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

JavaScript - mocne strony

Autor: Douglas Crockford
ISBN: 978-83-246-1998-6
Tytuł oryginału: JavaScript: The Good Parts
Format: 168x237, stron: 160

Poznaj doskonałą użyteczność języka JavaScript!

- Jak efektywnie wykorzystać najlepsze funkcje JavaScript?
- Jak pisać programy, aby ustrzec się błędów?
- Jak zdefiniować podzbiór języka i tworzyć idealne aplikacje?

Warto poznać język JavaScript, ponieważ stanowi on jedno z ważniejszych narzędzi w informatyce – dzięki temu, że jest jednocześnie podstawowym i domyślnym językiem przeglądarek internetowych oraz językiem programowania. JavaScript pozwala na tworzenie wydajnego kodu bibliotek obiektowych czy aplikacji opartych na technice AJAX. Jego skrypty służą najczęściej do zapewniania interaktywności, sprawdzania poprawności formularzy oraz budowania elementów nawigacyjnych. Dość łatwa składnia sprawia, że pisanie pełnoprawnych i wydajnych aplikacji w tym języku nie jest trudne nawet dla początkujących programistów.

Książka „JavaScript – mocne strony" to wyjątkowy podręcznik do nauki tego popularnego, dynamicznego języka programowania. Dowiesz się z niej, jak efektywnie wykorzystać wszystkie jego mocne strony (m.in. funkcje, dynamiczne obiekty, literały obiektowe) oraz jak unikać pułapek. Poznasz elementy składowe języka oraz sposoby ich łączenia, zrozumiesz, na czym polega dziedziczenie prototypowe, w jaki sposób brak kontroli typów ma pozytywny wpływ na pisanie aplikacji oraz dlaczego stosowanie zmiennych globalnych jako podstawowego modelu programowania nie jest dobrym pomysłem. Znając wszelkie ograniczenia języka JavaScript, będziesz mógł profesjonalnie wykorzystać jego najlepsze części.

- Gramatyka języka JavaScript
- Obiekty i funkcje
- Rekurencja
- Kaskadowe łączenie wywołań
- Literały obiektowe
- Dziedziczenie – pseudoklasyczne, prototypowe, funkcyjne
- Tablice
- Wyrażenia regularne
- Klasa znaków i kwantyfikator wyrażenia regularnego

Nie trać czasu – sięgaj tylko po to, co najlepsze w języku JavaScript!

Spis treści

ROZDZIAŁ 5.

Dziedziczenie

Dziedziczenie jest ważnym zagadnieniem w większości języków programowania.

W językach klasycznych (takich jak Java) dziedziczenie ma dwa główne zadania. Po pierwsze, jest formą wielokrotnego użycia kodu. Jeśli nowa klasa jest podobna do istniejącej, wystarczy określić dzielące je różnice. Ponowne wykorzystanie kodu jest ważne, ponieważ pozwala zmniejszyć koszty wytwarzania oprogramowania. Drugą zaletą dziedziczenia jest to, że zawiera ono w sobie specyfikację systemu typów. Uwalnia to programistów w dużej mierze od konieczności rzutowania z jednego typu na drugi, co jest istotną zaletą, ponieważ rzutowanie podważa całą wartość bezpieczeństwa systemu typów.

JavaScript, jako język bez kontroli typów, nigdy nie wymaga rzutowania. Hierarchia dziedziczenia obiektu nie ma tu znaczenia. Ważne jest, co obiekty potrafią robić, a nie po czym dziedziczą.

JavaScript posiada dużo bogatszy zasób możliwości ponownego wykorzystania kodu. Może naśladować klasyczne wzorce, ale dostarcza również innych, bardziej ekspresyjnych. Zbiór możliwych wzorców dziedziczenia w JavaScripcie jest bardzo szeroki. W tym rozdziale przyjrzymy się kilku najprostszym przypadkom. Dużo bardziej skomplikowane są również możliwe, ale zazwyczaj lepiej jest trzymać się tych najprostszych.

