Technologie Internetu. Programowanie obiektowe w JavaScript

Aleksander Denisiuk (denisjuk@pja.edu.pl)
Polsko-Japońska Akademia Technik Komputerowych
Wydział Informatyki w Gdańsku
ul. Brzegi 55, 80-045 Gdańsk

6 maja 2020

Najnowsza wersja tego dokumentu dostępna jest pod adresem
http://users.pja.edu.pl/~denisjuk/

Podstawy obiektów

Utworzenie obiektu

- ✔ Obiekt == Tablica assocjacyjna == Hash
- ✔ Przechowuje dowolne pary `klucz: vartość`
- ✔ Metoda obiekta — jest *funkcją*, dodaną do tablicy
- ✔ Utworzenie obiektu

```
let user = new Object();
let user = {};
```

- ✔ Utworzenie obiektu z właściwościami

```
let user = {
  name: "John",
  age: 30
}
```

  - ✘ dopuszczalny jest przecinek po ostatniej właściwości

- ✔ Właściwość zawsze jest tekstem (`String`)

Właściwości obiektów

- ✔ Dostęp do właściwości przez kropkę

```
alert(user.age);       // 30
alert(user.name);      // "John"
alert(user.nosuchkey) // undefined
```

- ✔ Dodawanie właściwości

```
user.isAdmin = true;
```

- ✔ Usuwanie właściwości

```
delete user.age;
```

- ✔ Właściwość ze spacją

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true
};
```

Nawias kwadratowy

- ✔ Dostęp do właściwości

```
alert(user["likes birds"]); // true
alert(user["age"]);         // 30
```

- ✔ Użycie zmiennej

```
let key = "likes birds";
user[key]) = true;
```

Obliczalne właściwości

- ✔ Nazwa właściwości pochodzi ze zmiennej:

```
let fruit = prompt("Wybierz owoc:", "apple");
let bag = {
  [fruit]: 5
};
alert( bag.apple ); // 5, jeżeli fruit="apple"
```

- ✔ Wyrażenia jako nazwa:

```
let fruit = "apple";
let bag = {
  [fruit + 'Computer']: 5, // appleCoputer
};
```

Właściwość ze zmiennej


```
function makeUser(name, age) {
  return {
    name: name,
    age: age
  };
}
let user = makeUser("John", 30);
alert(user.name); // John
```

✔ Można zastąpić przez:

```
function makeUser(name, age) {
  return {
    name,
    age
  };
}
```

Operator `in`

- ✔ `(user.key === undefined)` równoważne `(key in user)`
  - ✘ wyjątek:

```
let obj = {
  test: undefined
};
alert( obj.test ); // undefined
alert( "test" in obj ); // true
```

  - ✓ zazwyczaj wartości takich właściwości ustawiane są w `null`

Wyliczenie właściwości


```
for (let key in object) {
  // key - nazwa pola
  // object[key] - wartość
  ...
}
```

✔ Kolejność: kolejność utworzenia

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25;
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

Pola całkowite

- ✔ Pola całkowite: kolejność arytmetyczna

```
let codes = {
  "48": "Polska",
  "49": "Niemcy",
  "41": "Szwajcaria",
  "1": "USA"
};

for (let code in codes) {
  alert(code); // 1, 41, 48, 49
}
```

- ✔ Jak zrobić, aby Polska była pierwsza na liście?

Rozwiązanie


```
let codes = {
  "+48": "Polska",
  "+49": "Niemcy",
  "+41": "Szwajcaria",
  "+1": "USA"
};

for (let code in codes) {
  alert(+code); // 48, 49, 41, 1
}
```

Kopiowanie obiektów

✔ Zmienne prostych typów kopiowane są przez wartość:

```
let message = "Hello!";
let phrase = message;
```

✘ Dwie zmienne o tej samej wartości

✔ Obiekty kopiowane są przez referencję:

```
let user = { name: "John" };
let admin = user;
```

✘ Dwie zmienne-referencje na ten sam obiekt

```
admin.name = 'Wojtek';
alert(user.name); // Wojtek
```

Porównywanie obiektów

- ✔ Dwa obiekty są równe $\iff$ to jest ten sam obiekt

```
let a = {};
let b = a;
alert( a == b ); // true
alert( a === b ); // true
```

- ✔ W pozostałych przypadkach nie są równe

```
let a = {};
let b = {};
alert( a == b ); // false
```

- ✔ W innych operatorach porównywania obiekty są konwertowane na typy proste

Obiekty stałe można zmieniać


```
const user = {
  name: "John"
};
user.age = 25;
alert(user.age); // 25

// Błąd
user = {
  name: "Pete"
};
```

Klonowanie obiektów

✔ Można skopiować każdą właściwość

```
let user = {
  name: "John",
  age: 30
};

let clone = {};
for (let key in user) {
  clone[key] = user[key];
}

clone.name = "Wojtek";
alert( user.name ); // John
```

Funkcja `Object.Assign`


```
Object.assign(dest, [src1, src2, src3...])
```

- ✔ Kopiuje właściwości wszystkich obiektów `src1, src2, src3...` do obiektu `dest`
  - ✘ jeżeli `dest` ma już taką właściwość, zostanie one zastępiona
- ✔ Można użyć do klonowania

