

Technologie internetowe

Wiesław Pawłowski

Trochę formalności

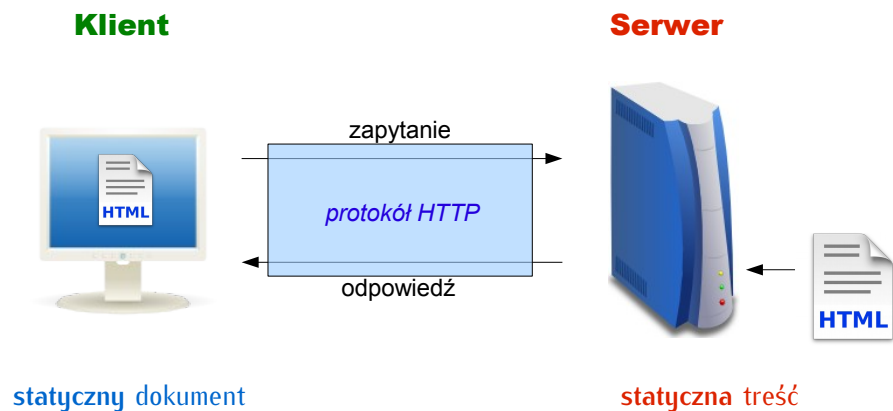
- **Udział** w wykładzie nieobowiązkowy (choć **zalecany**!)
- **Znajomość treści** omawianych na wykładzie podczas laboratoriów – **obowiązkowa**
- Slajdy z wykładu, informacje i materiały pomocnicze:

<http://szuflandia.pjwstk.edu.pl/~wpawlowski/TIN/>

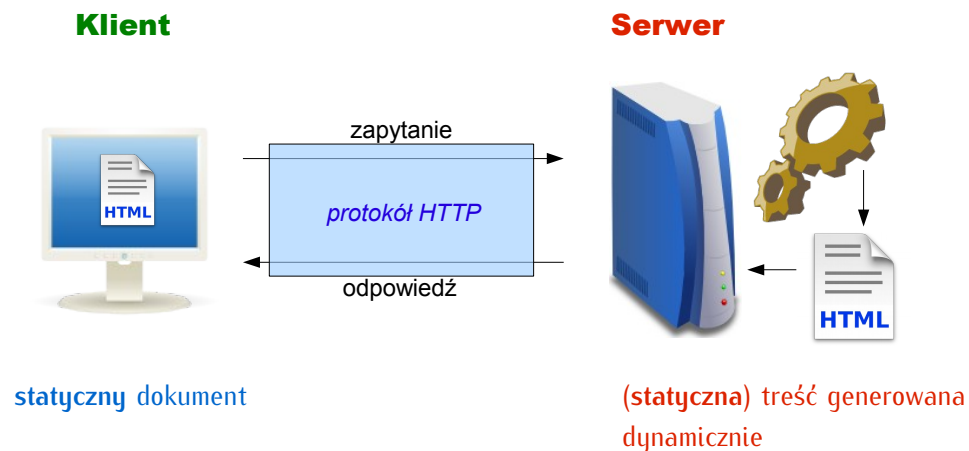
użytkownik: **tin**

hasło: **pj2@16**

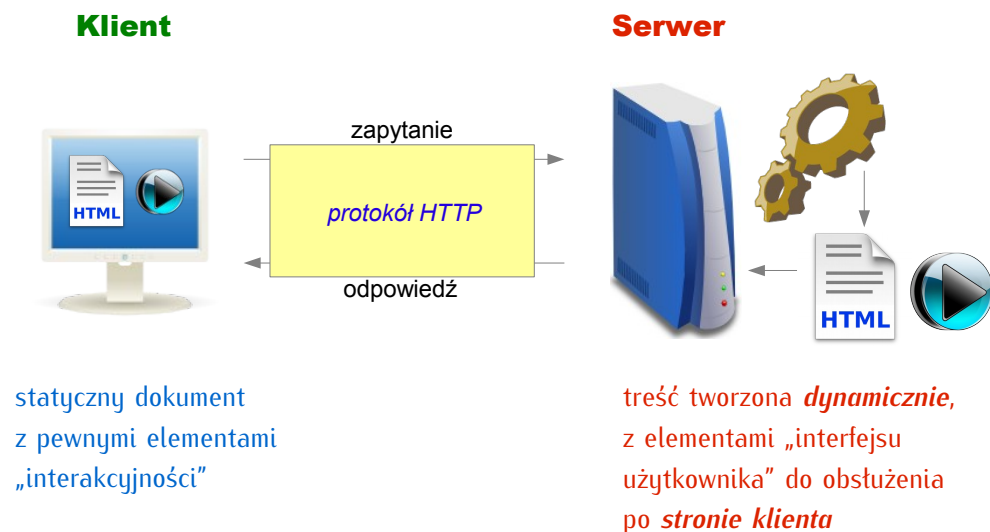
Od statycznych dokumentów do aplikacji



Od statycznych dokumentów do aplikacji

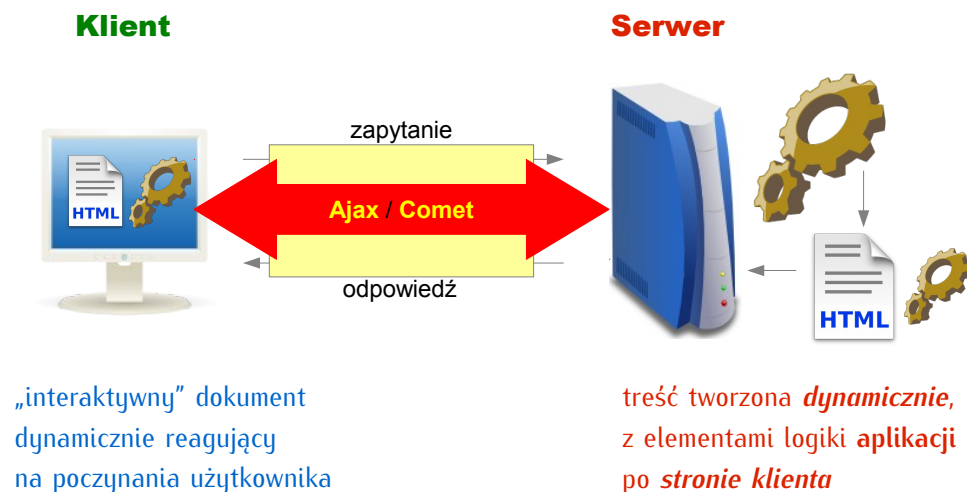


Od statycznych dokumentów do aplikacji



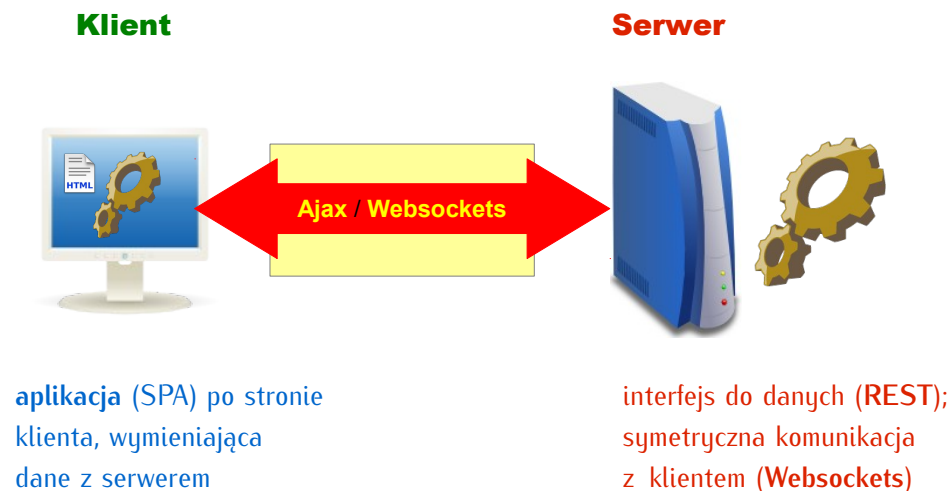
5

Od statycznych dokumentów do aplikacji



6

Od statycznych dokumentów do aplikacji



7

O czym (między innymi) będziemy mówić

- Metody prezentacji treści oraz tworzenia interfejsu użytkownika aplikacji/serwisów internetowych
 - ▷ języki (X)HTML(5) i CSS
- Metody i metodologie tworzenia logiki aplikacji
 - ▷ po stronie serwera (Node.js, Express, ...)
 - ▷ po stronie klienta (JavaScript)
 - ▷ w trybie mieszanym (Ajax, Comet/Websockets)
- Protokół HTTP
 - ▷ architektura, klienci, serwery
 - ▷ REST

8

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)
 - ▷ niebawem: edycja 6 – wiele nowości
- 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

