Język JavaScript

JavaScript

- **Podstawowe fakty**
  - „popularny i mało znany” (sic!)
  - geneza
    - pomysł firmy Netscape (1995, Brendan Eich)
    - Mocha → LiveScript → JavaScript
      (Sun + Netscape =  ?)
  - język skryptowy „ogólnego przeznaczenia”
  - interpretery często obecne w programach użytkowych (nie tylko przeglądarki!!!)
  - (w zasadzie) brak związków z językiem **Java**

JavaScript

- **Ważniejsze cechy**
  - składnia wzorowana na C/Java
  - luźne typowanie (ang. *loose typing*)
  - obiektowość
    - obiekty dynamiczne
    - dziedziczenie prototypowe
  - programowanie funkcyjne
    - funkcje „obiektami pierwszego rzędu”
    - mechanizm domknięć (ang. *closures*)

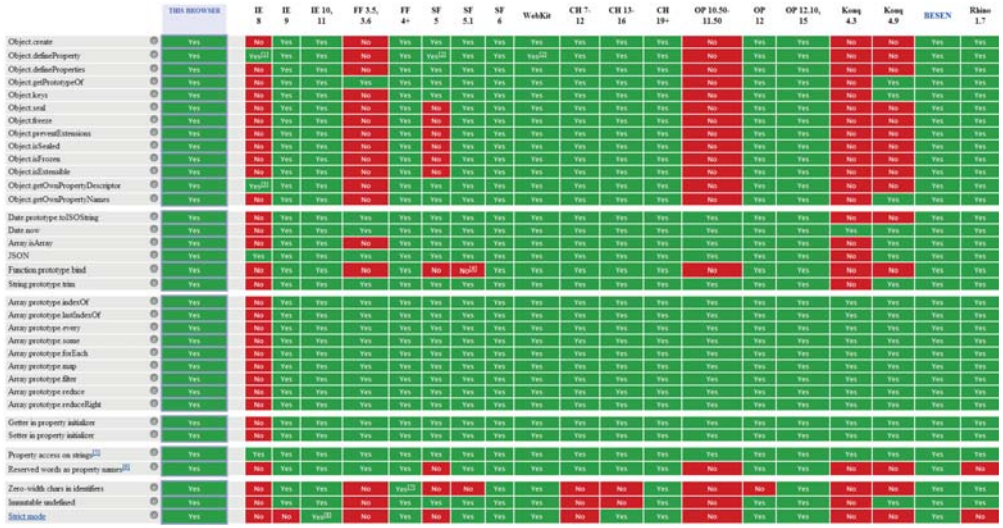
JavaScript – standaryzacja

- **ECMA-262** (`http://www.ecmascript.org`)
  - edycja 3 – podstawa większości implementacji
  - aktualna: edycja 5 (zwana też 3.1) (XII 2009)
- implementacje i „dialekty” – ECMA (3 / „3+”)
  - Netscape (B. Eich) / Mozilla SpiderMonkey (C)
  - Mozilla Rhino (Java)
  - Microsoft JScript .NET 8.0
  - Adobe ActionScript 3, ExtendScript (PS i „koledzy”)

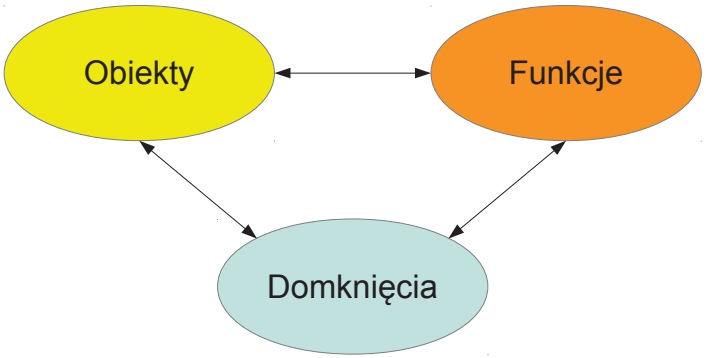
„Silniki" JavaScript w przeglądarkach

- **Firefox** (JIT)
  - SpiderMonkey → TraceMonkey → JägerMonkey
- **Chrome**
  - V8 (JIT)
- **Internet Explorer**
  - 5.5-8.0 JScript; 9, 10, 11 Chakra (JIT)
- **Opera**
  - ≤ 9.0 Linear(A/B); 9.5-10 Futhark; 10.5+ Carakan (JIT)
- **Safari** JavaScriptCore, SquirrelFish (JIT)

Obsługa ECMAScript 5 w przeglądarkach



Najważniejsze mechanizmy



JavaScript – podstawy

- **Składnia**
  - wykorzystywany zestaw znaków: Unicode
  - nazwy – małe i wielkie litery są rozróżniane
  - komentarze
    - `// do końca linii to komentarz`
    - `/* wszystko „pomiędzy" to komentarz */`
  - literały (dla typów „prostych")

```
1.2              // liczba
"hello world"    // łańcuch znaków
'Hi'             // j.w.
true             // wartość logiczna „prawda"
/^javascript/gi  // wyrażenie regularne
```