```
let user = {
  name: "John",
  age: 30
};
let clone = Object.assign({}, user);
```

- ✔ Jeżeli wartość jest obiektem, będzie proste kopiowanie

Odśmiecanie pamięci

Osiągalne elementy

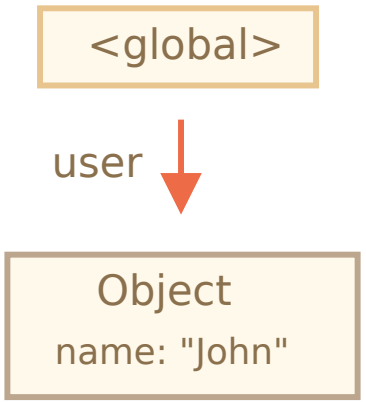
- ✔ Wartości *korzeniowe*
  - ✘ Zmienne lokalne oraz parametry bieżącej funkcji
  - ✘ Zmienne i parametry wszystkich funkcji w łańcuchu wywołań
  - ✘ Zmienne globalne
  - ✘ Niektóre wewnętrzne wartości JavaScriptu
- ✔ Wszystkie wartości, dostępne z wartości korzeniowych po referencjach bądź łańcuchach referencji.

Garbage Collector

- ✔ Automatycznnie dealokuje dane, które nie są osiągalne
- ✔ Przykładowo:

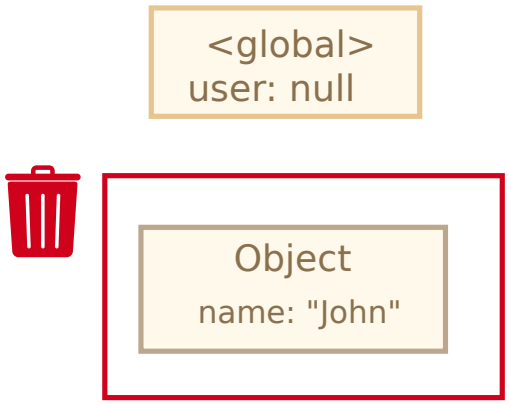
✔

```
let user = {
  name: "John"
};
```



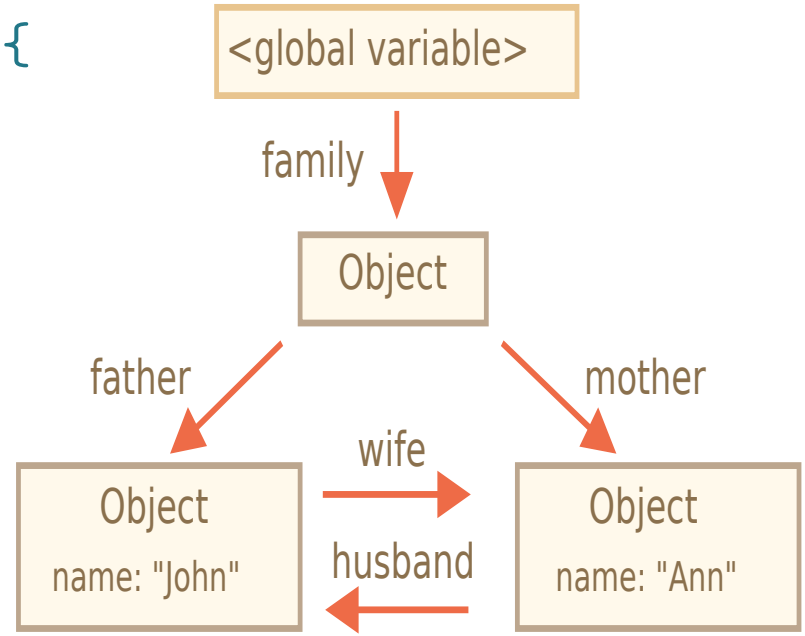
✔

```
user = null;
```



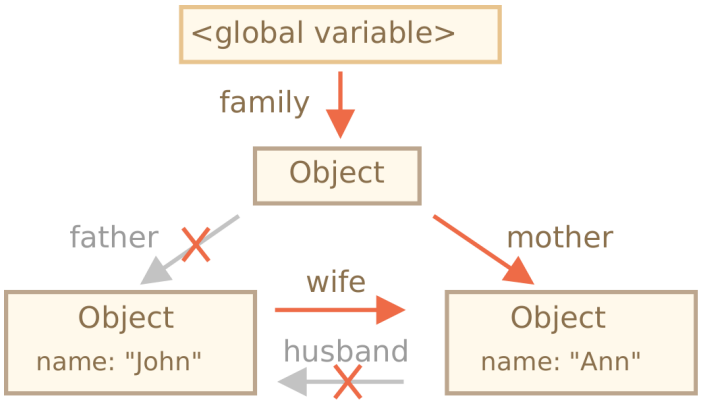
Obiekty powiązane

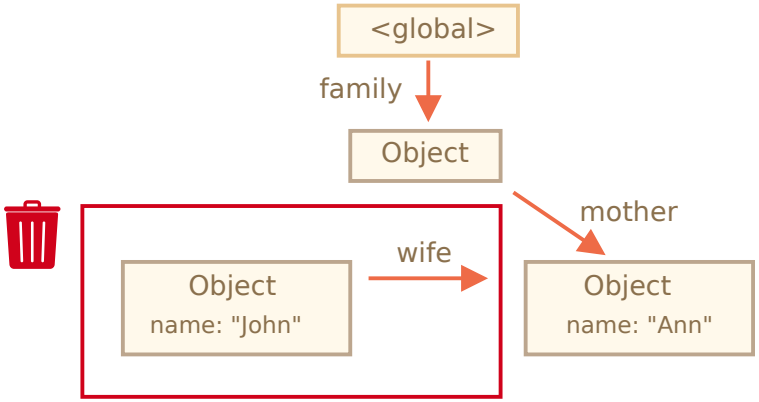