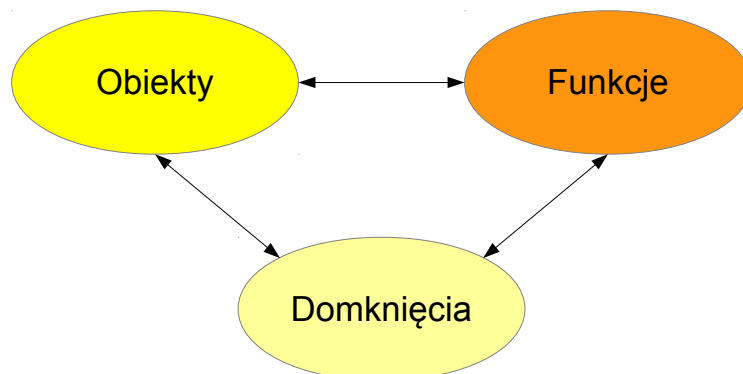
- **Firefox (JIT)**
 - ▷ SpiderMonkey → TraceMonkey → JägerMonkey
- **Chrome**
 - ▷ V8 (JIT)
- **Internet Explorer**
 - ▷ 5.5-8.0 JScript; 9, 10, 11 Chakra (JIT)
- **Opera**
 - ▷ ≤ 9.0 Linear(A/B); 9.5-10 Futhark; 10.5+ Carakan (JIT)
- **Safari JavaScriptCore, SquirrelFish (JIT)**

Obsługa ECMAScript 5 w przeglądarkach

	THIS BROWSER	IE 8	IE 9	IE 10, 11	FF 3.5, 3.6	FF 4+	SF 5	SF 5.1	SF 6	WebKit	CH 7-12	CH 13-16	CH 19+	OP 10.50-11.50	OP 12	OP 12.10, 15	Konq 4.3	Konq 4.9	BESEN	Rhino 1.7
Object.create	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.defineProperty	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.defineProperties	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.getPrototypeOf	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes
Object.keys	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes
Object.seal	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.freeze	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.preventExtensions	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.isFrozen	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.isExtensible	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.getOwnPropertyDescriptor	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
Object.getOwnPropertyNames	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes
Date.prototype.toJSON	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Date.now	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.isArray	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
JSON	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Function.prototype.bind	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes
String.prototype.trim	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Array.prototype.indexOf	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.lastIndexOf	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.every	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.some	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.forEach	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.map	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.filter	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.reduce	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.reduceRight	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Getter in property initializer	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Setter in property initializer	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Property access on strings	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Reserved words as property names	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	No
Zero-width chars in identifiers	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No	No	Yes	No	No	Yes	No	No	No	Yes	Yes
Immutable undefined	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	Yes	No	No	Yes	Yes	No	Yes	Yes	Yes
Strict mode	Yes	No	No	Yes	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No	Yes	No

<http://kangax.github.io/compat-table/es5/>

Najważniejsze mechanizmy



JavaScript – podstawy

- **Składnia**
 - ▷ wykorzystywany zestaw znaków: **Unicode**
 - ▷ nazwy – **małe** i **wielkie** litery są **rozróżniane**
 - ▷ komentarze
 - `// do końca linii to komentarz`
 - `/* wszystko „pomiedzy” 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:

litera + **cyfry** + { **'_'**, **'\$'** }

- ▷ średniki

- rozdzielają instrukcje

- **ale** ...

Dobra Rada nr 1

Średniki traktujemy jako **obowiązkowe**:

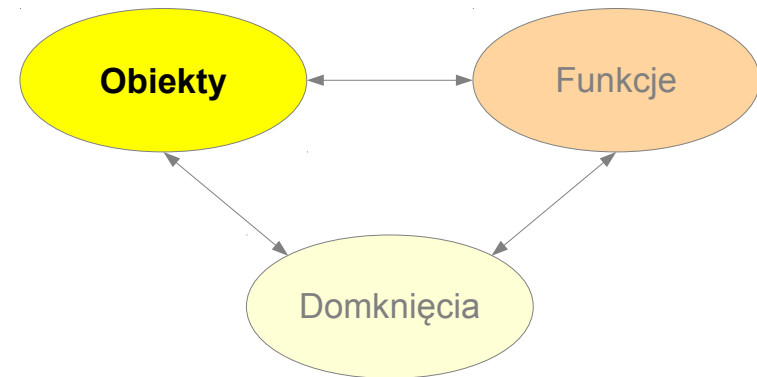
```
ins_1;
ins_2;
...
ins_n;
```

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

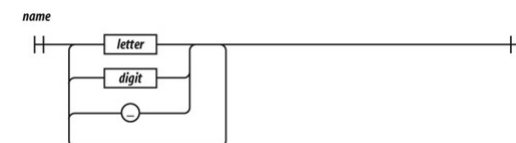
JavaScript – obiekty

• 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**



Dobra Rada nr 2

Zawsze używaj **literałów obiektowych**:

`var obj = {};` 

`var obj = new Object();` 

JavaScript – obiekty

- Literały obiektowe**

```
1 var flight = {
2   airline: 'LOT',
3   number: 'L2305',
4   departure: {
5     IATA: 'WAW',
6     time: '2014-04-22 14:55',
7     city: 'Warsaw'
8   },
9   arrival: {
10    IATA: 'LAX',
11    time: '2014-04-22 23:05',
12    city: 'Los Angeles'
13  }
14 };
```

- ▷ wartością może być w szczególności **literał obiektowy**

JavaScript – obiekty

- Dostęp do wartości pól**

- ▷ dwie metody dostępu

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

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

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

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

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

JavaScript – obiekty

- Dostęp do wartości pól**

- ▷ próba odczytu wartości składowej **niezdefiniowanego obiektu** rzuca wyjątek **TypeError**
- ▷ operator **&&** pozwala się „zabezpieczyć”

```
1. flight.status          // undefined
2. flight.status.delay    // "TypeError"
3. flight.status && flight.status.delay // undefined
```

JavaScript – obiekty

• Modyfikacja wartości atrybutów

- ▷ wartość „pola” modyfikujemy przez **podstawienie**

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

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

```
1. person['middle-name'] = 'Maria';
2. person.nickname = 'Kowal';
3. flight.equipment = {
4.   model: 'Boeing 777'
5. };
6. flight.status = 'overdue';
```

JavaScript – obiekty

• Referencje

- ▷ służą do manipulowania obiektami

```
1 var x = person;
2 x.nickname = 'Kowal';
3 var nick = person.nickname;
4   // wartość 'nick' równa jest 'Kowal' bo
5   // x i person są referencjami do tego
6   // samego obiektu
7
8 var x = {}, b = {}, c = {};
9   // każda z referencji wskazuje na inny
10  // „pusty obiekt”
11
12 a = b = c = {};
13   // teraz a, b oraz c wskazują na ten
14   // sam pusty obiekt
```

- ▷ zawsze pokazują na **docelowy obiekt** – nigdy na inną referencję (jak np. w języku C)

JavaScript – obiekty

• Referencje

- ▷ zmiana wartości referencji

```
1 // Niech items będzie tablicą napisów (obiekt)
2 var items = ["raz", "dwa", "trzy"];
3
4 // Niech itemsRef będzie referencją do tej samej tablicy
5 var itemsRef = items;
6
7 // Zmieniamy wartość items
8 items = ["nowa", "tablica"];
9
10 if (items !== itemsRef) {
11   console.log("Dwie różne");
12 } else {
13   console.log("Jedna, ale zmieniona");
14 }
```

- ▷ co wypisze powyższy program?

„Dwie różne”

JavaScript – obiekty

• Referencje

- ▷ zmiana wartości referencji – inny przykład

```
1 // Niech item będzie napisem
2 var item = "test";
3
4 // Niech itemRef wskazuje na ten sam napis
5 var itemRef = item;
6
7 // Konkatenujemy wartość item z napisem „ing”
8 item += "ing";
9
10 if (item !== itemRef) {
11   console.log("Dwa różne");
12 } else {
13   console.log("Jeden, ale zmieniony");
14 }
```

- ▷ jaki będzie efekt i dlaczego?

„Dwie różne”

Dobra Rada nr 3

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

x === y

x !== y

Dobra Rada nr 4

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

- Operator **typeof**

```
1 // Sprawdzamy, czy num jest łańcuchem znaków
2 if (typeof num === 'string') {
3   // zamieniamy go na liczbę
4   num = parseInt(num, 10);
5 }
6
7 // Sprawdzamy, czy str jest łańcuchem znaków
8 if (typeof str === 'string') {
9   // „rozbijamy” go do tablicy
10  str = str.split(',');
11 }
```

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

JavaScript – typowanie

- Atrybut **constructor**

```
1 // Sprawdzamy, czy num jest łańcuchem znaków
2 if (num.constructor === String) {
3   // zamieniamy go na liczbę
4   num = parseInt(num, 10);
5 }
6
7 // Sprawdzamy, czy str jest tablicą
8 if (str.constructor === Array) {
9   // zamieniamy ją na łańcuch
10  str = str.join(',');
11 }
```