JavaScript – podstawy

- **Składnia**
  - słowa kluczowe (* – dodane w ECMA Script 5)

```
break       do          instanceof  typeof
case        else        new         var
catch       finally     return      void
continue    for         switch      while
debugger*   function    this        with
default     if          throw
delete      in          try
```

słowo kluczowe – tzn. wykorzystywane w języku

JavaScript – podstawy

- **Składnia**
  - słowa zastrzeżone (według ECMA Script 3)

```
abstract    enum        int         short
boolean     export      interface   static
byte        extends     long        super
char        final       native      synchronized
class       float       package     throws
const       goto        private     transient
debugger    implements  protected   volatile
double      import      public
```

30 „zablokowanych” identyfikatorów (sic!)

JavaScript – podstawy

- **Składnia**
  - identyfikatory
    - pierwszy znak:
      `litera | _ | $`
    - dalej: ciąg znaków ze zbioru:
      `litery + cyfry + { '_', '$' }`
  - średniki
    - rozdzielają instrukcje
    - **ale** …

Dobra Rada nr 1

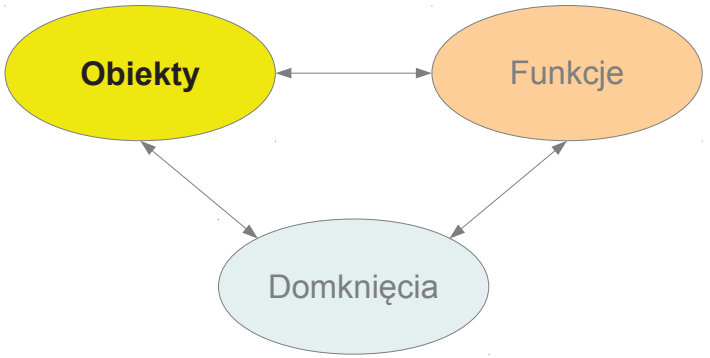
Średniki traktujemy jako **obowiązkowe**:

```
ins_1;
ins_2;
  …
ins_n;
```

JavaScript – podstawy

- **Wartości i typy**
  - wartości proste
    - liczby, napisy, wartości logiczne, `null`, `undefined`
  - wartości referencyjne
    - wszystkie pozostałe
  - typy proste
    - `Number`, `String`, `Boolean`, `Null`, `Undefined`
  - typy referencyjne
    - `Object`, …

Najważniejsze mechanizmy



JavaScript – obiekty

Obiekt

dynamiczna kolekcja właściwości

JavaScript – obiekty

- **Literały obiektowe**

```
1 var empty = {};
2
3 var person = {
4     "first-name": 'Jan',
5     "last-name": 'Kowalski'
6 };
```

  - ciąg par „`nazwa:wartość`”
  - jeśli nazwa jest „legalną” nazwą JavaScript i nie jest słowem zastrzeżonym (kluczowym), to nie musi być ujmowana w cudzysłów

JavaScript – obiekty

- **Literały obiektowe**

```
var flight = {
    airline: 'LOT',
    number: 'L2305',
    departure: {
        IATA: 'WAW',
        time: '2013-04-22 14:55',
        city: 'Warsaw'
    },
    arrival: {
        IATA: 'LAX',
        time: '2013-04-22 23:05',
        city: 'Los Angeles'
    }
};
```

  - wartością może być w szczególności literał obiektowy

JavaScript – obiekty

- **Dostęp do wartości pól**
  - dwie metody dostępu

```
person["first-name"]     // "Jan"
flight.departure.IATA    // "WAW"
```

  - próba odczytu wartości nieistniejącego pola zwraca wartość **`undefined`**

```
person["middle-name"]    // undefined
flight.status            // undefined
person["FIRST-NAME"]     // undefined
```

  - operator **`||`** pozwala określić „wartość domyślną”

```
var middle = person["middle-name"] || "(none)";
var status = flight.status || "unknown";
```


JavaScript – obiekty

- **Dostęp do wartości pól**
  - próba odczytu wartości składowej niezdefiniowanego obiektu rzuca wyjątek **`TypeError`**
  - operator **`&&`** pozwala się „zabezpieczyć”

```
flight.status                          // undefined
flight.status.delay                    // "TypeError"
flight.status && flight.status.delay // undefined
```


JavaScript – obiekty

- **Modyfikacja wartości atrybutów**
  - wartość „pola” modyfikujemy przez podstawienie

```
person['first-name'] = 'Aleksander';
```

  - próba modyfikacji wartości nieistniejącego pola powoduje dodanie go do obiektu!

```
person['middle-name'] = 'Maria';
person.nickname = 'Kowal';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```

JavaScript – obiekty

- **Referencje**
  - służą do manipulowania obiektami