```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;
  return {
    father: man,
    mother: woman
  }
}
let family = marry({
  name: "John"
}, {
  name: "Ann"
})
```



Usuwanie powiązań


```
delete family.father;
delete family.mother.husband;
```





Metody obiektów

Metoda obiektu

✔ Metoda — to pole, wartością którego jest funkcja

```
let user = {
  name: "John",
  age: 30
};
user.sayHi = function() {
  alert("Cześć!");
};
user.sayHi();
```

✔ Można użyć zdefiniowanej nieanonimowej funkcji

Zapis skrócony


```
user = {
  sayHi: function() {
    alert("Cześć");
  }
};
```

✔ to samo co

```
user = {
  sayHi() {
    alert("Cześć");
  }
};
```

this

- ✔ `this` jest referencją na bieżący obiekt
  - ✘ może być użyty w każdej funkcji
  - ✘ obliczany podczas wykonania

```
let user = { name: "John" };
let admin = { name: "Wojtek" };
function sayHi() {
  alert( this.name );
}
user.f = sayHi;
admin.f = sayHi;
user.f(); // John
admin.f(); // Wojtek
admin['f'](); // Wojtek
```

Poza obiektem

- ✔ Jeżeli funkcja jest wykonana nie w kontekście obiektu, to `this === undefined`

```
sayHi(); // undefined
```

  - ✘ w *starym* JavaScripcie obiekt `window`

- ✔ Funkcje-strzałki nie mają własnego `this`
  - ✘ wykorzystany jest `this` nadrzędnego kontekstu

```
let user = {
  firstName: "John",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};
user.sayHi(); // John
```

Konwersja obiektów

Konwersja na proste typy

- ✔ W kontekście logicznym zawsze `true`
- ✔ Trzy warianty (`hint`):
  - ✘ `string`
  - ✘ `number`
  - ✘ `default`
- ✔ Objekt `obj` konwertowany jest metodą:
  1. `obj[Symbol.toPrimitive](hint)`, jeżeli taka metoda istnieje
  2. jeżeli `hint == string`, metodą `obj.toString()`
     (a) jeżeli takiej metody brak, to `obj.valueOf()`
  3. jeżeli `hint == number` albo `hint == default`, metodą `obj.valueOf()`
     (a) jeżeli takiej metody brak, to `obj.toString()`

obj[Symbol.toPrimitive]


```
let user = {
 name: "John",
 money: 1000,
 [Symbol.toPrimitive](hint) {
  alert(`hint: ${hint}`);
  if(hint == "string") return `{name: "${this.name}"}`;
  return this.money;
 }
};

alert(user); // hint: string -> {name: "John"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500
```

✔ Metoda powinna zwrócić typ prosty (nie obiekt)

toString/valueOf


```
let user = {
  name: "John",
  money: 1000,
  toString() {
    return `{name: "${this.name}"}`;
  },
  valueOf() {
    return this.money;
  }
};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

Konstruktor

Funkcja konstruktor


```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("John");
alert(user.name); // John
alert(user.isAdmin); // false
```

- ✔ Każda funkcja może być konstruktorem
  - ✘ anonimowa:

    ```
    let user = new function(...) {...}
    ```
- ✔ Można bez nawiasu, jeżeli nie ma parametrów:

  ```
  let user = new User;
  ```

return

- ✔ Jeżeli konstruktor zawiera `return`, to on zwraca
  - ✘ obiekt, jeżeli `return` jest z obiektem
  - ✘ `this`, jeżeli `return` jest pusty bądź z prostym typem danych

```
function BigUser() {
  this.name = "John";
  return { name: "Godzilla" };  // <-- ten obiekt
}

function SmallUser() {
  this.name = "John";
  return; // <-- zwraca this
}
```

Metody w konstruktorze

✔ W konstruktorze można również tworzyć metody

```
function User(name) {
  this.name = name;
  this.sayHi = function() {
    alert( "Nazywam się: " + this.name );
  };
}
```

Typy Danych

Metody danych prymitywnych

- ✔ Dane prymitywne `string`, `number`, `symbol`, `boolean` mają metody
- ✔ Przykładowo

```
let name = "John";
name.toUpperCase();
1.23456.toFixed();
123456..toString(16);
```

  - ✘ utworzony jest tymczasowy obiekt odpowiednio `String`, `Number`, `Symbol`, `Boolean`
  - ✘ wykonuje się odpowiednia metoda
  - ✘ obiekt jest usuwany
    - ✓ `let x = Number(0)` tworzy prymityw
    - ✓ `let x = new Number(0)` tworzy obiekt (nie jest zalecane)

Number

- ✔ `num.toString(base)` ($1 \leqslant \texttt{base} \leqslant 36$)
- ✔ `num.toFixed(precision)` — zaokrąglenie

```
let num = 255;
alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
alert( 123456..toString(36) ); // 2n9c

let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // 0.30
alert( 2.52.toFixed() ); // 3
```

String

- ✔ `str.length`
- ✔ `str[i] === str.charAt(i)` — numeracja od zera
- ✔ `str.toUpperCase()`, `str.toLowerCase()`
- ✔ `str.indexOf(substr, pos)`, `str.lastIndexOf(substr, position)`
- ✔ `includes`, `startsWith`, `endsWith`
- ✔ `substring`, `substr`, `slice`
- ✔ Wyrażenia regularne
- ✔ etc

```
str[0] = 'h'; // nie działa!
```