W klasycznych językach obiekty są instancjami klas, a klasa może dziedziczyć po innej klasie. JavaScript jest językiem prototypowym, co oznacza, że obiekty dziedziczą bezpośrednio z innych obiektów.

Dziedziczenie pseudoklasyczne

Język JavaScript ma wewnętrznie głęboko rozdartą naturę. Jego mechanizm prototypowy jest zaciemniany przez niektóre skomplikowane elementy składni, które wyglądają bardziej klasycznie. Zamiast pozwolić obiektom dziedziczyć bezpośrednio z innych obiektów, JavaScript wprowadza niepotrzebny poziom abstrakcji, w którym obiekty tworzone są przy użyciu funkcji konstruktorów.

Kiedy tworzony jest obiekt funkcji, konstruktor `Function` zwracający obiekt funkcji wykonuje kod podobny do poniższego:

```
this.prototype = {constructor: this};
```

Nowy obiekt funkcji otrzymuje właściwość `prototype`, której wartością jest obiekt posiadający właściwość `constructor`, której to z kolei wartością jest nowy obiekt funkcji. Obiekt `prototype` jest miejscem, gdzie złożone mają być odziedziczone właściwości. Każda funkcja otrzymuje obiekt `prototype`, ponieważ język nie posiada sposobu określenia, które funkcje są przeznaczone do użycia w roli konstruktorów. Właściwość `constructor` nie jest zbyt użyteczna. To obiekt `prototype` ma znaczenie.

Kiedy funkcja jest wywoływana według wzorca wywołania konstruktora z użyciem słowa `new`, zmienia się sposób wykonania funkcji. Gdyby operator `new` był metodą, a nie operatorem, mógłby być zaimplementowany w ten sposób:

```
Function.method('new', function ( ) {

    // Tworzymy nowy obiekt dziedziczący z prototypu konstruktora.

    var that = Object.beget(this.prototype);

    // Wywołujemy konstruktor, wiążąc this do nowego obiektu.

    var other = this.apply(that, arguments);

    // Jeśli zwracana wartość nie jest obiektem,
    // podmień ją na nowo utworzony obiekt.

    return (typeof other === 'object' && other) || that;
});
```

Możemy zdefiniować konstruktor i rozszerzyć jego prototyp:

```
var Mammal = function (name) {
    this.name = name;
};

Mammal.prototype.get_name = function ( ) {
    return this.name;
};

Mammal.prototype.says = function ( ) {
    return this.saying || '';
};
```

Tworzymy instancję:

```
var myMammal = new Mammal('Mój ssak');
var name = myMammal.get_name( ); // 'Mój ssak'
```

Następnie możemy utworzyć inną pseudoklasę dziedziczącą z `Mammal`, definiując jej konstruktor i zastępując jej prototyp instancją `Mammal`:

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'miau';
};

// Zastępujemy Cat.prototype instancją Mammal

Cat.prototype = new Mammal( );

// Rozszerzamy nowy prototyp metodami purr i get_name

Cat.prototype.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
```

```
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};
Cat.prototype.get_name = function () {
        return this.says() + ' ' + this.name + ' ' + this.says();
};

var myCat = new Cat('Kicia');
var says = myCat.says();  // 'miau'
var purr = myCat.purr(5);  // 'r-r-r-r-r'
var name = myCat.get_name();  // 'miau Kicia miau'
```

Pseudoklasyczne dziedziczenie miało w zamiarach wyglądać bardziej obiektowo, lecz w praktyce wygląda dziwnie i obco. Część brzydoty tego rozwiązania możemy ukryć, używając metody `method` i definiując metodę `inherits`:

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent();
    return this;
});
```

Obie te metody zwracają `this`, więc możemy je wykorzystać przy łączeniu wywołań. Możemy teraz utworzyć nasz obiekt `Cat` za pomocą jednej instrukcji[1]:

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'miau';
}.
    inherits(Mammal).
    method('purr', function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
            if (s) {
                s += '-';
            }
            s += 'r';
        }
        return s;
    }).
    method('get_name', function () {
        return this.says() + ' ' + this.name + ' ' + this.says();
    });
```

Dzięki ukryciu całego mechanizmu operującego na obiekcie `prototype`, ten kod wygląda nieco mniej obco. Ale czy naprawdę coś ulepszyliśmy? Mamy teraz funkcje konstruktorów przypominające klasy, ale pod tą powierzchnią wciąż możemy się natknąć na nieprzewidziane zachowanie. Nie mamy prywatnego dostępu — wszystkie właściwości są publiczne. Nie mamy możliwości wywoływania nieprzesłoniętych wersji metod obiektów nadrzędnych z poziomu obiektów przesłaniających te metody.

Co gorsza, istnieje pewne ryzyko związane z użyciem funkcji konstruktorów. Jeśli zapomnimy użyć słowa `new` przy wywoływaniu konstruktora, `this` nie będzie powiązane z nowym

[1] Kod ten w celu poprawnego nadpisania metody `get_name` wymaga użycia implementacji metody `method` podanej na końcu rozdziału 1, a nie jej modyfikacji warunkowej pokazanej na końcu podrozdziału „Rozszerzanie typów" w rozdziale 4. — *przyp. tłum.*

obiektem, lecz z obiektem globalnym. Zamiast więc dodać właściwości do nowego obiektu, zaśmiecimy nimi globalną przestrzeń nazw. To bardzo niebezpieczna możliwość, tym bardziej, że nie otrzymamy żadnego ostrzeżenia ani podczas kompilacji, ani podczas wykonania kodu.

Jest to poważny błąd projektowy języka. Aby sobie jakoś z nim radzić, istnieje konwencja nakazująca nadawanie wszystkim funkcjom pełniącym rolę konstruktorów (i żadnym innym) nazw zaczynających się od wielkich liter. To pozwala przynajmniej mieć nadzieję, że analiza kodu pozwoli wyłapać wszystkie brakujące zastosowania `new`. Lepszą alternatywą jest jednak nie używanie w ogóle słowa `new`.

Pseudoklasyczna forma dziedziczenia pozwala poczuć się wygodniej programistom nieznającym dobrze JavaScriptu, ale równocześnie ukrywa ona prawdziwą naturę tego języka. Notacja inspirowana podejściem klasycznym może zachęcać do tworzenia przesadnie głębokich i skomplikowanych hierarchii. Większość takich skomplikowanych hierarchii klas spowodowana jest ograniczeniami statycznej kontroli typów. JavaScript nie ma takich ograniczeń. W językach klasycznych dziedziczenie klas jest jedyną formą wielokrotnego wykorzystania kodu. JavaScript ma dużo szersze możliwości.

Określenia obiektów

Czasami zdarza się, że konstruktor pobiera bardzo dużą liczbę parametrów. Może to być kłopotliwe, bo na przykład utrudnia zapamiętanie kolejność argumentów. W takich wypadkach lepszym rozwiązaniem może być napisanie konstruktora, który pobiera pojedyncze **określenie obiektu** (ang. *object specifier*). Określenie takie zawiera specyfikację obiektu, który ma być skonstruowany. Tak więc, zamiast takiego wywołania funkcji:

```
var myObject = maker(f, l, m, c, s);
```

możemy napisać:

```
var myObject = maker({
   first: f,
   last: l,
   state: s,
   city: c
});
```

Argumenty mogą być teraz wymienione w dowolnej kolejności, mogą być pominięte, jeśli konstruktor jest w stanie przypisać im wartości domyślne, a cały kod jest łatwiejszy w czytaniu.

Podejście to może mieć dodatkowe zalety przy korzystaniu z formatu JSON (patrz dodatek E). Tekst formatu JSON jest w stanie opisać jedynie dane, ale czasami dane reprezentują jakiś obiekt i wygodnie byłoby powiązać dane z jego metodami. Okazuje się to banalne, gdy konstruktor pobiera określenie obiektu, ponieważ możemy po prostu przesłać obiekt JSON do konstruktora, który z kolei zwróci w pełni utworzony obiekt.

Dziedziczenie prototypowe

W wypadku podejścia czysto prototypowego, nie używamy w ogóle klas. Skupiamy się wyłącznie na obiektach. Dziedziczenie prototypowe jest koncepcyjnie prostsze od klasycznego: nowy obiekt może dziedziczyć właściwości starego obiektu. Jest to może mniej powszechne podejście, ale jest za to bardzo łatwe do zrozumienia. Punktem wyjścia jest utworzenie jakiegoś

pożytecznego obiektu. W następnym kroku możemy utworzyć wiele obiektów podobnych do niego. Proces klasyfikacji polegający na rozbiciu aplikacji na zbiór zagnieżdżonych klas abstrakcyjnych może być całkowicie pominięty.

Zacznijmy więc od utworzenia takiego pożytecznego obiektu, używając literału obiektowego:

```
var myMammal = {
   name : "Mój ssak",
   get_name : function () {
      return this.name;
   },
   says : function () {
      return this.saying || '';
   }
};
```

Mając obiekt taki jak ten, możemy utworzyć więcej instancji korzystając z funkcji `Object.beget` z rozdziału 3. Następnie możemy je dostosować do naszych potrzeb:

```
var myCat = Object.beget(myMammal);
myCat.name = 'Kicia';
myCat.saying = 'miau';
myCat.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
       if (s) {
          s += '-';
       }
       s += 'r';
    }
    return s;
};
myCat.get_name = function () {
   return this.says() + ' ' + this.name + ' ' + this.says();
};
```

Jest to **dziedziczenie różnicowe**. Zdefiniowanie nowego obiektu polega na określeniu różnic między nim a obiektem, z którego został utworzony.

Czasami wygodnie jest, gdy struktura danych dziedziczy z innej struktury. Oto przykład: przypuśćmy, że analizujemy kod języka takiego jak Java lub TEX, w którym para nawiasów oznacza zasięg. Zmienne zdefiniowane w ramach zasięgu nie są widoczne poza nim. W pewnym sensie wewnętrzny zasięg dziedziczy po zewnętrznym. Obiekty JavaScriptu dobrze się nadają do reprezentacji takiej zależności. Funkcja `block` wywoływana jest, kiedy napotkany zostaje lewy nawias klamrowy. Funkcja `parse` będzie pobierać symbole z zasięgu oraz dodawać nowo napotkane:

```
var block = function () {

    // Zapamiętujemy dotychczasowy zasięg. Tworzymy nowy zasięg,
    // który będzie zawierał wszystko to, co dotychczasowy.

    var oldScope = scope;
    scope = Object.beget(scope);

    // Przechodząc nad lewym nawiasem klamrowym wchodzimy do nowego zasięgu.

    advance('{');

    // Analizujemy tekst, używając nowego zasięgu.

    parse(scope);
```

```
    // Przechodząc nad prawym nawiasem klamrowym wychodzimy z zasięgu
    // i przywracamy stary zasięg.