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*

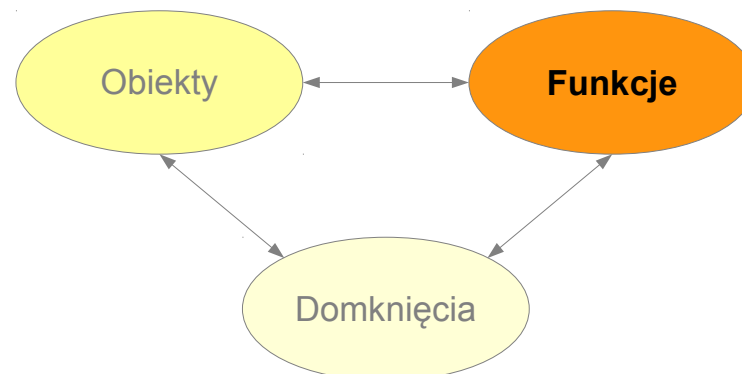
```
var person = 'John';
var tmp = new String(person);
```

```
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;
}
```

 $\cong$

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

Dobra Rada nr 5

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

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

OK – bat **ŁAJ** !?!?!?

Dobra Rada nr 5

stosuj **1TBS** (One True Brace Style)

```
var f = function (a, b) {
  if (...) {
    a();
  } else {
    b();
  }
};
```

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 // Przykład 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”

Object.create – ES5

- `Object.create(proto[, propertiesObj])`
 - `proto` – prototyp (istniejący obiekt)
 - `propertiesObj` – obiekt opisujący dodawane do prototypu atrybuty i ich właściwości

```

{
  p: { configurable: false, value: 1},
  q: {
    get function () { return 10; },
    set function (val) {
      console.log('ustawiam `q` na ', val);
    }
  }
}

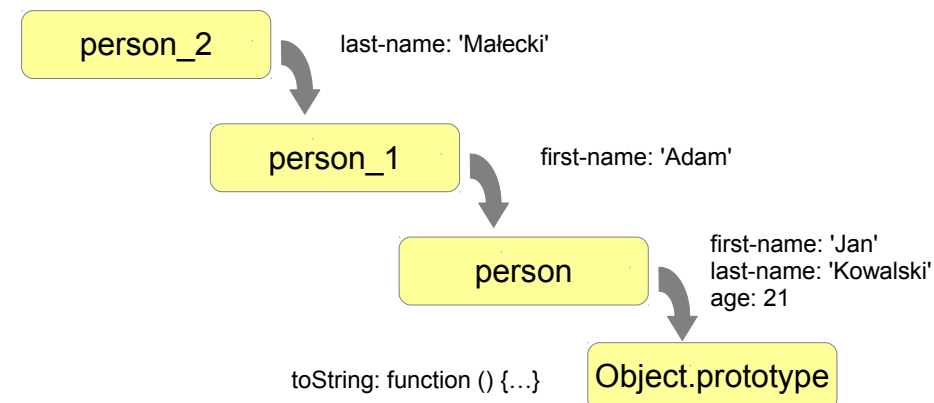
```

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 **z badać 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();
```

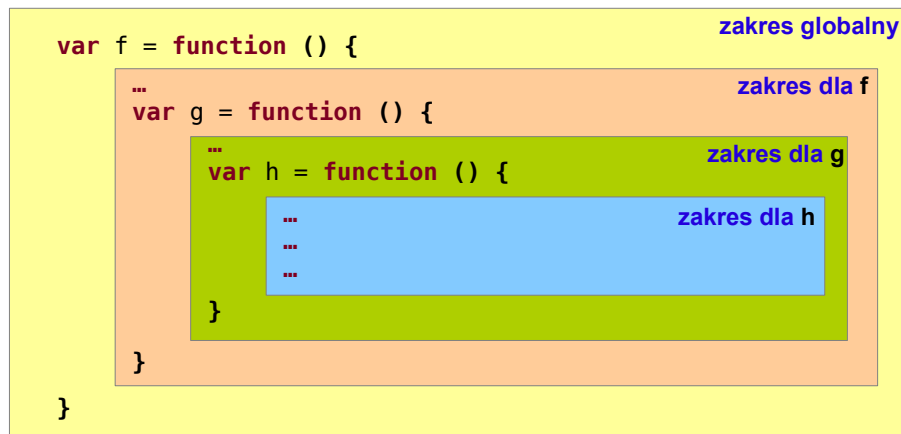
Wynik: **undefined**
Ahoj!

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 = true;
} ...
  