```
var x = person;
x.nickname = 'Kowal';
var nick = person.nickname;
    // wartość 'nick' równa jest 'Kowal' bo
    // x i person są referencjami do tego
    // samego obiektu

var x = {}, b = {}, c = {};
    // każda z referencji wskazuje na inny
    // „pusty obiekt”

a = b = c = {};
    // teraz a, b oraz c wskazują na ten
    // sam pusty obiekt
```

  - z… …ję (j…

JavaScript – obiekty

- **Referencje**
  - zmiana wartości referencji

```
// Niech items będzie tablicą napisów (obiekt)
var items = ["raz", "dwa", "trzy"];

// Niech itemsRef będzie referencją do tej samej tablicy
var itemsRef = items;

// Zmieniamy wartość items
items = ["nowa", "tablica"];

if (items !== itemsRef) {
    console.log("Dwie różne");
} else {
    console.log("Jedna, ale zmieniona");
}
```

  - co wypisze powyższy program?

„Dwie różne”

JavaScript – obiekty

- **Referencje**
  - zmiana wartości referencji – inny przykład

```
// Niech item będzie napisem
var item = "test";

// Niech itemRef wskazuje na ten sam napis
var itemRef = item;

// Konkatenujemy wartość item z napisem „ing”
item += "ing";

if (item !== itemRef) {
    console.log("Dwa różne");
} else {
    console.log("Jeden, ale zmieniony");
}
```

  - jaki będzie efekt i **dlaczego**?

„Dwie różne”

Dobra Rada nr 2

Chcąc sprawdzić **ró(w|ż)ność** wartości **x** i **y** piszemy:

x === y

x !== y

JavaScript – typowanie

- Operator **typeof**

```
// Sprawdzamy, czy num jest łańcuchem znaków
if (typeof num === 'string') {
    // zamieniami go na liczbę
    num = parseInt(num, 10);
}

// Sprawdzamy, czy str jest łańcuchem znaków
if (typeof str === 'string') {
    // „rozbijamy" go do tablicy
    str = str.split(',');
}
```

- **typeof undefined === 'undefined'**
- **typeof null === ?**

Dobra Rada nr 3

Wartość **undefined** wyrażenia **typeof x** powinna oznaczać, że element (zmienna) **x** nie istnieje

```
var x;
typeof x === 'undefined'
// y - „świeża" zmienna
typeof y === 'undefined'
```

Wniosek: zmienne należy inicjalizować (zmienne typów referencyjnych – wartością **null**)

JavaScript – typowanie

- Atrybut **constructor**

```
// Sprawdzamy, czy num jest łańcuchem znaków
if (num.constructor === String) {
    // zamieniami go na liczbę
    num = parseInt(num, 10);
}

// Sprawdzamy, czy str jest tablicą
if (str.constructor === Array) {
    // zamieniamy ją na łancuch
    str = str.join(',');
}
```

obecny w każdej wartości **x** typu *referencyjnego*

JavaScript – typowanie

- **typeof** a **constructor** – porównanie

wartość x	typeof x	x.constructor
{ naz : "wart" }	'object'	Object
["raz", "dwa"]	'object'	Array
function () {}	'function'	Function
"łańcuch"	'string'	String
123	'number'	Number
true	'boolean'	Boolean
new User()	'object'	User

Typy proste – obiektowo

- Wartości typów prostych zachowują się jak obiekty

```
var person = 'John';
console.log(person.length);
```

- ale nie całkiem …

```
person.surname = 'Smith';
console.log(person.surname); // undefined
```

Co się dzieje?

Typy proste – obiektowo

- JavaScript dokonuje małego *hocus-pocus*

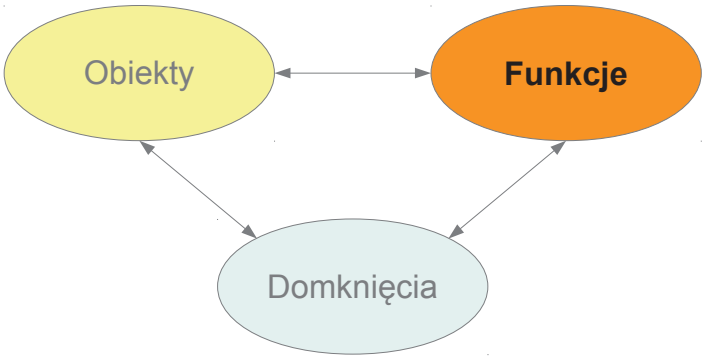
```
var person = 'John';
var tmp = new String(person);

