wszystkie platformy)	448
Obsługa błędów	449
Procedura obsługi zdarzenia onerror	449
Instrukcja try...catch	452
Techniki debugowania	458
Używanie komunikatów	458
Używanie konsoli Javy	459
Wysyłanie komunikatów do konsoli JavaScriptu (tylko Opera 7+)	460
Rzucanie własnych wyjątków	460
Narzędzie The JavaScript Verifier	462
Debugery	462
Microsoft Script Debugger	462
Venkman — debugger dla Mozilli	465
Podsumowanie	474
Rozdział 15. JavaScript i XML	475
Obsługa XML DOM w przeglądarkach	475
Obsługa XML DOM w Internet Explorerze	475
Obsługa XML DOM w Mozilli	480
Ujednoczenie implementacji	485
Obsługa XPath w przeglądarkach	496
Wprowadzenie do XPath	496
Obsługa XPath w Internet Explorerze	497
Obsługa XPath w Mozilli	498
Obsługa XSLT w przeglądarkach	503
Obsługa XSLT w Internet Explorerze	505
Obsługa XSLT w Mozilli	509
Podsumowanie	511
Rozdział 16. Komunikacja między klientem a serwerem	513
Pliki cookies	513
Składniki plików cookies	514
Inne ograniczenia bezpieczeństwa	515

Cookies w języku JavaScript	515
Cookies na serwerze	517
Przekazywanie cookies między klientem a serwerem	521
Ukryte ramki	523
Używanie ramek iframe	524
Żądania HTTP	526
Używanie nagłówków	528
Bliźniacze implementacje obiektu XML HTTP	529
Wykonywanie żądania GET	530
Wykonywanie żądania POST	531
Żądania LiveConnect	532
Wykonywanie żądania GET	532
Wykonywanie żądania POST	534
Inteligentne żądania HTTP	536
Metoda get()	536
Metoda post()	539
Zastosowania praktyczne	540
Podsumowanie	540
Rozdział 17. Usługi sieciowe	543
Podstawowe informacje na temat usług sieciowych	543
Czym jest usługa sieciowa?	543
WSDL	544
Usługi sieciowe w Internet Explorerze	547
Używanie komponentu WebService	547
Przykład użycia komponentu WebService	549
Usługi sieciowe w Mozilli	551
Rozszerzone uprawnienia	551
Używanie metod SOAP	552
Używanie obiektów proxy WSDL	556
Rozwiązanie dla różnych przeglądarek	560
Obiekt WebService	560
Usługa Temperature Service	562
Używanie obiektu TemperatureService	564
Podsumowanie	565
Rozdział 18. Praca z modułami rozszerzającymi	567
Do czego służą moduły rozszerzające?	567
Popularne moduły rozszerzające	568
Typy MIME	569
Osadzanie modułów rozszerzających	570
Dołączanie parametrów	571
Netscape 4.x	571
Wykrywanie modułów rozszerzających	572
Wykrywanie modułów rozszerzających w stylu Netscape	572
Wykrywanie modułów rozszerzających ActiveX	577
Wykrywanie modułów rozszerzających w różnych przeglądarkach	579
Aplety Java	580
Osadzanie apletów	580
Odwołania do apletów w kodzie JavaScript	581
Tworzenie apletów	582

Komunikacja skryptu JavaScript z językiem Java	583
Komunikacja języka Java ze skryptem JavaScript	586
Filmy Flash	589
Osadzanie filmów Flash	590
Odwołania do filmów Flash	590
Komunikacja języka JavaScript z filmami Flash	591
Komunikacja Flasha z językiem JavaScript	594
Kontrolki ActiveX	596
Podsumowanie	599
Rozdział 19. Zagadnienia związane z wdrażaniem aplikacji JavaScriptu	601
Bezpieczeństwo	601
Polityka jednakowego pochodzenia	602
Zagadnienia związane z obiektem okna	603
Zagadnienia dotyczące przeglądarki Mozilla	604
Ograniczenia zasobów	607
Zagadnienia dotyczące lokalizacji	608
Sprawdzanie języka w kodzie JavaScript	608
Strategie umiędzynaradawiania	609
Zagadnienia dotyczące ciągów znaków	610
Optymalizacja kodu JavaScript	613
Czas pobierania	613
Czas wykonania	619
Zagadnienia dotyczące własności intelektualnej	635
Obfuskacja	635
Microsoft Script Encoder (wyłącznie Internet Explorer)	636
Podsumowanie	637
Rozdział 20. Rozwój języka JavaScript	639
ECMAScript 4	639
Propozycja firmy Netscape	640
Implementacje	646
ECMAScript dla języka XML	648
Podejście	648
Pętla for each...in	650
Nowe klasy	650
Implementacje	660
Podsumowanie	660
Skorowidz	661

4

Dziedziczenie

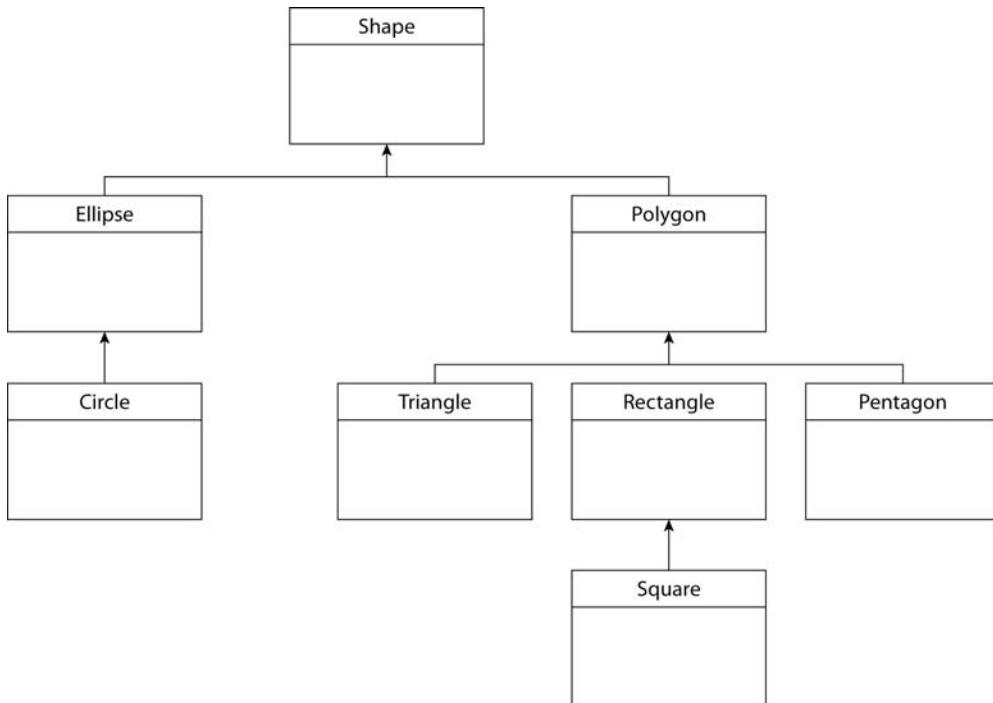
Prawdziwie obiektowy język programowania musi obsługiwać dziedziczenie, czyli możliwość korzystania (**dziedziczenia**) z metod i właściwości jednej klasy przez inną klasę. W poprzednim rozdziale nauczyłeś się definiować właściwości i metody klasy. Czasem chcemy, by dwie różne klasy mogły korzystać z tych samych metod. Wtedy właśnie przydaje się dziedziczenie.