```

==>

```

var x;
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` ??

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” ...

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
```

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**
 - **prototype** – 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**

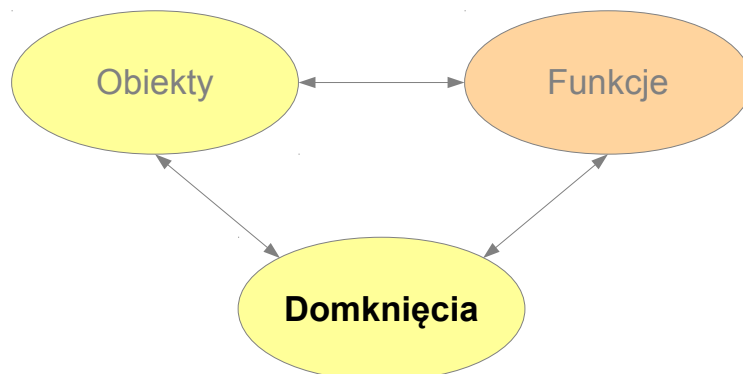
Globalny obiekt aplikacji

```

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

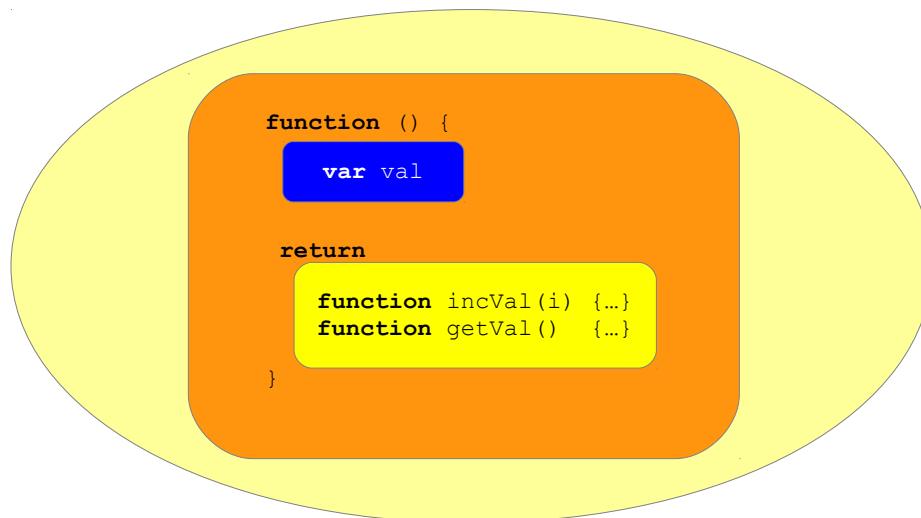
daje funkcji dostęp do wszystkich zmiennych (oraz funkcji), które były widoczne w momencie jej deklarowania

Domknięcia – przykład

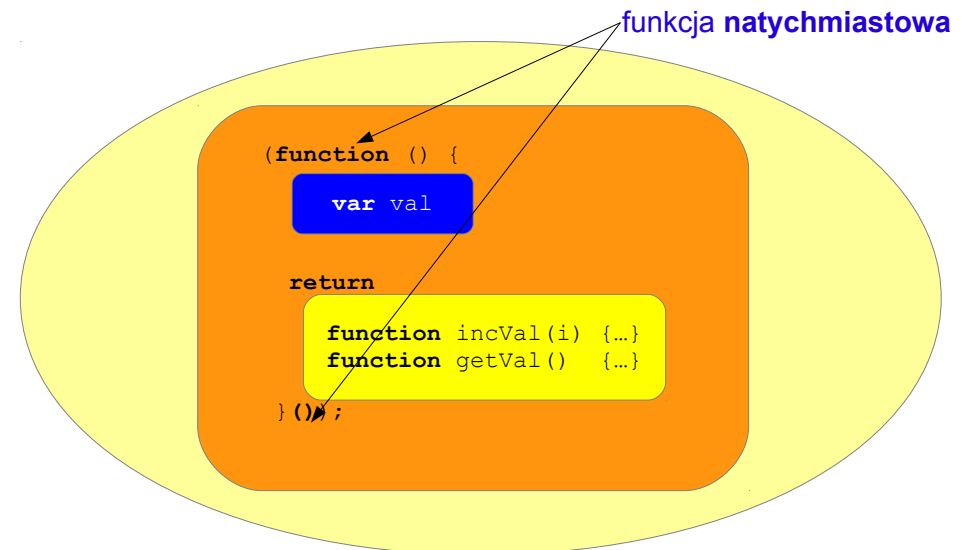
```
var myObject = (function () {
    var val = 0; ← zmienna „prywatna” myObject
    return {
        incVal: function (i) { val += i; },
        getVal: function () { return val; }
    };
})();