    advance('}');
    scope = oldScope;
};
```

Dziedziczenie funkcyjne

Przedstawione dotąd wzorce dziedziczenia mają pewien słaby punkt: brak prywatności. Wszystkie właściwości obiektów są widoczne. Brakuje zmiennych prywatnych oraz prywatnych metod. Czasami nie ma to większego znaczenia, ale czasami może mieć znaczenie ogromne. Zniechęceni tym faktem niektórzy niedoinformowani programiści ukuli zasadę używania zmiennych **niby-prywatnych**. Mając właściwość, którą chcieli uczynić prywatną, nadawali jej dziwnie wyglądającą nazwę, mając nadzieję, że inni użytkownicy ich kodu będą udawać, że nie widzą nazw dziwnie wyglądających. Na szczęście istnieje dużo lepsza alternatywa oparta o omawiany wcześniej wzorzec modułu.

Zaczynamy od napisania funkcji, która będzie wytwarzać obiekty. Dajemy jej nazwę zaczynającą się od małej litery, ponieważ nie będzie ona wymagać użycia słowa `new`. Działanie funkcji składa się z czterech kroków:

- Funkcja tworzy nowy obiekt. Jest na to wiele sposobów: użycie literału obiektowego, wywołanie funkcji konstruktora ze słowem `new`, użycie metody `Object.beget` do utworzenia nowego obiektu na podstawie istniejącego, wreszcie wywołanie dowolnej innej funkcji zwracającej obiekt.
- Opcjonalnie funkcja deklaruje zmienne i metody prywatne. Są to zwykłe zmienne (`var`) funkcji.
- Następnie funkcja rozszerza nowo utworzony obiekt o metody. Metody te mają uprzywilejowany dostęp do parametrów i zmiennych zdefiniowanych w poprzednim kroku.
- Na koniec nowy obiekt jest zwracany.

Oto szablon pseudokodu do tworzenia konstruktora funkcyjnego (pogrubiony tekst dodano dla podkreślenia niektórych fragmentów):

```
var constructor = function (spec, my) {
    vat that, inne prywatne zmienne instancyjne;
    my = my || {};

    // tu dodajemy do obiektu my zmienne i funkcje współdzielone

    that = nowy obiekt;

    // tu dodajemy do obiektu that metody uprzywilejowane

    return that;
}
```

Obiekt `spec` zawiera wszystkie informacje niezbędne konstruktorowi do utworzenia instancji. Zawartość obiektu `spec` może być skopiowana do zmiennych prywatnych lub przetworzona przez inne funkcje, bądź też metody mogą pobierać informacje z tego obiektu w razie potrzeby. (Można też to uprościć zastępując `spec` pojedynczą wartością. Jest to wygodne, gdy tworzony obiekt nie wymaga pełnego obiektu `spec`.)

Obiekt `my` służy przechowywaniu chronionych danych, które mogą być współdzielone z konstruktorami w ramach łańcucha dziedziczenia. Użycie tego obiektu jest opcjonalne. Jeśli nie jest on przekazany do funkcji, tworzony jest pusty.

Następnie deklarujemy prywatne zmienne instancyjne oraz prywatne metody obiektu. Odbywa się to poprzez zwykłe zadeklarowanie zmiennych. Zmienne i funkcje wewnętrzne utworzone wewnątrz konstruktora stają się zmiennymi i funkcjami prywatnymi instancji. Funkcje wewnętrzne mają dostęp do `spec`, `my`, `that` i innych zmiennych prywatnych.

Następnie dodajemy współdzielone dane chronione do obiektu `my`. Wykonuje się to poprzez przypisanie:

```
my.member = value;
```

Teraz tworzymy nowy obiekt i przypisujemy go do `that`. Jest wiele sposobów utworzenia obiektu. Możemy użyć literału obiektowego. Możemy wywołać pseudoklasyczny konstruktor, używając operatora `new`. Możemy wywołać metodę `Object.beget` na prototypie obiektu. Wreszcie możemy skorzystać z dowolnego innego konstruktora funkcyjnego, przekazując mu obiekt `spec` (najczęściej ten sam, który został przekazany do tego konstruktora) oraz obiekt `my`. Obiekt `my` pozwala na współdzielenie informacji z innymi konstruktorami. Inne konstruktory mogą również umieszczać swoje własne chronione dane w tym obiekcie, w celu współdzielenia ich z naszych konstruktorem.

Następnie rozszerzamy `that`, dodając metody uprzywilejowane stanowiące interfejs obiektu. Możemy przypisywać nowe funkcje bezpośrednio do `that`. Lub też, w bardziej bezpieczny sposób, najpierw zdefiniować funkcje jako metody prywatne, a następnie przypisać je do `that`:

```
var methodical = function () {
   ...
};
that.methodical = methodical;
```

Zaletą definiowania metod w dwóch etapach jest to, że jeśli inne metody będą chciały wywoływać `methodical`, będą mogły zrobić to przez wywołanie `methodical()` zamiast `that.metho`
`↪dical()`. Jeśli instancja zostanie uszkodzona lub zmanipulowana, tak że metoda `that.`
`↪methodical` zostanie zastąpiona inną, wówczas funkcje wywołujące funkcję `methodical` będą nadal działać poprawnie, ponieważ funkcja prywatna `methodical` pozostanie nienaruszona mimo modyfikacji obiektu.

Na koniec zwracamy `that`.

Zastosujmy ten wzorzec do naszego przykładu z ssakami. Nie potrzebujemy tutaj zmiennej `my`, więc po prostu ją opuścimy, ale za to przyda nam się obiekt `spec`.

Właściwości `name` i `saying` są teraz całkowicie prywatne. Dostęp do nich jest możliwy tylko dzięki uprzywilejowanym metodom `get_name` i `says`:

```
var mammal = function (spec) {
   var that = {};

   that.get_name = function () {
      return spec.name;
   };

   that.says = function () {
      return spec.saying || '';
   };
```

```
    return that;
};

var myMammal = mammal({name: 'Mój ssak'});
```

W podejściu pseudoklasycznym konstruktor `Cat` musiał duplikować pracę wykonywaną przez konstruktor `Mammal`. We wzorcu funkcyjnym nie jest to konieczne, ponieważ konstruktor `Cat` wywoła konstruktor `Mammal`, który sam wykona swoje zadanie. Konstruktor `Cat` zajmuje się tylko różnicami między nimi:

```
var cat = function (spec) {
    spec.saying = spec.saying || 'miau';
    var that = mammal(spec);
    that.purr = function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
            if (s) {
                s += '-';
            }
            s += 'r';
        }
        return s;
    };
    that.get_name = function () {
        return that.says() + ' ' + spec.name + ' ' + that.says();
    };
return that;
};

var myCat = cat({name: 'Kicia'});
```

Wzorzec funkcyjny umożliwia nam również wywoływanie metod z obiektów nadrzędnych. Napiszmy metodę `superior`, która pobierać będzie nazwę metody i zwracać funkcję wywołującą tę metodę. Funkcja ta będzie wywoływać oryginalną metodę, nawet gdy właściwość została zmieniona przez obiekt potomny:

```
Object.method('superior', function (name) {
    var that = this,
        method = that[name];
    return function () {
        return method.apply(that, arguments);
    };
});
```

Wypróbujmy ją na obiekcie `coolcat`, który jest podobny do obiektu `cat`, ale ma nieco ciekawszą metodę `get_name`, która wywołuje nieprzesłoniętą wersję metody z obiektu nadrzędnego. Wymaga to tylko niewielkich przygotowań. Zadeklarujemy zmienną `super_get_name` i przypiszemy jej wynik wywołania metody `superior`:

```
var coolcat = function (spec) {
    var that = cat(spec),
        super_get_name = that.superior('get_name');
    that.get_name = function (n) {
        return 'Teraz ' + super_get_name() + ' w nowej, lepszej wersji';
    };
    return that;
};

var myCoolCat = coolcat({name: 'Kocur'});
var name = myCoolCat.get_name();  // 'Teraz miau Kocur miau w nowej, lepszej wersji'
```

Wzorzec funkcjonalny jest bardzo elastyczny. Wymaga on mniej pracy niż wzorzec pseudoklasyczny i umożliwia ukrywanie implementacji oraz dostęp do nieprzesłoniętych wersji metod w obiektach nadrzędnych.

Jeśli cały dostęp do stanu obiektu jest prywatny, obiekt taki jest zabezpieczony przed jakąkolwiek manipulacją. Właściwości obiektu mogą być podmieniane lub usunięte, ale integralność obiektu nie zostanie naruszona. Jeśli utworzymy obiekt używając wzorca funkcyjnego i żadna z metod nie używa zmiennych `this` lub `that`, wówczas obiekt będzie **wytrzymały**. Wytrzymały obiekt jest po prostu kolekcją funkcji tworzących **zbiór możliwości**.

Wytrzymałego obiektu nie da się uszkodzić. Obiekt wytrzymały nie pozwala atakującemu na dostęp do jego wewnętrznego stanu, z wyjątkiem dostępu na jaki zezwalają metody.

Części

Obiekty da się tworzyć ze zbiorów części. Na przykład możemy napisać funkcję, która dodaje do dowolnego obiektu podstawowe możliwości obsługi zdarzeń. Dodaje ona metody `on` i `fire` oraz prywatny rejestr zdarzeń:

```
var eventuality = function (that) {
   var registry = {};

   that.fire = function (event) {

      // Wywołujemy zdarzenie na obiekcie. Zdarzenie może być albo
      // łańcuchem zawierającym nazwę zdarzenia lub obiektem zawierającym
      // właściwość type, która przechowuje nazwę zdarzenia. Procedury obsługi
      // zdarzeń zarejestrowane przez metodę on, których nazwa pokrywa się
      // z nazwą zdarzenia, zostaną wywołane.

      var array,
          func,
          handler,
          i,
          type = typeof event === 'string' ? event : event.type;

      // Jeśli tablica procedur istnieje dla tego zdarzenia, wówczas
      // wywołujemy kolejno każdą procedurę z tablicy.

      if (registry.hasOwnProperty(type)) {
         array = registry[type];
         for (i = 0; i < array.length; i += 1) {
            handler = array[i];

            // Obiekt handler zawiera właściwość method i opcjonalną listę parametrów.
            // Jeśli method jest nazwą, wyszukujemy odpowiednią funkcję.

            func = handler.method;
            if (typeof func === 'string') {
               func = this[func];
            }

            // Wywołujemy procedurę. Jeśli były podane parametry, przekazujemy je.
            // Jeśli nie, przekazujemy obiekt zdarzenia.

            func.apply(this, handler.parameters || [event]);
         }
      }
```

```
        return this;
    };

    that.on = function (type, method, parameters) {

        // Rejestrujemy zdarzenie. Tworzymy rekord z procedurą obsługi zdarzenia.
        // Dodajemy go do listy procedur obsługi, tworząc nową, jeśli jeszcze
        // nie istnieje dla tego typu.

        var handler = {
            method: method,
            parameters: parameters
        };
        if (registry.hasOwnProperty(type)) {
            registry[type].push(handler);
        } else {
            registry[type] = [handler];
        }
        return this;
    };
    return that;
};
```

Możemy wywołać funkcję `eventuality` na dowolnym indywidualnym obiekcie, wyposażając go w metody obsługi zdarzeń. Możemy również wywołać ją wewnątrz funkcji konstruktora obiektu przed zwróceniem `that`:

```
eventuality(that);
```

W ten sposób konstruktor może złożyć obiekt jakby z części. Brak kontroli typów JavaScriptu jest wielką zaletą w tym przypadku, ponieważ nie jesteśmy obciążeni systemem typów dbającym o hierarchię dziedziczenia klas. Zamiast tego możemy się skupić na zawartości obiektu.

Jeślibyśmy chcieli zezwolić metodzie `eventuality` na dostęp do prywatnego stanu obiektu, moglibyśmy przekazać jej parametr `my`.