Dziedziczenie w praktyce

Najprostszym sposobem na opisanie dziedziczenia jest posłużenie się klasycznym przykładem — figurami geometrycznymi. Tak naprawdę istnieją dwa typy figur płaskich: elipsy (które są okrągłe) i wielokąty (które mają pewną ilość boków). Koło to rodzaj elipsy z jednym ogniskiem; trójkąty, czworokąty i pięciokąty to rodzaje wielokątów z różną ilością boków. Kwadrat to rodzaj czworokąta z wszystkimi bokami równymi. Jest to idealny przykład powiązania dziedzicznego.

W przykładzie tym „figura” (Shape) jest **klasą bazową** (wszystkie klasy są jej potomkami) dla klas „elipsa” (Ellipse) i „wielokąt” (Polygon). Elipsa ma jedną właściwość zwaną „ogniska” (foci) wskazującą ilość ognisk elipsy. Koło (Circle) jest potomkiem elipsy, więc nazywamy je **podklasą** elipsy, a sama elipsa jest **nadklasą** dla koła. Podobnie trójkąt (Triangle), czworokąt (Rectangle) i pięciokąt (Pentagon) są podklasami wielokąta, a wielokąt jest nadklasą dla każdej z tych klas. Wreszcie kwadrat (Square) jest potomkiem czworokąta.

Powiązania dziedziczne najlepiej opisać przy użyciu diagramu — w tym momencie pomocny okazuje się **uniwersalny język modelowania**, czyli UML. Jednym z wielu przeznaczeń UML jest wizualna reprezentacja złożonych powiązań między obiektami, takich jak dziedziczenie. Na rysunku 4.1 widać diagram UML opisujący związek klasy Shape z jej podklasami.

**Rysunek 4.1.**

W UML każdy prostokąt reprezentuje klasę opisaną nazwą. Linie biegnące od wierzchu trójkąta, czworokąta i pięciokąta zbiegają się i wskazują na figurę, pokazując, że każda z tych klas jest potomkiem figury. Podobnie strzałka biegnąca od kwadratu do czworokąta symbolizuje powiązanie dziedziczne pomiędzy tymi klasami.

Więcej na temat UML można przeczytać w książce Instant UML (Wrox Press, ISBN 1861000871).

Implementacja dziedziczenia

Aby zaimplementować dziedziczenie w języku ECMAScript, zaczynamy od klasy bazowej dla wszystkich potomków. Kandydatami na klasy bazowe są klasy stworzone przez programistę. Ze względów bezpieczeństwa obiekty własne i obiekty hosta nie mogą być klasami bazowymi. Zapobiega to publicznemu udostępnieniu skompilowanego kodu poziomu przeglądarki, który mógłby potencjalnie zostać użyty w złych zamiarach.

Po wybraniu klasy bazowej można przejść do tworzenia podklas. To, czy klasa bazowa będzie w ogóle używana, zależy wyłącznie od nas. Czasami pojawia się potrzeba stworzenia klasy bazowej, która nie ma być wykorzystywana bezpośrednio. Zamiast tego udostępnia ona jedynie wspólne podklasom cechy funkcjonalne. W takich okolicznościach klasę bazową uważa się za **abstrakcyjną**.

ECMAScript nie pozwala na dosłowne definiowanie klas abstrakcyjnych, tak jak niektóre inne języki, ale czasami tworzone są klasy bazowe, które nie są przeznaczone do użytku. Zwykle jedynie w dokumentacji opisane są jako abstrakcyjne.

Stworzone podklasy dziedziczą wszystkie właściwości i metody nadklasy, w tym implementacje konstruktora i metod. Pamiętaj, że wszystkie właściwości i metody są publiczne, a zatem podklasy mogą się do nich odwoływać bezpośrednio. Podklasy mogą dodawać nowe właściwości i metody niewystępujące w nadklasach lub zastępować właściwości i metody nadklasy własnymi implementacjami.

Sposoby dziedziczenia

Jak zwykle w języku ECMAScript można implementować dziedziczenie na kilka sposobów. Wynika to z faktu, że dziedziczenie w JavaScriptcie nie jest jawne, tylko emulowane. Oznacza to, że interpreter nie obsługuje wszystkich szczegółów związanych z dziedziczeniem. Zadaniem dla programisty jest obsługa dziedziczenia w sposób najbardziej adekwatny do okoliczności.

Maskowanie obiektów

O maskowaniu obiektów nie myślano jeszcze, kiedy opracowywano pierwszą wersję ECMAScriptu. Koncepcja ta ewoluowała, w miarę jak programiści coraz lepiej rozumieli, jak tak naprawdę działają funkcje i, w szczególności, jak posługiwać się słowem `this` w kontekście funkcji.

Rozumowanie jest następujące: konstruktor przypisuje wszystkie właściwości i metody (przy deklarowaniu klas wzorcem konstruktora) słowem `this`. Ponieważ konstruktor to po prostu funkcja, można uczynić konstruktor klasy `ClassA` metodą klasy `ClassB` i ją wywołać. `ClassB` zostanie wówczas wyposażona we wszystkie właściwości i metody zdefiniowane w konstruktorze klasy `ClassA`. Na przykład, klasy `ClassA` i `ClassB` można zdefiniować następująco:

```
function ClassA(sColor) {
    this.sColor = sColor;
    this.sayColor = function () {
        alert(this.sColor);
    };
}

function ClassB(sColor) {
}
```

Jak zapewne pamiętasz, słowo kluczowe `this` wskazuje na bieżąco tworzony w konstruktorze obiekt. Jednak w metodzie `this` wskazuje obiekt, do którego metoda należy. Zgodnie z omawianą teorią stworzenie klasy `ClassA` jako normalnej funkcji, a nie jako konstruktora, tworzy pewien rodzaj dziedziczenia. Można to zrobić w konstruktorze klasy `ClassB` tak:

```
function ClassB(sColor) {
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;
}
```

W kodzie tym metoda `newMethod` jest przypisywana do `ClassA` (pamiętaj, że nazwa funkcji jest tylko wskaźnikiem na nią). Następnie metoda ta jest wywoływana poprzez przekazanie argumentu `sColor` z konstruktora klasy `ClassB`. Ostatni wiersz kodu usuwa referencję do klasy `ClassA`, aby nie można jej było później wywołać.