Tablice

Tablice sekwencyjne


```
let arr = new Array("my", "array");
alert(arr.length);  // 2

arr = [ "my", "array" ];
alert(arr[0]);

arr = ["one", "two"];
arr.push("three");

arr = new Array(2);
alert(arr.length);  // 2
```

Pętle


```
let fruits = ["Jabłko", "Pomarańcza", "Śliwka"];

for(let i=0; i<fruits.length; i++) {
  alert(fruits[i]);
}

for (let fruit of fruits) {
  alert( fruit );
}
```

Stos i Kolejka


```
var arr = [3,5,7]
arr.push(9)
var last = arr.pop()     //= 9
var last = arr.pop()    // = 7
alert(arr.length)    // = 2

var arr = [4,6,8]
arr.unshift(2) // arr = [2,4,6,8]
arr.unshift(0) // arr = [0,2,4,6,8]
var last = arr.shift()
      // last = 0, arr = [2,4,6,8]
arr.shift()  // arr = [4,6,8]
```

split


```
var names = 'Olek, Bolek, Lolek';

var arr = names.split(', ');

for (var i=0; i<arr.length; i++) {
  doSomething(arr[i]);
}
```

✔ drugi argument — wielkość tablicy

```
alert( "a,b,c,d".split(',', 2) ); // a,b
```

✔ rozbić na litery:

```
var str = "test";
alert( str.split('') ); // t,e,s,t
```

join

```
var arr = ['Olek', 'Lolek', 'Bole'];

var str = arr.join(';');

alert(str); // Olek;Lolek;Bolek
```

Usuwanie elementu — delete


```
var arr = ['Olek', 'Lolek', 'Bole'];

delete arr[1];

alert(arr[1]); // undefined
```

- ✔ usuwa się para `klucz => wartość`
- ✔ inne metody:
  - ✘ `pop`, `shift`
  - ✘ `splice`

Usuwanie elementu — splice


```
arr.splice(index[, deleteCount, el1, ..., elN])
```

- ✔ usunąć `deleteCount` elementów
- ✔ poczynając od `index`
- ✔ wstawić na ich miejsce `el1, ..., elN`

splice. Przykłady


```
arr.splice(index[, deleteCount, el1, ..., elN])
```

- ✔ usunąć jeden element, poczynając od miejsca 1
  `arr.splice(1, 1);`
- ✔ usunąć przedostatni element `arr.splice(-2, 1);`
- ✔ usunąć wszyskie elementy, poczynając od miejsca 2
  `arr.splice(1);`
- ✔ usunąć trzy elementy, na ich miejsce wstawić dwa inne
  `arr.splice(0,3,"raz","dwa");`
- ✔ wstawić dwa elementy `arr.splice(3,0,"raz","dwa");`
- ✔ elementy usunięte `removed=arr.splice(-3,2);`

Kopiowanie elementów — `slice`


```
arr.slice(start, end);
```

- ✔ pierwotna tablica się nie zmienia
- ✔ `end` się nie zalicza
- ✔ `start` może być ujemnym
- ✔ bez `end` kopiowanie do końca tablicy
- ✔ bez `start` kopiowanie całej tablicy

Sortowanie — sort


```
var arr = [ 1, 2, 15 ];

arr.sort();

alert( arr );  // 1, 15, 2
```

Własna funkcja porównująca


```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a < b) return -1;
}
var arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);
alert(arr);  // 1, 2, 15
```

- ✔ funkcja porównywania $f(a, b)$ zwraca wartość
  - ✘ dodatnią, jeżeli $a > b$
  - ✘ ujemną, jeżeli $a < b$
  - ✘ inną, jeżeli $a = b$

```
arr.sort( (a, b) => a - b );
```

Obrócenie tablicy — reverse


```
var arr = [1,2,3];
arr.reverse();

alert(arr); // 3,2,1
```

Połączenie tablic — concat


```
var arr = [1,2];
var newArr = arr.concat(3,4);

alert(newArr); // 1,2,3,4

var arr = [1,2];
var newArr = arr.concat( [3,4], 5);

alert(newArr); // 1,2,3,4,5
```

Wyszukiwanie elementów — indexOf/lastIndexOf


```
arr.indexOf(searchElement[, fromIndex])
```

- ✔ Zwraca numer elementu `searchElement` w tablicy, albo -1, jeżeli takiego elementu nie ma
- ✔ Przy wyszukiwaniu używany jest operator identyczności (`===`)
- ✔ Jeżeli dany jest `fromIndex`, to wyszukiwanie zaczyna się od tego miejsca

```
arr.lastIndexOf(searchElement[, fromIndex])
```