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

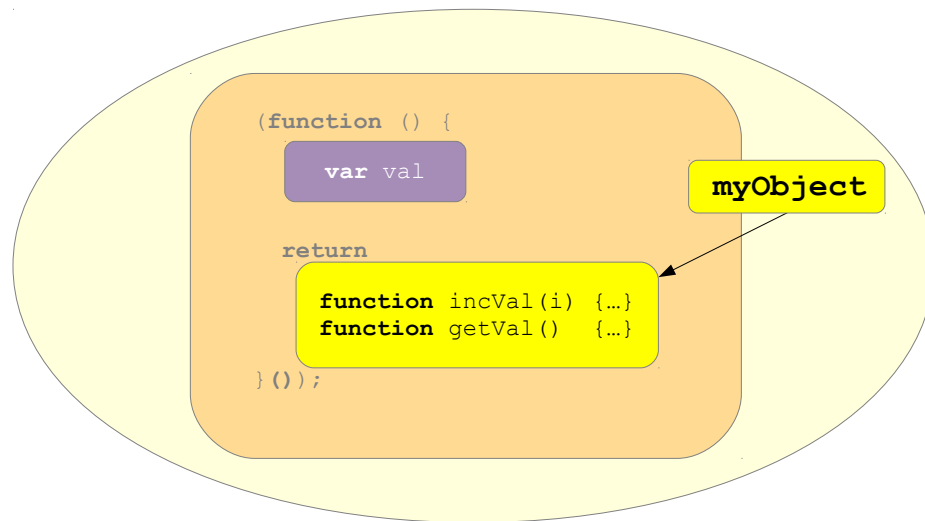
Domknięcia – przykład



Domknięcia – przykład



Domknięcia – przykład



Domknięcia

- Zasada działania / wykorzystania
 - ▷ 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 CommonJS

- Najważniejsza implementacja – **Node.js** (zawiera pewne rozszerzenia w stosunku do specyfikacji CommonJS)
 - ▷ prosta i zwarta składnia
 - ▷ moduły zaprojektowane z myślą o synchronicznym ładowaniu
 - ▷ głównie stosowane po stronie serwera

Do modułów CommonJS wrócimy omawiając Node.js

Moduły AMD

- Główna implementacja – **RequireJS**
 - ▷ nieco bardziej skomplikowana składnia niż w przypadku CommonJS
 - ▷ moduły zaprojektowane z myślą o asynchronicznym wykorzystaniu

AMD = Asynchronous Module Definition
 - ▷ główne zastosowanie – po stronie przeglądarki

Moduły ES6

- Połączenie koncepcji CommonJS i AMD
 - ▷ prosta i zwarta składnia, podobna do CommonJS
 - ▷ wsparcie zarówno dla ładowania synchronicznego, jak i asynchronicznego
 - ▷ API umożliwiające konfigurowanie mechanizmu ładowania – np. ładowanie warunkowe

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);

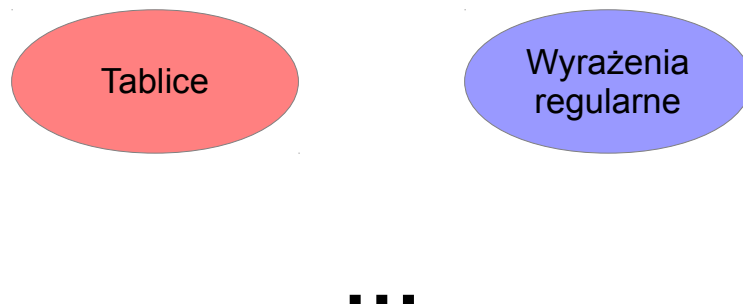
// importujemy wszystko
import * from MyModule;
console.log(myobject);
console.log(hello);

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

ES6 / Harmony to „*plany na najbliższą przyszłość JS*”

`node --v8-options | grep harmony`

Inne mechanizmy



Tablice – dwa sposoby patrzenia

- tablice jako **stosy**

```
var cs = ['red', 'blue'];
cs.push('brown');
cs[3] = 'black';
console.log(cs.length); // 4
var c = cs.pop();
console.log(c);          // 'black'
```