~~console.log(person.length);~~

console.log(tmp.length);
tmp = null;
```

Mechanizm podobny do
„**auto-boxing**”
znanego np. z Javy

Najważniejsze mechanizmy c.d.



JavaScript – funkcje (wprowadzenie)

- Funkcje w JavaScript to:
  - „normalne wartości”
  - obiekty (sic!)
  - podstawowe narzędzie modularyzacji kodu
- Dwie metody definiowania
  - z pomocą wyrażeń funkcyjnych
  - używając instrukcji funkcyjnych

JavaScript – wyrażenia funkcyjne

- Składowe wyrażenia (literału) funkcyjnego
  - słowo kluczowe **`function`**
  - nazwa (opcjonalna) – jej brak oznacza, że literał wprowadza tzw. „funkcję anonimową”
  - ciąg parametrów ujęty w nawiasy „zwykłe”
  - ciąg instrukcji ujęty w nawiasy klamrowe
- Przykład

```
1 var add = function (a, b) {
2     return a + b;
3 };
```


JavaScript – instrukcje funkcyjne

- Składowe instrukcji funkcyjnej
  - słowo kluczowe **`function`**
  - nazwa (**obowiązkowa**)
  - ciąg parametrów ujęty w nawiasy „zwykłe”
  - ciąg instrukcji ujęty w nawiasy klamrowe

```
function add(x, y) {
  return x + y;
}
```

≅

```
var add = function add(x, y) {
  return x + y;
};
```


Dobra Rada nr 4

Funkcje **definiuj wyłącznie** jako wartości zmiennych, używając literałów funkcyjnych

```
var f = function () {
   ...
};
```

OK – bat **ŁAJ** ?!?!?!?

JavaScript – funkcje

- Przeciążanie nazw nie jest możliwe

```
1  var sendMessage = function (msg, obj) {
2      // jeśli podano komunikat i obiekt
3      if (arguments.length === 2) {
4          obj.handleMsg(msg);
5      } else {
6          console.log(msg);
7      }
8  };
9
10 sendMessage("Ahoj przygodo!");
11
12 sendMessage("Jak się masz?", {
13     handleMsg: function (msg) {
14         console.log("Wiadomość: " + msg);
15     }
16 });
```

„pseudo-tablica” **`arguments`** pozwala dotrzeć do argumentów

JavaScript – funkcje

- „Pseudo-tablica” **`arguments`** raz jeszcze

```
1  // Funkcja zwracająca tablicę wszystkich swoich argumentów
2  var makeArray = function () {
3      // Tablica pomocnicza
4      var arr = [], i;
5      // Przebiegamy listę argumentów
6      for (i = 0; i < arguments.length; i += 1) {
7          arr.push(arguments[i]);
8      }
9      // Zwracamy wynik
10     return arr;
11 };
```

- zmienna **`i`** „sterująca pętlą **for**” – jeden z nielicznych przypadków, kiedy brak inicjalizacji nie jest problemem

JavaScript – prototypy

- Każdy obiekt w JS
  - zawiera informację o swoim „prototypie”
  - dziedziczy wszystkie atrybuty prototypu
- Prototyp dowolnego literału obiektowego to
  - **`Object.prototype`**
- Tworząc nowy obiekt można wybrać jego prototyp
  - do wersji ES 3 włącznie – dosyć „pokrętne”
  - w ES 5 – metoda **`Object.create(…)`**

JavaScript – prototypy

- Metoda **`object`** (≈ **`Object.create`** z **ES5**)

```
1  var object = function (o) {
2      var F = function () {};
3      F.prototype = o;
4      return new F();
5  };
6
7  // Przyklad zastosowania
8  var another_person = object(person);
9
10 // Ukryta referencja do prototypu nie ma
11 // znaczenia podczas modyfikacji
12 another_person['first-name'] = 'Adam';
13 another_person['middle-name'] = 'Tomasz';
14 another_person.nickname = 'Kania';
15 console.log(person['first-name']); // Jan
16
17 // ale ma znaczenie "podczas odczytu"
18 person.profession = 'aktor';
19 console.log(another_person.profession); // aktor
```

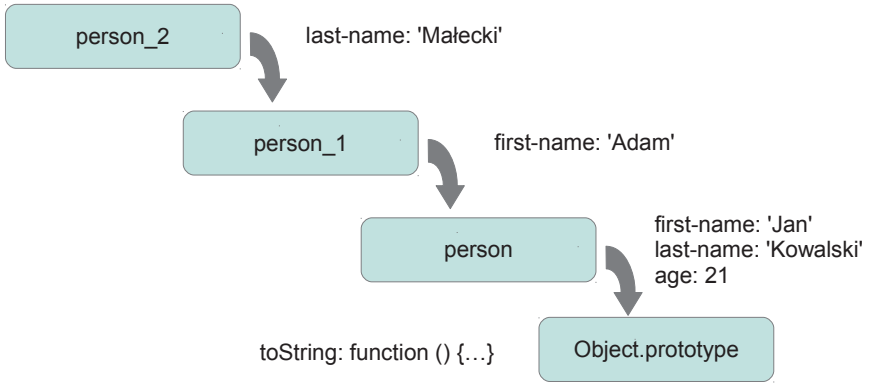

JavaScript – prototypy

- Oczywiście obiekt utworzony w oparciu o pewien prototyp może stać się prototypem dla kolejnego tworzonego obiektu …
- Tak powstają „łańcuchy prototypów”
  - każdy obiekt ma dokładnie jeden
  - jego długość to minimum 2

„Dziedziczenie prototypowe”

JavaScript – łańcuch prototypów