Wszystkie nowe właściwości i metody muszą być dodane po wierszu, który usuwa nową metodę. W innym przypadku narażamy się na ryzyko nadpisania nowych właściwości i metod tymi pochodzącymi z nadklasy:

```
function ClassB(sColor, sName) {
  this.newMethod = ClassA;
  this.newMethod(sColor);
  delete this.newMethod;
```

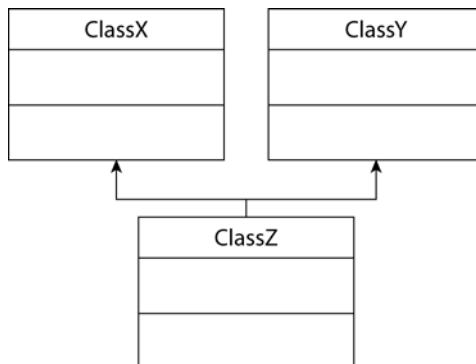
```
  this.sName = sName;
  this.sayName = function () {
    alert(this.sName);
  };
}
```

Aby dowieść, że to działa, możemy uruchomić następujący przykład:

```
var oObjA = new ClassA("czerwony");
var oObjB = new ClassB("niebieski", "Mikołaj");
oObjA.sayColor();
oObjB.sayColor();
oObjB.sayName();
```

Co ciekawe, maskowanie obiektów pozwala na stosowanie **dziedziczenia wielokrotnego**, co oznacza, że klasa może być potomkiem kilku nadklas. Dziedziczenie wielokrotne jest reprezentowane w UML tak, jak widać na rysunku 4.2.

Rysunek 4.2.



Jeżeli na przykład istnieją dwie klasy `ClassX` i `ClassY`, a chcemy, by klasa `ClassZ` była potomkiem obu tych klas, to możemy napisać:

```
function ClassZ() {
  this.newMethod = ClassX;
  this.newMethod();
  delete this.newMethod;

  this.newMethod = ClassY;
```

```

    this.newMethod();
    delete this.newMethod;
}

```

Minusem jest to, że jeśli klasy `ClassX` i `ClassY` mają właściwości lub metody o tej samej nazwie, to `ClassY` ma pierwszeństwo, ponieważ dziedziczenie cech po niej następuje później. Poza tym drobnym problemem dziedziczenie wielokrotne poprzez maskowanie obiektów jest proste.

Ponieważ ta metoda dziedziczenia zrodziła się „w praniu”, trzecia edycja ECMAScriptu wprowadza dwie nowe metody obiektu `Function`: `call()` i `apply()`.

Metoda call()

Metoda `call()` jest najbardziej podobna do klasycznej metody maskowania obiektów. Jej pierwszy argument to obiekt, który ma wskazywać `this`. Wszystkie pozostałe argumenty są przekazywane bezpośrednio do samej funkcji. Na przykład:

```

function sayColor(sPrefix, sSuffix) {
    alert(sPrefix + this.sColor + sSuffix);
};

```

```

var oObj = new Object();
oObj.sColor = "czerwony";

```

```

// wyświetla "Kolorem jest czerwony. Naprawdę ładny kolor. "

```

```

sayColor.call(oObj, "Kolorem jest ", ". Naprawdę ładny kolor. ");

```

W tym przykładzie funkcja `sayColor()` została zdefiniowana poza obiektem i wskazuje na słowo `this`, mimo że nie została związana z żadnym obiektem. Obiektowi `oObj` nadano właściwość `sColor` o wartości "czerwony". W wywołaniu `call()` pierwszym argumentem jest `oObj`, co wskazuje, że słowo `this` w obrębie `sayColor()` powinno wskazywać wartość `oObj`. Drugi i trzeci argument to ciągi znakowe. Odpowiadają one argumentom `sPrefix` i `sSuffix` funkcji `sayColor()`, w efekcie wyświetlony zostaje tekst "Kolorem jest czerwony. Naprawdę ładny kolor. ".

Aby wykorzystać to w schemacie dziedziczenia poprzez maskowanie obiektów, wystarczy zastąpić trzy wiersze, które przypisują, wywołują i usuwają nową metodę:

```

function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;

```

```

    ClassA.call(this, sColor);

```

```

    this.sName = sName;
    this.sayName = function () {
        alert(this.sName);
    };
}

```

W tym przypadku chcemy, by słowo kluczowe `this` w `ClassA` odpowiadało nowo utworzonemu obiektowi `ClassB`, przesyłamy je więc jako pierwszy argument. Drugi argument to `sColor`, tylko jeden dla każdej z klas.

Metoda apply()

Metoda `apply()` pobiera dwa argumenty: obiekt, który ma wskazywać `this` i tablicę argumentów do przesłania do funkcji. Na przykład:

```
function sayColor(sPrefix, sSuffix) {
    alert(sPrefix + this.sColor + sSuffix);
};
```

```
var oObj = new Object();
oObj.sColor = "czerwony";
```

// wyświetla "Kolorem jest czerwony. Naprawdę ładny kolor. "

```
sayColor.apply(oObj, new Array("Kolorem jest ", ". Naprawdę ładny kolor. "));
```

Przykład jest taki sam jak poprzednio, ale tym razem wywoływana jest metoda `apply()`. W wywołaniu pierwszym argumentem pozostaje `oObj`, który dalej wskazuje, że słowo `this` w funkcji `sayColor()` ma mieć przypisaną wartość `oObj`. Drugi argument to tablica składająca się z dwóch ciągów, które odpowiadają argumentom `sPrefix` i `sSuffix` funkcji `sayColor()`. Efektem jest ponownie wyświetlenie tekstu "Kolorem jest czerwony. Naprawdę ładny kolor. ".

Tę metodę również można użyć w miejsce trzech wierszy przypisujących, wywołujących i usuwających nową metodę:

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
```

```
ClassA.apply(this, new Array(sColor));
```

```
    this.sName = sName;
    this.sayName = function () {
        alert(this.sName);
    };
}
```

Ponownie przesyłamy `this` jako pierwszy argument. Drugi argument to tablica z tylko jedną wartością `sColor`. Zamiast tego, możemy jako drugi argument metody `apply()` przesłać cały obiekt argumentów klasy `ClassB`:

```
function ClassB(sColor, sName) {
    //this.newMethod = ClassA;
    //this.newMethod(sColor);
    //delete this.newMethod;
```

```
ClassA.apply(this, arguments);
```

```
    this.sName = sName;
    this.sayName = function () {
        alert(this.sName);
    };
}
```

Oczywiście, przesyłanie obiektu reprezentującego argumenty działa tylko wówczas, kiedy kolejność argumentów w konstruktorze nadklasy jest dokładnie taka sama jak kolejność argumentów w podklasie. Jeżeli tak nie jest, trzeba stworzyć odrębną tablicę, aby umieścić w niej argumenty w dobrej kolejności. Można też posłużyć się wówczas metodą `call()`.

Wiązanie łańcuchowe prototypów

Formą dziedziczenia, jaka w zamyśle miała być używana w języku ECMAScript, było wiązanie łańcuchowe prototypów. W poprzednim rozdziale przedstawiłem wzorzec prototypu służący do definiowania klas. Wiązanie łańcuchowe prototypów jest rozszerzeniem tego wzorca o ciekawy przepis na realizację dziedziczenia.

W poprzednim rozdziale dowiedziałeś się, że obiekt `prototype` jest szablonem, na którym opiera się obiekt w chwili tworzenia egzemplarza. Przypomnijmy w skrócie: wszelkie właściwości i metody obiektu `prototype` będą przesyłane do wszystkich egzemplarzy tej klasy. Wiązanie łańcuchowe prototypów wykorzystuje tę możliwość, by urzeczywistnić dziedziczenie.

Gdyby klasy z poprzedniego przykładu zdefiniować na nowo, posługując się wzorcem prototypu, wyglądałyby następująco:

```
function ClassA() {
}

ClassA.prototype.sColor = "czerwony";
ClassA.prototype.sayColor = function () {
    alert(this.sColor);
};

function ClassB() {
}
```

```
ClassB.prototype = new ClassA();
```

Cała magia wiązania łańcuchowego prototypów ujawnia się w wyróżnionym, powyższym wierszu. Sprawiamy tu, że właściwość `prototype` klasy `ClassB` staje się egzemplarzem klasy `ClassA`. Ma to sens, ponieważ zależy nam na wszystkich właściwościach i metodach `ClassB`, ale nie chcemy przypisywać każdej z osobna do właściwości `prototype` klasy `ClassB`. Czyż istnieje lepszy sposób niż przekształcenie `prototype` w egzemplarz klasy `ClassA`?

Jak widać, w wywołaniu konstruktora `ClassA` nie przesłano żadnego parametru. Jest to norma w przypadku wiązania łańcuchowego prototypów. Musimy więc zadbać o to, by konstruktor działał poprawnie bez żadnych argumentów.

Podobnie jak przy maskowaniu, wszelkie nowe właściwości i metody podklasy muszą być definiowane po przypisaniu właściwości `property`, ponieważ wszystkie metody przypisane wcześniej zostaną usunięte. Dlaczego? Jako że właściwość `property` jest w całości zastępowana nowym obiektem, pierwotny obiekt, do którego dodawaliśmy metody, jest niszczone. Tak więc kod dodający właściwość `sName` i metodę `sayName()` do klasy `ClassB` powinien wyglądać tak:

```
function ClassB() {
}

ClassB.prototype = new ClassA();

ClassB.prototype.sName = "";
ClassB.prototype.sayName = function () {
    alert(this.sName);
};
```

Kod ten można przetestować, uruchamiając następujący przykład:

```
var oObjA = new ClassA();
var oObjB = new ClassB();
oObjA.sColor = "czerwony";
oObjB.sColor = "niebieski";
oObjB.sName = "Mikołaj";
oObjA.sayColor();
oObjB.sayColor();
oObjB.sayName();
```

Dodatkowo przy wiązaniu łańcuchowym prototypów operator `instanceof` działa w dość unikalny sposób. Dla wszystkich egzemplarzy `ClassB` operator `instanceof` zwraca `true` zarówno dla `ClassA`, jak i dla `ClassB`. Na przykład:

```
var oObjB = new ClassB();
alert(oObjB instanceof ClassA); // wyświetla "true"
alert(oObjB instanceof ClassB); // wyświetla "true"
```

W świecie luźnej kontroli typów, jaka obowiązuje w języku ECMAScript, jest to niezwykle przydatne narzędzie, które nie jest dostępne, gdy posługujemy się maskowaniem obiektów.

Minusem wiązania łańcuchowego prototypów jest brak obsługi dziedziczenia wielokrotnego. Jak zapewne pamiętasz, wiązanie łańcuchowe polega na nadpisaniu właściwości prototypu klasy innym typem obiektu.

Metoda hybrydowa

Dziedziczenie poprzez maskowanie obiektów posługuje się przy definiowaniu klas wzorcem konstruktora, nie korzystając w ogóle z prototypów. Główny problem polega tu na tym, że musimy użyć wzorca konstruktora, który (o czym przekonałeś się w poprzednim rozdziale) nie jest optymalny. Jeżeli z kolei zastosujemy wiązanie łańcuchowe prototypów, tracimy możliwość posługiwania się konstruktorami z argumentami. Jak radzą sobie z tym programiści? Odpowiedź jest prosta: stosują obydwie metody.

W poprzednim rozdziale dowiedziałeś się, że najlepszy sposób na tworzenie klas polega na stosowaniu wzorca konstruktora do definiowania właściwości i wzorca prototypu do definiowania metod. To samo dotyczy dziedziczenia — używamy maskowania do dziedziczenia właściwości od konstruktora i wiązania łańcuchowego prototypów, by dziedziczyć metody po obiekcie prototypu. Spójrzmy na poprzedni przykład napisany od nowa przy użyciu obydwu metod dziedziczenia:

```
function ClassA(sColor) {
    this.sColor = sColor;
}

ClassA.prototype.sayColor = function() {
    alert(this.sColor);
};

function ClassB(sColor, sName) {
    ClassA.call(this, sColor);
    this.sName = sName;
```

```

}

ClassB.prototype = new ClassA();

ClassB.prototype.sayName = function () {
    alert(this.sName);
};

```

W przykładzie tym dziedziczenie następuje w dwóch wyróżnionych wierszach. Najpierw w konstruktorze klasy `ClassB` używane jest maskowanie obiektów, aby odziedziczyć właściwość `sColor` po klasie `ClassA`. W drugim wyróżnionym wierszu zastosowane zostało wiązanie łańcuchowe prototypów, by odziedziczyć metody klasy `ClassA`. Ponieważ metoda hybrydowa korzysta z wiązania łańcuchowego prototypów, operator `instanceof` wciąż będzie działał poprawnie.

Kod ten testuje następujący przykład:

```

var oObjA = new ClassA("czerwony");
var oObjB = new ClassB("niebieski", "Mikołaj");
oObjA.sayColor(); // wyświetla "czerwony"
oObjB.sayColor(); // wyświetla "niebieski"
oObjB.sayName(); // wyświetla "Mikołaj"

```

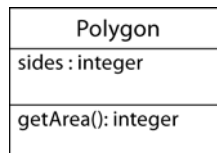
Bardziej praktyczny przykład

Tworząc prawdziwe aplikacje lub witryny internetowe, raczej nie będziemy tworzyć klas o nazwach w rodzaju `ClassA` i `ClassB`. Bardziej prawdopodobne, że będziemy tworzyć klasy reprezentujące konkretne rzeczy, np. figury geometryczne. Jeżeli przypomnisz sobie przykład z figurami z początku tego rozdziału, uświadomisz sobie, że klasy reprezentujące wielokąt (`Polygon`), trójkąt (`Triangle`) i czworokąt (`Rectangle`) tworzą ciekawy zbiór danych do analizy.

Tworzenie klasy bazowej

Zastanówmy się najpierw nad klasą `Polygon` reprezentującą wielokąt. Jakie właściwości i metody będą w niej konieczne? Po pierwsze, ważna jest informacja o liczbie boków, z jakich składa się wielokąt, należałoby więc wprowadzić właściwość `iSides`, która będzie liczbą całkowitą. Co jeszcze może być potrzebne wielokątowi? Możemy chcieć wyliczyć pole wielokąta, dodajmy więc metodę `getArea()`, która będzie je obliczać. Na rysunku 4.3 widać reprezentację UML tej klasy:

Rysunek 4.3.



W UML właściwości reprezentowane są nazwą właściwości i typem, które pojawiają się w polu tuż pod nazwą klasy. Metody znajdują się pod właściwościami — w tym przypadku widoczna jest nazwa właściwości i typ wartości, jaką zwraca.

W języku ECMAScript klasę tą można zapisać następująco:

```
function Polygon(iSides) {
    this.iSides = iSides;
}

Polygon.prototype.getArea = function () {
    return 0;
};
```

Zauważ, że klasa Polygon sama w sobie nie jest na tyle konkretna, by można jej było używać. Metoda `getArea()` zwraca 0, ponieważ pełni jedynie rolę miejsca na metody podklas, które ją zastąpią.

Tworzenie podklas

Zastanówmy się teraz nad klasą Triangle reprezentującą trójkąt. Trójkąt ma trzy boki, więc klasa ta musi zastąpić właściwość `iSides` klasy Polygon i ustalić jej wartość na 3. Metoda `getArea()` również musi być zastąpiona, by skorzystać ze wzoru na pole trójkąta, którym jest $1/2 \times \text{podstawa} \times \text{wysokość}$. Skąd jednak metoda weźmie wartości podstawy i wysokości? Ponieważ muszą zostać wprowadzone konkretne ich wartości, konieczne jest stworzenie właściwości `iBase` (podstawa) i `iHeight` (wysokość). Reprezentacja UML-u trójkąta widoczna jest na rysunku 4.4.

Rysunek 4.4.

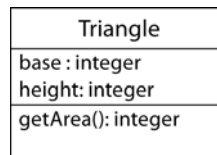
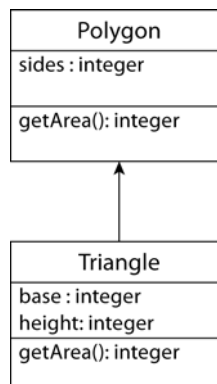


Diagram ten ukazuje jedynie nowe właściwości i zastępowane przez klasę Triangle metody. Gdyby klasa Triangle nie definiowała własnej wersji `getArea()`, metoda ta nie zostałaby wymieniona na diagramie. Byłaby traktowana jako odziedziczona po klasie Polygon. Nieco lepiej wyjaśnia to kompletny diagram UML reprezentujący powiązania między klasami Polygon i Triangle (rysunek 4.5).

Rysunek 4.5.



W diagramach UML nigdy nie powtarza się metod odziedziczonych, chyba że zostały zastąpione (lub przeładowane, co akurat w języku ECMAScript nie jest możliwe).

Kod klasy Triangle wygląda następująco:

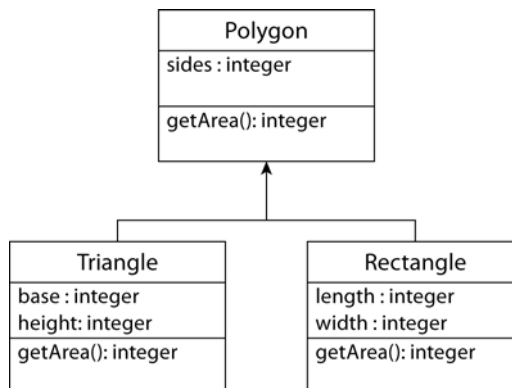
```
function Triangle(iBase, iHeight) {
  Polygon.call(this, 3);
  this.iBase = iBase;
  this.iHeight = iHeight;
}
Triangle.prototype = new Polygon();
Triangle.prototype.getArea = function () {
  return 0.5 * this.iBase * this.iHeight;
};
```

Zwróćmy uwagę, że konstruktor klasy Triangle przyjmuje dwa argumenty, iBase i iHeight, mimo że konstruktor klasy Polygon przyjmuje tylko jeden argument iSides. Wynika to z faktu, że z góry wiadomo ile boków ma trójkąt i nie chcemy, aby programista mógł to zmieniać. Więc kiedy używamy maskowania obiektów liczba 3 jest przesyłana do konstruktora Polygon jako liczba boków dla tego obiektu. Następnie wartości iBase i iHeight są przypisywane do odpowiednich właściwości.

Po zastosowaniu wiązania łańcuchowego prototypów do dziedziczenia metod klasa Triangle zastępuje metodę getArea() własną, by udostępnić wzór na pole trójkąta.

Ostatnia klasa to Rectangle, reprezentująca czworokąt, która również jest potomkiem klasy Polygon. Czworokąty mają cztery boki, a ich pole oblicza się, mnożąc długość przez szerokość, które są dwoma właściwościami, jakie musi wprowadzić klasa Rectangle. Na diagramie UML klasa ta pojawia się obok klasy Triangle, ponieważ dla obu tych klas nadklasa jest Polygon (patrz: rysunek 4.6).

Rysunek 4.6.



Kod ECMAScript klasy Rectangle wygląda następująco:

```
function Rectangle(iLength, iWidth) {
  Polygon.call(this, 4);
  this.iLength = iLength;
```

```

    this.iWidth = iWidth;
  }

  Rectangle.prototype = new Polygon();
  Rectangle.prototype.getArea = function () {
    return this.iLength * this.iWidth;
  };

```

Zwróć uwagę, że konstruktor klasy `Rectangle` również nie przyjmuje `iSides` jako argumentu i ponownie wartość stała (4) zostaje przesłana wprost do konstruktora klasy `Polygon`. Również na podobieństwo `Triangle`, klasa `Rectangle` wprowadza dwie nowe właściwości jako argumenty dla konstruktora i zastępuje metodę `getArea()` własną wersją.

Testowanie kodu

Stworzony dla tego przykładu kod klas można przetestować, uruchamiając następujący przykład:

```

var oTriangle = new Triangle(12, 4);
var oRectangle = new Rectangle(22, 10);

alert(oTriangle.iSides);      // wyświetla "3"
alert(oTriangle.getArea());  // wyświetla "24"

alert(oRectangle.iSides);    // wyświetla "4"
alert(oRectangle.getArea()); // wyświetla "220"

```

Kod ten tworzy trójkąt o podstawie 12 i wysokości 4 oraz prostokąt o długości 22 i szerokości 10. Następnie wyświetlane są liczby boków i pola dla każdej figury, aby dowieść, że właściwość `iSides` została poprawnie ustawiona i że metoda `getArea()` zwraca stosowne wartości. Pole trójkąta powinno wynosić 24, a pole prostokąta 220.

Co z dynamicznymi prototypami?

W poprzednim przykładzie zastosowano wzorzec hybrydowy konstruktor-prototyp do definiowania obiektów, aby ukazać dziedziczenie, ale czy będzie to działać również z dynamicznymi prototypami? Otóż nie.

Dziedziczenie nie działa z dynamicznymi prototypami z uwagi na unikatową naturę obiektu `prototype`. Spójrzmy na następujący kod (który jest nieprawidłowy, ale mimo to warto go przestudiować):