- ✔ Szuka od końca tablicy

Array.from()


```
Array.from(arrayLike[, mapFn[, thisArg]])
```

- ✔ Tworzy nową instację tablicy z obiektu podobnego do tablicy lub obiektu iterowalnego
  - ✘ obiekt z właściwością `length` oraz liczbowymi elementami
  - ✘ obiekt dla którego można wyliczyś jego elementy
    - ✓ `Map`
    - ✓ `Set`
- ✔ `mapFn` — funkcja mapująca
- ✔ `thisArg` — wartość używana jako `this` podczas wykonywania `mapFn`

forEach

- ✔ `arr.forEach(callback[, thisArg])`
- ✔ odpala na dażdym elemencie `callback(item, i, arr)`
  - ✘ `item` jest bieżącym elementem
  - ✘ `i` jest indeksem
  - ✘ `arr` jest całą tablicą
- ✔ `thisArg` pozwala odpalić iterator w innym kontekście
- ✔ wydrukować tablicę:

```
function logArrayElements(element, index, array) {
  console.log('a[' + index + '] = ' + element);
}

[2, 5, , 9].forEach(logArrayElements);
```

filter

- ✔ `arr.filter(callback[, thisArg])`
- ✔ Tworzy nową tablicę z elementów, na których `callback(item, i, arr)` zwróci `true`
  - ✘ `item` jest bieżącym elementem
  - ✘ `i` jest indeksem
  - ✘ `arr` jest całą tablicą
- ✔ `thisArg` pozwala odpalić iterator w innym kontekście
- ✔ tablica z liczb dodatnich:

```
var arr = [1, -1, 2, -2, 3];
var positiveArr = arr.filter(function(number) {
  return number > 0;
});
alert( positiveArr ); // 1,2,3
```

map

- ✔ `arr.map(callback[, thisArg])`
- ✔ Tworzy nową tablicę — wyniki odpalaniacallback(item, i, arr) na każdym elemencie
  - ✘ item jest bieżącym elementem
  - ✘ i jest indeksem
  - ✘ arr jest całą tablicą
- ✔ thisArg pozwala odpalić iterator w innym kontekście
- ✔ Długości łańcuchów tekstowych:

```
var names = ['HTML', 'CSS', 'JavaScript'];
var nameLengths = names.map(function(name) {
  return name.length;
});
alert( nameLengths ); // 4,3,10
```

every/some

- ✔ `arr.every(callback[, thisArg])` zwraca `true`, jeżeli `callback` zwróci `true` dla każdego elementu
- ✔ `arr.some(callback[, thisArg])`zwraca `true`, jeżeli `callback` zwróci `true` dla co najmniej jednego elementu
- ✔ Pzykład:

```
var arr = [1, -1, 2, -2, 3];

function isPositive(number) {
  return number > 0;
}

alert( arr.every(isPositive) ); // false
alert( arr.some(isPositive) ); // true
```

reduce/reduceRight

- ✔ `arr.reduce(callback[, initialValue])` odpala `callback(previousValue, currentItem, i, arr)` na każdym elemencie
  - ✘ `previousValue` wynikiem obliczenia na poprzedniej iteracji
  - ✘ `item` jest bieżącym elementem
  - ✘ `i` jest indeksem
  - ✘ `arr` jest całą tablicą
- ✔ Jeżeli nie ma `initialValue`, to iteracje się zaczynają od drugiego elementu, a pierwszy jest wykorzystany jako `previousValue`
- ✔ `arr.reduceRigh(callback[, initialValue])` — analogicznie, tylko z prawej strony

Przykład z `reduce`

- ✔ Suma elementów:

```
var arr = [1, 2, 3, 4, 5]
var result = arr.reduce(function(sum, current) {
  return sum + current;
}, 0);
alert( result ); // 15
```

- ✔ Co będzie bez `initialValue`?

```
var arr = [1, 2, 3, 4, 5]
var result = arr.reduce(function(sum, current) {
  return sum + current;
});
alert( result ); // ?
```

Wielokropek

Operator reszty

- ✔ Pozostałe argumenty funkcji
  - ✘ tablica

```
function sum(...args) { // args - tablica
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sum(1) ); // 1
alert( sum(1, 2) ); // 3
alert( sum(1, 2, 3) ); // 6
```

Wyodrębnić kilka pierwszych argumentów


```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName );
  alert( titles[0] );
  alert( titles[1] );
}

showName("Cezary", "Obracht-Prondzyński", "prof. zw.",
   "dr hab");
  // titles = ["prof. zw.", "dr hab"]
```

Reszta jest zawsze na końcu


```
// arg2 po ...rest - bezsens!
function f(arg1, ...rest, arg2) {
  // Błąd
}
```

Lista argumentów


```
function func(a,b) {
  alert(arguments[0])
  alert(arguments[1])
  alert(arguments[2])
}
func(1,2,3)

function sum() {
  let s = 0
  for(let i=0; i<arguments.length; i++) {
      s += arguments[i];
  }
  return s
}
```

Uwagi o `arguments`

- ✔ `arguments` nie jest tablicą, reszta — jest
- ✔ Funkcje-strzałki nie mają `arguments`
  - ✘ korzystają z `arguments` z kontekstu nadrzędnego

Operator rozwinięcia

✔ Przekształca tablicę w listę argumentów

```
let arr1 = [3, 5, 1];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr2) );
alert( Math.max(...arr1, ...arr2) );
alert( Math.max(1, ...arr1, 2, ...arr2, 25) );
```

Połączenie tablic


```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged)
```

Tekst na tablicę


```
let str = "Cześć";

alert( [...str] );
```

✔ Analogicznie `alert( Array.from(str) )`

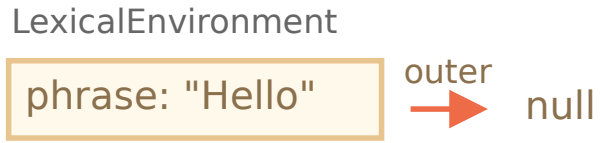
- ✘ `Array.from()` działa dla obiektu podobnego do tablicy lub obiektu iterowalnego
- ✘ operator rozwinięcia — tylko dla obiektu iterowalnego