```
1 var person_1 = object(person);
2 person_1['first-name'] = 'Adam';
3 var person_2 = object(person_1);
4 person_2['last-name'] = 'Małecki';
```



Obiekty w JavaScript – „refleksja”

- Aby sprawdzić jakie atrybuty ma dany obiekt, można zbadać typy ich wartości:

```
1 typeof flight.number    // 'number'
2 typeof flight.status    // 'string'
3 typeof flight.arrival   // 'object'
4 typeof flight.manifest  // 'undefined' <- brak atrybutu
```

- **Uwaga!** Łańcuch prototypów jest brany pod uwagę!

```
1 typeof flight.toString    // 'function'
2 typeof flight.constructor // 'function'
```

- Można skorzystać z metody **`hasOwnProperty`**:

```
1 flight.hasOwnProperty('number')       // true
2 flight.hasOwnProperty('constructor')  // false
```


Obiekty w JavaScript – enumeracja

- **Problem**

  dotrzeć do wartości wszystkich pól obiektu

- **Rozwiązanie**

  wykorzystanie konstrukcji **`for`** … **`in`**

```
1 var attr;
2 for (attr in another_person) {
3     if (typeof another_person[attr] !== 'function') {
4         console.log(attr + ': ' + another_person[attr]);
5     }
6 }
```

- **Uwaga!**

  kolejność wyliczania jest „nieokreślona”

Obiekty w JavaScript – enumeracja

- **Przykład niepoprawnego użycia**

```
1 var t = [], i;
2 t[3]='c'; t[1]='x'; t[2]='b';
3 for (i in tab) {
4     console.log(tab[i]);
5 }
```

np. w IE <= 8.0 wynikiem będzie:

```
c
x
b
```

Obiekty w JavaScript – enumeracja

- Jeśli interesuje nas konkretna lista atrybutów

```
1  var i;
2  var props = [
3          'first-name',
4          'middle-name',
5          'last-name'
6      ];
7
8  for (i = 0; i < props.length; i += 1) {
9      console.log(props[i] + ': ' + another_person[props[i]]);
10 }
```


JavaScript – obiekty

- Operator **`delete`** – usuwanie atrybutów

```
1 console.log(another_person.nickname); // Kania
2
3 // Usuwamy atrybut 'nickname' z another_person
4 delete another_person.nickname;
5
6 // Teraz odwołania do wartości 'nickname' będą
7 // „przekierowywane” do prototypu - tzn. obiektu person
8 console.log(another_person.nickname); // Maria
```

- dynamiczne obiekty:

  „*pole* **=** *wartość*” **+** „**`delete`** *pole*”

JavaScript – zakres nazw

```
1  var fun_A = function () {
2      var a = 1,
3          fun_B = function () {
4              var b = 2;
5              console.log(a, b);
6          };
7      console.log(a, b);
8  };
9  fun_A();
```

`ReferenceError`: b is not defined

JavaScript – zakres nazw

```
1  var info = 'Bueee';
2  var fun = function () {
3      console.log(info);
4      var info = 'Ahoj!';
5      console.log(info);
6  };
7
8  fun();
```

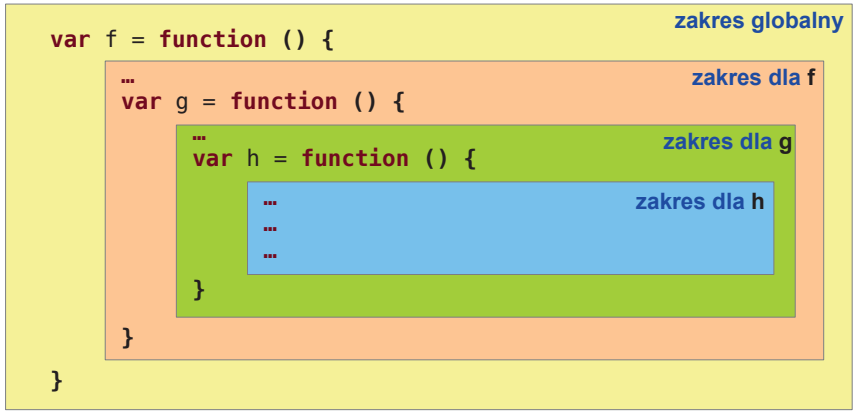
Wynik: **`undefined`**
`Ahoj!`

JavaScript – zakres nazw