```

function Polygon(iSides) {
  this.iSides = iSides;

  if (typeof Polygon._initialized == "undefined") {

    Polygon.prototype.getArea = function() {
      return 0;
    };
  }
}

```

```

        Polygon._initialized = true;
    }
}

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.iBase = iBase;
    this.iHeight = iHeight;

    if (typeof Triangle._initialized == "undefined") {
        Triangle.prototype = new Polygon();
        Triangle.prototype.getArea = function() {
            return 0.5 * this.iBase * this.iHeight;
        };

        Triangle._initialized = true;
    }
}

```

W powyższym kodzie widzimy klasy Polygon i Triangle zdefiniowane przy użyciu dynamicznych prototypów. Błąd tkwi w wyróżnionym wierszu, w którym tworzony jest Triangle.prototype. Z logicznego punktu widzenia miejsce tworzenia prototypu jest dobre, ale funkcjonalnie kod nie będzie działać. Dokładnie chodzi o to, że w chwili wykonywania tego kodu, obiekt ten będzie już stworzony i związany z oryginalnym obiektem prototype. Mimo że zmiany w obiekcie prototypu zostaną uwzględnione prawidłowo dzięki mechanizmowi „bardzo późnego wiązania”, zastąpienie obiektu prototype nie będzie miało wpływu na ten obiekt. Zmiany zostaną uwzględnione tylko w tworzonych później egzemplarzach obiektu, a pierwszy egzemplarz pozostanie niepoprawny.

Aby prawidłowo korzystać z dynamicznych prototypów i dziedziczenia, konieczne jest przypisanie nowego obiektu prototype poza obszarem konstruktora:

```

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.iBase = iBase;
    this.iHeight = iHeight;

    if (typeof Triangle._initialized == "undefined") {
        Triangle.prototype.getArea = function () {
            return 0.5 * this.iBase * this.iHeight;
        };

        Triangle._initialized = true;
    }
}

Triangle.prototype = new Polygon();

```

Kod ten działa, ponieważ obiekt prototype zostaje przypisany, zanim powstaną jakiegokolwiek egzemplarze obiektów. Niestety oznacza to, że kod nie będzie w pełni zhermetyzowany w konstruktorze, a w końcu właśnie to jest głównym celem metody dynamicznych prototypów.

Alternatywne wzorce dziedziczenia

Z uwagi na ograniczenia możliwości realizacji dziedziczenia w języku ECMAScript (choćby wynikające z braku zakresu prywatnego i braku łatwego dostępu do metod nadklasy), programiści z całego świata stale próbują eksperymentować z kodem, szukając własnych sposobów na implementację dziedziczenia. W tym punkcie przyjrzymy się kilku alternatywom dla standardowych wzorców dziedziczenia w języku ECMAScript.

zInherit

Wiązanie łańcuchowe prototypów w istocie kopiuje wszystkie metody z obiektu do obiektu reprezentującego prototyp klasy (prototype). A może istnieje inny sposób osiągnięcia tego efektu? Otóż istnieje. Za pomocą biblioteki **zInherit** (dostępnej pod adresem <http://www.nczonline.net/downloads>) możliwa jest realizacja dziedziczenia metod bez korzystania z wiązania łańcuchowego prototypów. Ta niewielka biblioteka obsługuje wszystkie współczesne przeglądarki (Mozilla, IE, Opera, Safari) oraz niektóre starsze (Netscape 4.x, IE/Mac).

Aby móc korzystać z biblioteki zInherit, trzeba dołączyć plik zinherit.js znacznikiem `<script/>`. Dołączanie zewnętrznych plików JavaScriptu zostało omówione szczegółowo w rozdziale 5., „JavaScript w przeglądarce”.

Biblioteka **zInherit** dodaje do klasy `Object` dwie metody: `inheritFrom()` i `instanceOf()`. Metoda `inheritFrom()` zajmuje się „pracą fizyczną”, kopiując metody z danej klasy. Oto wiersz kodu, który realizuje dziedziczenie metod klasy `ClassA` przez klasę `ClassB`, używając wiązania łańcuchowego prototypów:

```
ClassB.prototype = new ClassA();
```

Wiersz ten można zastąpić teraz następującym:

```
ClassB.prototype.inheritFrom(ClassA);
```

Metoda `inheritFrom()` przyjmuje jeden argument, będący nazwą klasy, z której mają być skopiowane metody. Zauważmy, że w odróżnieniu od wiązania łańcuchowego prototypów ten wzorec nie tworzy nawet nowego egzemplarza klasy, z której dziedziczy, sprawiając że proces jest bezpieczniejszy, a programista nie musi przejmować się argumentami dla konstruktora.

Wywołanie metody `inheritFrom()` musi następować dokładnie tam, gdzie normalnie następuje przypisanie prototypu, aby dziedziczenie mogło działać poprawnie.

Metoda `instanceOf()` zastępuje operator `instanceof`. Ponieważ wzorec ten nie korzysta w ogóle z wiązania łańcuchowego prototypów, poniższy wiersz kodu nie będzie działać:

```
ClassB instanceof ClassA
```

Rekompensuje to metoda `instanceOf()`, współpracując z `inheritFrom()` i śledząc wszystkie nadklasy:

```
ClassB.instanceOf(ClassA);
```


Wielokąty kontratakują

Cały przykład z wielokątami można przepisać, korzystając z biblioteki *zInherit*, zastępując w nim tylko dwa wiersze (wyróżnione):

```
function Polygon(iSides) {
    this.iSides = iSides;
}

Polygon.prototype.getArea = function () {
    return 0;
};

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.iBase = iBase;
    this.iHeight = iHeight;
}
```

```
Triangle.prototype.inheritFrom(Polygon);
```

```
Triangle.prototype.getArea = function () {
    return 0.5 * this.iBase * this.iHeight;
};
```

```
function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.iLength = iLength;
    this.iWidth = iWidth;
}
```

```
Rectangle.prototype.inheritFrom(Polygon);
```

```
Rectangle.prototype.getArea = function () {
    return this.iLength * this.iWidth;
};
```

Aby przetestować ten kod, możemy posłużyć się tym samym przykładem co wcześniej, dodając kilka dodatkowych wierszy testujących metodę `instanceOf()`:

```
var oTriangle = new Triangle(12, 4);
var oRectangle = new Rectangle(22, 10);
```