Domknięcia

Środowisko leksykalne

- ✔ Każda funkcja, blok kodu oraz skrypt ma powiązany ukryty obiekt `LexicalEnvironment`, środowisko leksykalne
- ✔ Dwie części:
  - ✘ `Environment Record` — obiekt, który zawiera wszystkie lokalne zmienne
    - ✓ i inną informację, np. wartość `this`
  - ✘ referencja na zewnętrzne środowisko leksykalne

```
let phrase = "Hello";

alert(phrase);
```



Zmienne a środowisko leksykalne

✔ Zmienne to są właściwości odpowiedniego środowiska leksykalnego

początek wykonania ---------

<puste>

outer → null

```
let phrase;
```

----

phrase: undefined

```
phrase = "Hello";
```

----

phrase: "Hello"

```
phrase = "Bye";
```

----

phrase: "Bye"

Deklaracja funkcji

✔ Funkcje są tworzone w momencie utworzenia środowiska leksykalnego

początek wykonania ---------


```
let phrase = "Hello";
```

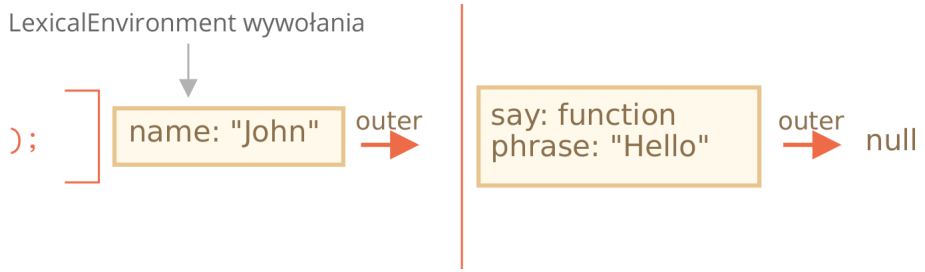
----

say: function
phrase: "Hello"

```
function say(name) {
  alert( `${phrase}, ${name}` );
}
```

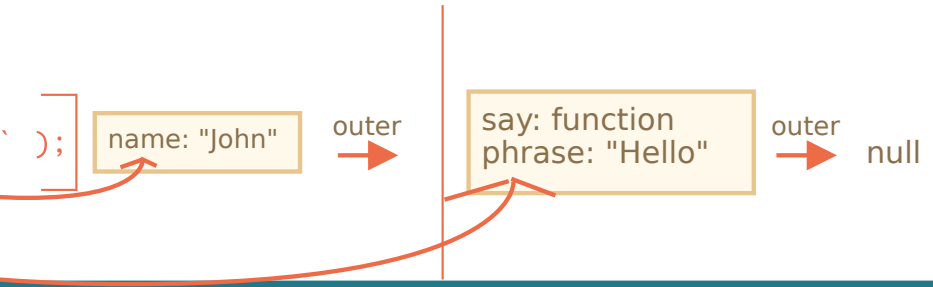
Zewnętrzne środowisko leksykalne


```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```



- ✔ Jeżeli w środowisku leksykalnym nie ma zmiennej, JavaScript będzie szukał jej w zewnętrznym środowisku
  - ✘ aż do globalnego
    - ✓ jeżeli tam się nie znajdzie, będzie błąd wykonania
      - ✗ w starym JavaScripcie zostanie utworzona zmienna globalna

```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```



Przykład

✔ Jaki komunikat zostanie wyświetlony?

```
let name = "John";
function sayHi() {
  alert("Hi, " + name);
}
name = "Wojtek";
sayHi();
```

✘ Każde wywołanie funkcji tworzy nowe lokalne środowisko leksykalne

✘ Programista nie ma dostępu do środowiska leksykalnego

Funkcje włożone

✔ Funkcja zadeklarowana wewnątrz innej funkcji

```
function sayHiBye(firstName, lastName) {

function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );
}
```

Konstruktor obiektu


```
function User(name) {
  this.sayHi = function() {
    alert(name);
  };
}

let user = new User("John");
user.sayHi();
```

Licznik


```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}

let counter = makeCounter();
alert( counter() );
alert( counter() );
alert( counter() );
```

- ✔ Jakie komunikaty zostaną wyświetlone?
- ✔ Jakie wyzerować licznik?

Dwa liczniki


```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}
let counter1 = makeCounter();
let counter2 = makeCounter();
alert( counter1() );
alert( counter1() );
alert( counter2() );
```

✔ Jakie komunikaty zostaną wyświetlone?

Fabryka funkcji


```
function makeWorker() {
  let name = "Pete";
  return function() {
    alert(name);
  };
}

let name = "Wojtek";
let work = makeWorker();
work();
```

✔ Jaki komunikat zostanie wyświetlony?

if


```
let phrase = "Hello";

if( true ){
    let user = 'John';
    alert(`${phrase} ${user}`);
}

alert( user);
```

✔ Jakie komunikaty zostaną wyświetlone?

Pętle


```
for (let i = 0; i < 10; i++) {
}

alert(i);
```

✔ Jaki komunikat zostanie wyświetlony?

Blok


```
{
  let message = "Hello";
  alert(message);
}

alert(message);
```

✔ Jakie komunikaty zostaną wyświetlone?