```
1 var info = 'Bueee';
2 var fun = function () {
3     var info;
4     console.log(info);
5     info = true;
6     console.log(info);
7 };
8 fun();
```

Wynik: **undefined**
true

Zagnieżdżanie zakresów nazw



JavaScript – zakres nazw

- **Zakres**
  - wprowadzany przez funkcje
  - … i **tylko** funkcje (*„zakres bloku”* nie istnieje)
  - „wyciąganie” deklaracji zmiennych:

```
if (b) {                    var x;
    var x = true;   ==>     if (b) {
} …                             x = true;
                            } …
```

  - zmienna zadeklarowana wewnątrz funkcji *nie jest widoczna* na zewnątrz (lokalność)

Funkcje – zakres „leksykalny”

- Zakres określany w momencie definiowania:

```
1 var f1 = function () { var a = 1; f2(); },
2     f2 = function () { return a; };
3 f1();
```

ReferenceError: **a** is not defined

- **f2** ma dostęp tylko do zakresu „globalnego”

Funkcje – zakres „leksykalny"

```
1 var f1 = function () { var a = 1; f2(); },
2     f2 = function () {
3         console.log(a || 'nic' );
4     };
5 f1();         →       'nic'
6 var a = 15;
7 f1();         →       15
8 a = 100;
9 f1();         →       100
```


Kontekst – czym jest „this" ?

```
1 var o = {
2     x: 10,
3     m: function () {
4         var x = 1;
5         console.log(x, this.x);
6     }
7 };
8
9 o.m();
```

Wynik: **1, 10**

JavaScript – kontekst

```
1  var o = {
2      x: 10,
3      m: function () {
4          var x = 1,
5              f = function () {
6                  console.log(x, this.x);
7              };
8          f();
9      }
10 };
11
12 o.m();
```

Wynik: **1, undefined**

Funkcje – wywołanie

- **Anatomia wywołania**
  - argumenty
    - umieszczane w „pseudo-tablicy" **arguments**
    - przypisywane ewentualnym parametrom
    - liczba parametrów > **arguments.length**
      - „nadmiarowe" parametry → wartość **undefined**
  - referencja **this**
    - ustala „Obiekt Zmiennych"

Funkcje – wywołanie

- **Wzorce wywołania**
  - ustalają „charakter użycia” funkcji
  - cztery możliwości
    - metoda
    - „zwykła” funkcja (metoda globalna)
    - konstruktor
    - wywołanie pośrednie (`call`, `apply`)
  - główne pytanie:
    - na co wskazuje `this` ??

Technologie sieci Web – język JavaScript

Funkcje – wywołanie

- **Wzorce wywołania**
  - **metoda**
    - funkcja będąca wartością atrybutu obiektu
    - `this` wskazuje na ten obiekt
  - „zwykła funkcja”
    - funkcja nie będąca metodą
    - `this` wskazuje na „obiekt globalny” (zawsze!)
    - jest to również prawdą dla funkcji „zagnieżdżonych” …

Technologie sieci Web – język JavaScript

Funkcje – wywołanie

- **Przykład**: prosty obiekt z metodą

```
var myObject = {
    val: 0,
    increment: function (i) {
        this.val +=
            typeof i === 'number' ? i : 1;
    }
};
myObject.increment();
console.log(myObject.val);    →  1

myObject.increment(2);
console.log(myObject.val);    →  3
```

Technologie sieci Web – język JavaScript

Funkcje – wywołanie

- Dodajemy metodę „`double`”

```
myObject.double = function () {
    var helper = function () {
            this.val = this.val * 2;
        };
    helper();
};
myObject.double();
console.log(myObject.val);    →  3
```

Funkcje – wywołanie

- Na ratunek – „popularna sztuczka”

```
myObject.double = function () {
    var that = this,
        helper = function () {
            that.val = that.val * 2;
        };
    helper();
};
myObject.double();
console.log(myObject.val);    →   6
```


Funkcje – wywołanie

- **Wzorce wywołania c.d.**
  - **konstruktor**
    - funkcja **`F`** wywoływana z użyciem **`new`**
    - tworzony jest nowy obiekt
    - prototyp – wartość **`F.prototype`**
    - atrybut **`prototype`** mają wszystkie funkcje
    - **`this`** wskazuje na tworzony obiekt
  - **uwaga**:
    - wywołanie „konstruktora” bez **`new`** (czyli jako „zwykłej” funkcji) ma dosyć niespodziewane konsekwencje…

Funkcje – konstruktory

```
var Student = function (name, id) {
    this.name = name;
    this.id = id;
    return this;  // opcjonalne
};

Student.prototype.show = function () {
    return this.name + ' (' + this.id + ')';
};

var jan_kowalski =
    new Student('Jan Kowalski', '987654321');

console.log(jan_kowalski.show());
```


Funkcje – wywołanie

- **Wzorce wywołania c.d.**
  - z użyciem: **`call(ctx, ciąg_arg)`**
    - **`ctx`** – kontekst (dostarcza wartość **`this`**)