```
alert(oTriangle.iSides);
alert(oTriangle.getArea());
```

```
alert(oRectangle.iSides);
alert(oRectangle.getArea());
```

```
alert(oTriangle.instanceOf(Triangle)); // wyświetla "true"
alert(oTriangle.instanceOf(Polygon)); // wyświetla "true"
alert(oRectangle.instanceOf(Rectangle)); // wyświetla "true"
alert(oRectangle.instanceOf(Polygon)); // wyświetla "true"
```

Ostatnie cztery wiersze testują metodę `instanceOf()` i w każdym przypadku powinny zwrócić `true`.

Obsługa metody prototypów dynamicznych

Jak już wspomniano, wiązania łańcuchowego prototypów nie da się zastosować z zachowaniem ducha metody prototypów dynamicznych, który sprowadza się do utrzymania całego kodu klasy w obrębie konstruktora. Biblioteka *zInherit* poprawia ten problem, umożliwiając wywoływanie metody `inheritFrom()` z wnętrza konstruktora.

Spójrzmy na użyty wcześniej przykład klas wielokątów zapisanych metodą prototypów dynamicznych, tym razem uzupełniony o możliwości biblioteki *zInherit*:

```
function Polygon(iSides) {
    this.iSides = iSides;

    if (typeof Polygon._initialized == "undefined") {

        Polygon.prototype.getArea = function() {
            return 0;
        };

        Polygon._initialized = true;
    }
}

function Triangle(iBase, iHeight) {
    Polygon.call(this, 3);
    this.iBase = iBase;
    this.iHeight = iHeight;

    if (typeof Triangle._initialized == "undefined") {

        Triangle.prototype.inheritFrom(Polygon);
        Triangle.prototype.getArea = function() {
            return 0.5 * this.iBase * this.iHeight;
        };

        Triangle._initialized = true;
    }
}

function Rectangle(iLength, iWidth) {
    Polygon.call(this, 4);
    this.iLength = iLength;
    this.iWidth = iWidth;

    if (typeof Rectangle._initialized == "undefined") {

        Rectangle.prototype.inheritFrom(Polygon);
        Rectangle.prototype.getArea = function () {
            return this.iLength * this.iWidth;
        };

        Rectangle._initialized = true;
    }
}
```

Dwa wyróżnione wiersze w powyższym kodzie implementują dziedziczenie klasy Polygon dla dwóch potomków — klas Triangle i Rectangle. Kod działa, ponieważ tym razem obiekt prototype nie został nadpisany, co wynika z zastosowania metody inheritFrom(). Dodano do niego tylko metody. Tym sposobem możliwe jest ominięcie ograniczeń wiązania łańcuchowego prototypów i zaimplementowanie prototypów dynamicznie zgodnie z duchem tego wzorca.

Obsługa wielokrotnego dziedziczenia

Jedną z najbardziej przydatnych możliwości biblioteki *zInherit* jest obsługa dziedziczenia wielokrotnego, które nie jest dostępne przy stosowaniu wiązania łańcuchowego prototypów. Ponownie kluczowym czynnikiem, który to umożliwia, jest to, że inheritFrom() nie zastępuje obiektu prototype.

Aby dziedziczyć metody i właściwości, metoda inheritFrom() musi zostać użyta w powiązaniu z maskowaniem obiektów. Weźmy następujący przykład:

```
function ClassX() {
    this.sMessageX = "To jest komunikat X.";

    if (typeof ClassX._initialized == "undefined") {

        ClassX.prototype.sayMessageX = function() {
            alert(this.sMessageX);
        };

        ClassX._initialized = true;
    }
}

function ClassY() {
    this.sMessageY = "To jest komunikat Y.";

    if (typeof ClassY._initialized == "undefined") {

        ClassY.prototype.sayMessageY = function () {
            alert(this.sMessageY);
        };

        ClassY._initialized = true;
    }
}
```

ClassX i ClassY to małe klasy z jedną właściwością i jedną metodą. Powiedzmy, że stworzyliśmy klasę ClassZ, która ma być potomkiem obu tych klas. Klasę tą można zdefiniować następująco:

```
function ClassZ() {
    ClassX.apply(this);
    ClassY.apply(this);
    this.sMessageZ = "To jest komunikat Z.";

    if (typeof ClassZ._initialized == "undefined") {
```

```

ClassZ.prototype.inheritFrom(ClassX);
ClassZ.prototype.inheritFrom(ClassY);

ClassZ.prototype.sayMessageZ = function () {
    alert(this.sMessageZ);
};

ClassZ._initialized = true;
}
}

```

Zwróćmy uwagę, że pierwsze dwa wyróżnione wiersze dziedziczą właściwości (metodą `apply()`), a dwa kolejne wyróżnione wiersze dziedziczą metody (metodą `inheritFrom()`). Jak już wspomniano, ważna jest kolejność, w jakiej następuje dziedziczenie i generalnie lepiej dziedziczyć metody w tej samej kolejności co właściwości (co oznacza, że jeżeli właściwości są dziedziczone przez klasę `ClassX`, a potem przez `ClassY`, to metody powinny być dziedziczone przez klasy w tej samej kolejności).

Następujący kod testuje działanie przykładu z dziedziczeniem wielokrotnym:

```

var oObjZ = new ClassZ();
oObjZ.sayMessageX(); // wyświetla "To jest komunikat X."
oObjZ.sayMessageY(); // wyświetla "To jest komunikat Y."
oObjZ.sayMessageZ(); // wyświetla "To jest komunikat Z."

```

Powyższy kod wywołuje trzy metody:

1. Metoda `sayMessageX()`, odziedziczona po klasie `ClassX`, odwołuje się do właściwości `sMessageX`, także odziedziczonej po klasie `ClassX`.
2. Metoda `sayMessageY()`, odziedziczona po klasie `ClassY`, odwołuje się do właściwości `sMessageY`, także odziedziczonej po klasie `ClassY`.
3. Metoda `sayMessageZ()`, zdefiniowana w klasie `ClassX`, odwołuje się do właściwości `sMessageZ`, także zdefiniowanej w klasie `ClassZ`.

Te trzy metody powinny wyświetlić odpowiednie komunikaty pobrane z odpowiednich właściwości, dowodząc, że dziedziczenie wielokrotne działa.

xbObject

Strona DevEdge należąca do Netscape'a (<http://devedge.netscape.com>) zawiera wiele przydatnych informacji i narzędzi wspomagających pisanie skryptów dla programistów sieci WWW. Jedno z takich narzędzi to *xbObject* (można je pobrać pod adresem <http://archive.bclary.com/xbProjects-docs/xbObject/>), napisane przez Boba Clary'ego z firmy Netscape Communications w 2001 roku, kiedy pojawiła się przeglądarka Netscape 6 (Mozilla 0.6). Narzędzie współpracuje ze wszystkimi wersjami Mozilli, które pojawiły się od tamtego czasu oraz z innymi współcześnie używanymi przeglądarkami (IE, Opera, Safari).

Przeznaczenie

Narzędzie *xObject* ma z założenia udostępniać lepszy model obiektowy w języku JavaScript, umożliwiając nie tylko dziedziczenie, ale również przeładowywanie metod oraz możliwość wywoływania metod nadklasy. W tym celu *xObject* wymaga przejścia przez kilka kroków.

Na początek musimy **zarejestrować** klasę i przy okazji zdefiniować, której klasy ma być potomkiem. Wymaga to następującego wywołania:

```
_classes.registerClass("Nazwa_Podklasy", "Nazwa_Nadklasy");
```

Nazwy podklasy i nadklasy są tu przesyłane jako ciągi znakowe, a nie jako wskaźniki do swoich konstruktorów. Wywołanie to musi następować przed konstruktorem danej podklasy.

Można też wywołać registerClass() z jednym tylko argumentem, jeżeli nowa klasa nie jest potomkiem żadnej innej klasy.

Drugi krok to wywołanie w obrębie konstruktora metody defineClass(), z przesłaniem nazwy klasy oraz wskaźnika na coś, co Clary nazywa **funkcją prototypową**, służącą do inicjalizacji wszystkich właściwości i metod obiektu (o tym później). Na przykład:

```
_classes.registerClass("ClassA");
```

```
function ClassA(sColor) {
  _classes.defineClass("ClassA", prototypeFunction);

  function prototypeFunction() {
    //...
  }
}
```

Jak widać, funkcja prototypowa (nazwana tu prototypeFunction()) pojawia się w treści konstruktora. Jej głównym celem jest przypisanie wszystkich metod do klasy w stosownym czasie (w tym sensie działa jak dynamiczne prototypy).

Kolejny krok (jak dotąd trzeci) to wywołanie metody init() dla klasy. Metoda ta jest odpowiedzialna za ustawienie wszystkich właściwości dla klasy i musi przyjmować takie same argumenty jak sam konstruktor. Zgodnie z konwencją metoda init() jest wywoływana zawsze po wywołaniu metody defineClass(). Na przykład:

```
_classes.registerClass("ClassA");

function ClassA(sColor) {
  _classes.defineClass("ClassA", prototypeFunction);

  this.init(sColor);

  function prototypeFunction() {

    ClassA.prototype.init = function (sColor) {
      this.parentMethod("init");
      this.sColor = sColor;
    };
  }
}
```

```

    };
  }
}

```

Widzimy tu metodę `parentMethod()` wywoływaną w metodzie `init()`. W ten właśnie sposób *xbObject* umożliwia klasom wywoływanie metod nadklasy. Metoda `parentMethod()` przyjmuje dowolną ilość argumentów, ale pierwszy argument jest zawsze nazwą metody klasy nadrzędnej, która ma być wywołana (argument ten musi być ciągiem, a nie wskaźnikiem funkcji). Wszystkie następnne argumenty zostaną przesłane do metody nadklasy.

W tym przypadku najpierw wywoływana jest metoda `init()` nadklasy, co jest wymagane dla działania *xbObject*. Mimo że klasa `ClassA` nie rejestrowała żadnej nadklasy, *xbObject* tworzy domyślną nadklasę dla wszystkich klas i stamtąd właśnie pochodzi metoda `init()` z nadklasy.

Czwarty i ostatni krok polega na dodaniu metod innej klasy w obrębie funkcji prototypowej:

```

classes.registerClass("ClassA");

function ClassA(sColor) {
  _classes.defineClass("ClassA", prototypeFunction);

  this.init(sColor);

  function prototypeFunction() {

    ClassA.prototype.init = function (sColor) {
      this.parentMethod("init");
      this.sColor = sColor;
    };

    ClassA.prototype.sayColor = function () {
      alert(this.sColor);
    };

  }
}

```

Teraz możemy w normalny sposób stworzyć egzemplarz klasy `ClassA`:

```

var oObjA = new ClassA("czerwony");
oObjA.sayColor(); // wyświetla "czerwony"

```

Wielokąty — ostateczna rozgrywka

W tym momencie pewnie zastanawiasz się, czy będziesz miał okazję ujrzeć przykład z wielokątami przerobiony za pomocą *xbObject*. Jak najbardziej!

Na początku poprawiamy klasę `Polygon`, co jest bardzo proste:

```

_classes.registerClass("Polygon");

function Polygon(iSides) {

  _classes.defineClass("Polygon", prototypeFunction);

```

```

    this.init(iSides);

    function prototypeFunction() {

        Polygon.prototype.init = function(iSides) {
            this.parentMethod("init");
            this.iSides = iSides;
        };

        Polygon.prototype.getArea = function () {
            return 0;
        };

    }
}

```

Teraz przepisujemy klasę Triangle i czujemy w tym przykładzie pierwszy smak prawdziwego dziedziczenia:

```

_classes.registerClass("Triangle", "Polygon");

function Triangle(iBase, iHeight) {

    _classes.defineClass("Triangle", prototypeFunction);

    this.init(iBase, iHeight);

    function prototypeFunction() {
        Triangle.prototype.init = function(iBase, iHeight) {
            this.parentMethod("init", 3);
            this.iBase = iBase;
            this.iHeight = iHeight;
        };

        Triangle.prototype.getArea = function () {
            return 0.5 * this.iBase * this.iHeight;
        };
    }

}

```

Zwróćmy uwagę na wywołanie registerClass() tuż przed konstruktorem, gdzie tworzone jest powiązanie dziedziczne. Dodatkowo w pierwszym wierszu metody init() wywoływana jest metoda init() nadklasy (Polygon) z argumentem 3, który ustawia właściwość iSides na 3. Poza tym metoda init() jest bardzo podobna: prosty konstruktor przypisujący iBase i iHeight.

Klasa Rectangle wygląda ostatecznie bardzo podobnie do klasy Triangle:

```

_classes.registerClass("Rectangle", "Polygon");

function Rectangle(iLength, iWidth) {

    _classes.defineClass("Rectangle", prototypeFunction);

    this.init(iLength, iWidth);

```

```
function prototypeFunction() {
    Rectangle.prototype.init = function(iLength, iWidth) {
        this.parentMethod("init", 4);
        this.iLength = iLength;
        this.iWidth = iWidth;
    }

    Rectangle.prototype.getArea = function () {
        return this.iLength * this.iWidth;
    };
}
}
```

Podstawowa różnica między tą klasą a klasą `Triangle` (poza innymi wywołaniami `registerClass()` i `defineClass()`) to wywołanie metody `init()` z nadklasy z argumentem 4. Poza tym dodane zostały właściwości `iLength` i `iWidth` i zastąpiona została metoda `getArea()`.

Podsumowanie

W rozdziale tym zapoznałeś się z koncepcją dziedziczenia w języku ECMAScript (a tym samym w JavaScriptcie) korzystającego z maskowania obiektów i wiązania łańcuchowego prototypów. Dowiedziałeś się też, że łączne stosowanie tych metod jest optymalnym sposobem na stworzenie struktury dziedzicznej klas.

Przedstawiono tu też dwie alternatywne metody uzyskiwania dziedziczenia: *zInherit* i *xbObject*. Te darmowe biblioteki JavaScriptu dostępne do pobrania w internecie wprowadzają nowe i inne możliwości tworzenia struktur dziedzicznych.

W ten sposób kończę omawianie ECMAScriptu — rdzenia języka JavaScript. W następnych rozdziałach, opierając się na tym fundamencie, będziesz poznawał bardziej specyficzne dla sieci WWW aspekty tego języka.