- tablice jako **kolejki**

```
var cs = ['red'];
cs.unshift('blue', 'green');
console.log(cs.length); // 3
var c = cs.shift();
console.log(c);          // 'red'
console.log(cs.length); // 2
```

Tablice – zmiana kolejności elementów

```
var tab = [0, 1, 5, 10, 15];
tab.reverse();
console.log(tab); // [15, 10, 5, 1, 0]
tab.sort();
console.log(tab); // [0, 1, 10, 15, 5]
var compare = function (a, b) {
  if (a < b) {
    return -1;
  } else if (a > b) {
    return 1;
  } else {
    return 0;
  }
};
tab.sort(compare);
console.log(tab); // [0, 1, 5, 10, 15]
```

Tablice – ciekawsze z pozostałych metod

```
var tab = [15, 10, 5, 1, 0];
var t1 = tab.filter(function (a) { return a > 5; });
console.log(tab); // [15, 10, 5, 1, 0]
console.log(t1); // [15, 10]
var t2 = tab.map(function (a) { return a * 2; });
console.log(t2); // [30, 20, 10, 2, 0]

var t3 = tab.reduce(function (prev, curr) {
  return prev + curr;
});
console.log(t3); // 31

var tab2 = ['s', 'i', 'v', 'l'];
var t4 = tab2.reduceRight(function (prev, curr) {
  return prev + curr;
}, 'E');
console.log(t4); // Elvis
```

Wyrażenia regularne

```
var txt = "cat, bat, sat, fat";
var pat = /.at/;
var mat = pat.exec(txt)

console.log(mat.index); // 0
console.log(mat[0]); // cat
console.log(pat.lastIndex); // 0
mat = pat.exec(txt)

// Jaki będzie wynik?
console.log(mat.index); // 0
console.log(mat[0]); // cat
console.log(pat.lastIndex); // 0
```

Wyrażenia regularne

```
var txt = "cat, bat, sat, fat";
var pat = /.at/g;
var mat = pat.exec(txt)

console.log(mat.index); // 0
console.log(mat[0]); // cat
console.log(pat.lastIndex); // 3
mat = pat.exec(txt)

// Jaki będzie wynik?
console.log(mat.index); // ?
console.log(mat[0]); // ?
console.log(pat.lastIndex); // ?
```

Wyrażenia regularne

```
var txt = "cat, bat, sat, fat";
var pat = /.at/g;
var mat = pat.exec(txt)

console.log(mat.index);    // 0
console.log(mat[0]);       // cat
console.log(pat.lastIndex); // 3
mat = pat.exec(txt)

// Jaki będzie wynik?
console.log(mat.index);    // 5
console.log(mat[0]);       // bat
console.log(pat.lastIndex); // 8
```

Wyrażenia regularne

```
var txt = "cat, bat, sat, fat";
var pat = /.at/g;
var dispMatch = function () {
  console.log("tekst: " + RegExp.input + "");
  console.log("dopasowanie: " + RegExp.lastMatch + "");
  console.log("tekst przed: " + RegExp.leftContext + "");
  console.log("tekst za: " + RegExp.rightContext + "");
  console.log("ostatnia grupa: " + RegExp.lastParen + "");
};

if (pat.test(txt)) {
  dispMatch();
}
// Jaki będzie wynik?

tekst:      'cat, bat, sat, fat'
dopasowanie: 'cat'
tekst przed: ''
tekst za:   ', bat, sat, fat'
ostatnia grupa: ''
```

A co to takiego
grupa
???

Wyrażenia regularne

```
var txt = "Dzisiaj mamy 2015-03-24 w „zapisie komputerowym.”;
var pat = /(\d{4})-(\d{2})-(\d{2})/;
var dispGroups = function () {
  var i;
  for(i = 1; i <= 9; i += 1) {
    console.log("$" + i + ": " + RegExp["$" + i] + "");
  }
};

if (txt.match(pat)) {
  dispGroups();
}
// Jaki będzie wynik?
```

```
$1: '2015'
$2: '03'
$3: '25'
$4: ''
...
$9: ''
```

Grupy: \$1 ... \$9

Małe ćwiczonko-zagadka :)

Wyrażenia regularne w JS

```
var template =  
'<table border="{border}">' +  
'  <tr><td>{first}</td><td>{last}</td></tr>' +  
'</table>';
```

```
var data = {  
  first: "Jan",  
  last:  "Kowalski",  
  border: "2"  
};
```

Jak to zdefiniować?

```
template.supplant(data)
```



```
<table border="2">  
  <tr><td>Jan</td><td>Kowalski</td></tr>  
</table>
```