```
var o1 = { u: 40, x: 123 };
var o2 = { v: 10, y: 'abc' };
var f = function (name, mult) {
    this[name] *= mult;
};
console.log('(' + o1.u + ',' + o2.v + ')');
f.call(o1, 'u', 2);
f.call(o2, 'v', 3);
console.log('(' + o1.u + ',' + o2.v + ')');
```

Funkcje – wywołanie

- **Wzorce wywołania c.d.**
  - z użyciem: **`apply(ctx, tablica_arg)`**
    - **`ctx`** – kontekst (dostarcza wartość **`this`**)

```
var o1 = { u: 40, x: 123 };
var o2 = { v: 10, y: 'abc' };
var f = function (name, mult) {
    this[name] *= mult;
};
console.log('(' + o1.u + ',' + o2.v + ')');
f.apply(o1, ['u', 2]);
f.apply(o2, ['v', 3]);
console.log('(' + o1.u + ',' + o2.v + ')');
```

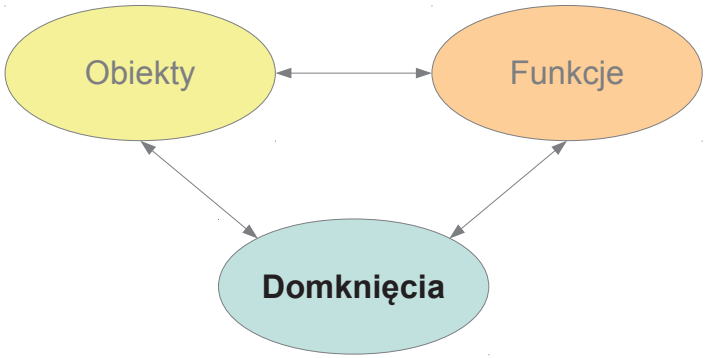

Problem z „globalnością”

- JavaScript niestety „zachęca” do posługiwania się zmiennymi globalnymi …
- Program wykonuje się w środowisku kodu potencjalnie pochodzącego z różnych źródeł
- Jak radzić sobie z tym problemem?
  - globalny obiekt aplikacji
  - mechanizm domknięć (ang. *closures*)
  - moduły – CommonJS, …, Harmony/**ES6**

Globalny obiekt aplikacji

```
1  var myApp = {};
2
3  myApp.person = {
4      "first-name": 'Jan',
5      "last-name": "Kowalski"
6  };
7
8  myApp.flight = {
9      airline: 'LOT',
10     number: 'L2305',
11     departure: {
12         IATA: 'WAW',
13         time: '2013-04-22 14:55',
14         city: 'Warsaw'
15     },
16     arrival: {
17         IATA: 'LAX',
18         time: '2013-04-22 23:05',
19         city: 'Los Angeles'
20     }
21 };
```


Najważniejsze mechanizmy c.d.



Domknięcie

zakres powstający w momencie deklarowania funkcji, pozwalający jej na dostęp i manipulowanie *wartościami zewnętrznymi*

Domknięcie

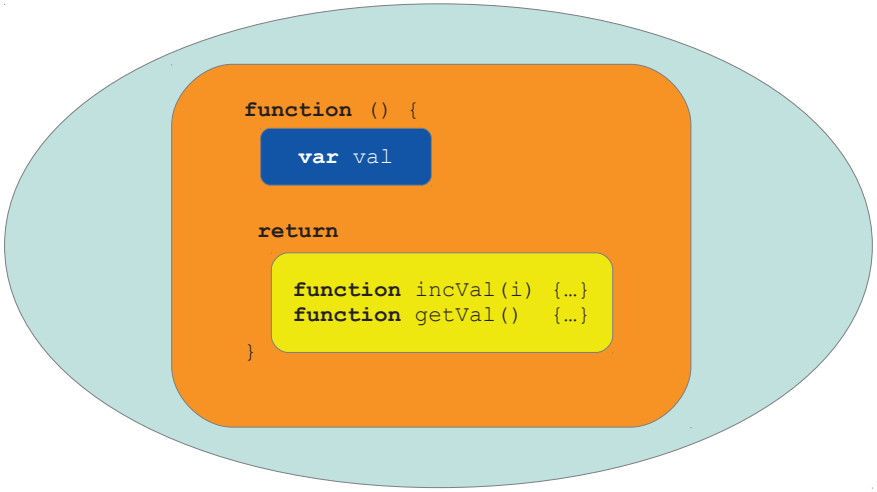
daje funkcji dostęp do wszystkich zmiennych (oraz funkcji), które były widoczne w momencie jej deklarowania

Domknięcia – przykład

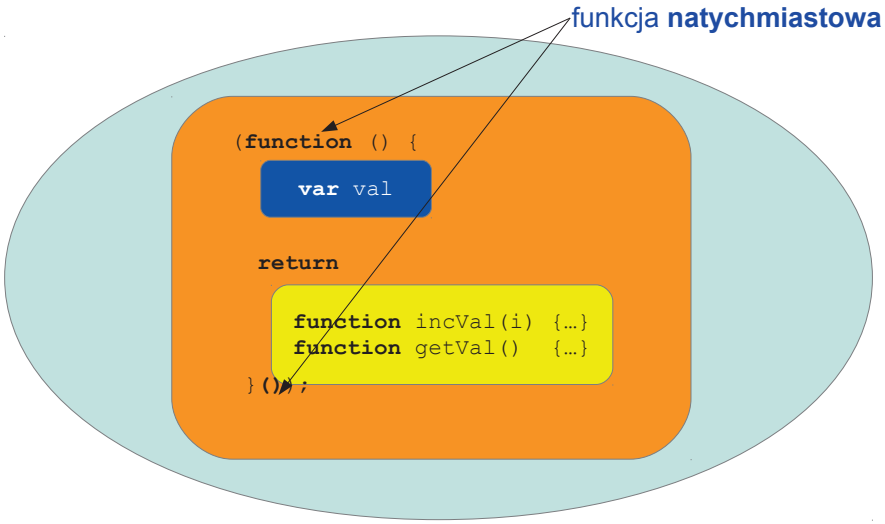
```
var myObject = (function () {
    var val = 0; ←   zmienna „prywatna” myObject
    return {
        incVal: function (i) { val += i; },
        getVal: function () { return val; }
    };
}());