IIFE

- ✔ Stary JavaScript, zmienne `var`: immediately-invoked function expressions

```
(function() {
  var message = "Hello";
  alert(message); // Hello
})();
```

Odśmiecanie pamięci


```
function f() {
  let value = 123;
  function g() { alert(value); }
  return g;
}
let g = f()
```

- ✔ Zmienna `value` jest osiągalna
  - ✘ Jeżeli zmienna nie jest wykorzystywana, silnik może ją usunąć

Zawieszone wykowanie

setTimeout

- ✔ Odpalić funkcję za `delay` milisekund

```
let timerId = setTimeout(func|code, [delay],
      [arg1], [arg2], ...)
```

- ✔ Odwołać:

```
clearTimeout(timerId);
```

- ✔ Przykład:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John");
```

0 milisekund


```
setTimeout(() => alert("World"));
alert("Hello");
```

- ✔ Zadanie ustawia się w kolejce
  - ✘ najpierw wykona się bieżący kod

setInterval

- ✔ Odpalać funkcję co `delay` milisekund

```
let timerId = setInterval(func|code, [delay],
      [arg1], [arg2], ...)
```

- ✔ Odwołać:

```
clearInterval(timerId);
```

- ✔ Przykład:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setInterval(sayHi, 1000, "Hello", "John");
```

Dekoratory Metod

Funkcja dekorator

- ✔ Funkcja, która modyfikuje zachowanie obiektu/funkcji
  - ✘ przykładowo, mamy funkcję `slow(x)`, która wykonuje ciężkie obliczenia

```
function slow(x) {
  // ciężkie obliczenia
  alert(`Called with ${x}`);
  return x;
}
```

  - ✘ zamieniamy na wersję z cache'owaniem
    - ✓ jeden raz obliczoną funckję zapamiętać w cache'u
    - ✓ przy następnym wywołaniu wziąć wartość z chache'a
      - ✘ jeżeli nie ma zapamiętanej, wywołać `slow(x)` i zapamiętać

Implementacja


```
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x);
    cache.set(x, result);
    return result;
  };
}

slow = cachingDecorator(slow);
alert( slow(2) ); // slow(2)
alert( "Again: " + slow(2) ); // z cache'a
```

Metoda obiektu

✔ Dla metod obiektów to nie działa:

```
let worker = {
  someMethod() {
    return 1;
  },
  slow(x) {
    // ciężkie obliczenia
    alert("Called with " + x);
    return x * this.someMethod();
  }
};
```

Próba implementacji


```
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (*)
    cache.set(x, result);
    return result;
  };
}
alert( worker.slow(1) ); // działa

worker.slow = cachingDecorator(worker.slow);
alert( worker.slow(2) ) // błąd (this === undefined)
```

Kontekst (funkcja `call()`)


```
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // (*)
    cache.set(x, result);
    return result;
  };
}
worker.slow = cachingDecorator(worker.slow);
alert( worker.slow(2) ) // Działa
```

✔ `func.call(context, arg1, arg2, ...)`

✘ `func.apply(context, args)`

Zapożyczenie metody

✔ Funkcja, która wyświetla swoje argumenty

```
function hash() {
  return arguments.join();
}
```

✘ nie działa, `arguments` nie jest tablicą

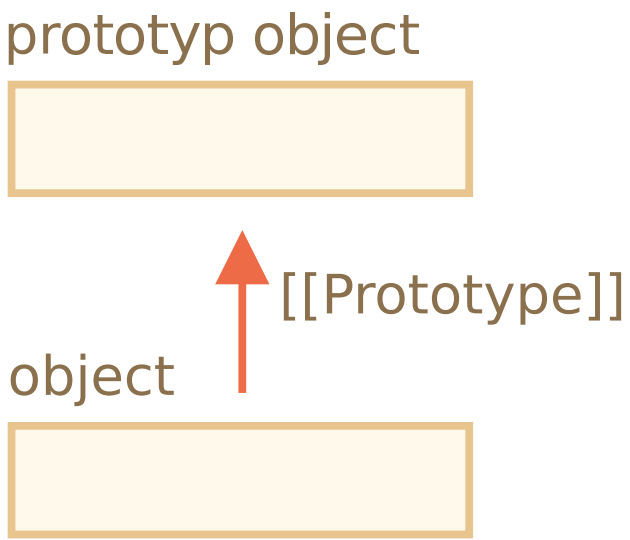
✔ Wykorzystanie metody `join` tablicy:

```
function hash() {
  return [].join.call(arguments);
}
```

Dziedziczenie

Dziedziczenie prototypowe

- ✔ Obiekty dziedziczą bezpośredno od obiektów (omijając klasy)
- ✔ Ukryta referencja `[[Prototype]]` wskazuje, jaki obiekt jest prototypem
  - ✘ może być `null`
- ✔ Jeżeli obiekt nie ma pewnej właściwości, jej się szuka w prototypie



Dostęp do prototypu

- ✔ Poprzez właściwość `__proto__`
  - ✘ powinien być wspierany tylko przez przeglądarki
    - ✓ jest wspierany przez wszystkie silniki
- ✔ Poprzez metody obiektu `Object`
  - ✘ `Object.create(proto, [descriptors])` — tworzy pusty objekt z `[[Prototype]]` równym `proto`
  - ✘ `Object.getPrototypeOf(obj)`
  - ✘ `Object.setPrototypeOf(obj, proto)`

Przykład


```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};
let rabbit = {
  jumps: true,
  __proto__: animal
};
let longEar = {
  earLength: 10,
  __proto__: rabbit
};
longEar.walk(); // Animal walk
alert(longEar.jumps); // true
```