myObject.inVal(5);
console.log(myObject.getVal);
```

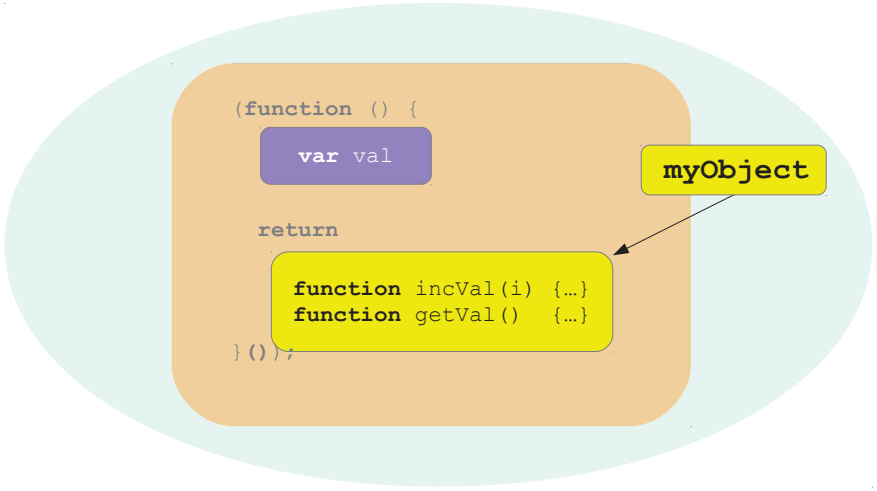

Domknięcia – przykład



Domknięcia – przykład



Domknięcia – przykład



Domknięcia

- Zasada działania / wykorzystania domknięć
  - funkcje zagnieżdżone w danej funkcji mają dostęp do wszystkich zmiennych w niej zdefiniowanych
  - co ważniejsze – nie zależy to od „czasu życia” funkcji zewnętrznej

domknięcia → sposób na modularyzację

Domknięcia – jeszcze jeden przykład

```
var keyMaker = (function () {
    var pref = '',
        seq = 0;
    return {
        setPref: function (p) { pref = p; },
        setSeq: function (s) { seq = s; },
        genKey: function () {
            seq += 1;
            return pref + seq;
        }
    };
}());
```

clientFun(keyMaker.genKey)

Wykorzystanie domknięć – „moduły”

- dzięki domknięciom możemy konstruować obiekty, które oferują jedynie pewien zestaw operacji/metod, ukrywając wewnętrzną reprezentację danych
  - „dane prywatne” modułu reprezentowane jako elementy domknięcia
  - przykład – `pref` i `seq` z poprzedniego slajdu
- ważniejsze podejścia do definiowania modułów
  - CommonJS Modules
  - AMD (Asynchronous Module Definition)
  - **ES6** / Harmony ← **przyszły standard** (?)

Moduły ES6/Harmony – definiowanie

```
module MyModule {
  // elementy eksportowane
  export let myobject = {};
  export function hello() {
    console.log("hello");
  };

  // elementy ukryte
  function goodbye() {
    // coś...
  }
}
```


Moduły ES6/Harmony – użycie

```
// importujemy tylko myobject
import myobject from MyModule;
console.log(myobject);

// importujemy wszystko
import * from MyModule;
console.log(myobject);
console.log(hello);

// jawnie wyliczmy elementy importowane
import {myobject, hello} from MyModule;
console.log(myobject);
console.log(hello);
```


ES6 / Harmony to (póki co) *„plany na przyszłość JS”*

```
node --v8-options | grep harmony
```