Przykrycie

✔ Przy nadaniu wartości nie jest wykorzystywany prototyp

```
let animal = {
  eats: true,
  walk() {
  }
};
let rabbit = {
  __proto__: animal
};
rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};
rabbit.walk(); // Rabbit! Bounce-bounce!
```

this

✔ `this` zawsze wskazuje na bieżący obiekt

```
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};
let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

rabbit.sleep();
alert(rabbit.isSleeping);
alert(animal.isSleeping);
```

Pętla `for in`

- ✔ Petla `for .. in` wylicza również dziedziczone właściwości
  - ✘ pozostałe metody je ignorują
  - ✘ filtracja: `obj.hasOwnProperty(prop)`

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true,
  __proto__: animal
};
alert(Object.keys(rabbit)); // jumps
for(let prop in rabbit) alert(prop) // jumps, eats
```

F.prototype

Konstruktor `new`

✔ Jeżeli funkcja `F` ma właściwość `prototype`, to `let obj = new F(...)` ustawia `obj.[[Prototype]]` w `F.prototype`

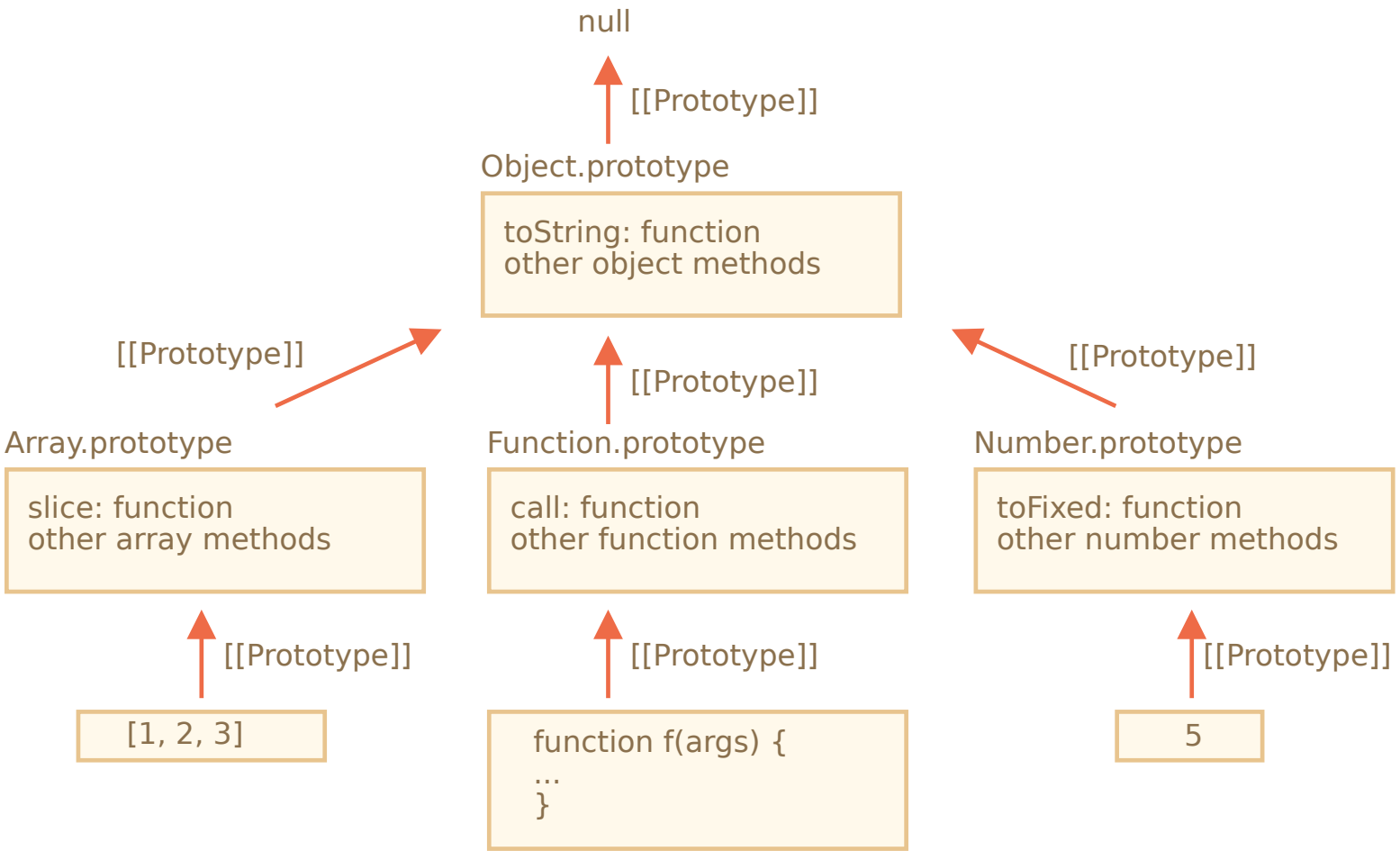
```
let animal = {
  eats: true
};
function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit");
alert( rabbit.eats ); // true
```

Domyślny `prototype`

- ✔ Każda funkcja `F` ma domyslną właściwość `prototype`
- ✔ obiekt `{constructor: F}`

Obiekty wbudowane

- ✔ Obiekty wbudowane mają wszystkie metody zaimplementowane w `prototype`



Zmiana domyślnych prototypów


```
Number.prototype.squared = function(){
    return this*this;
}

alert 2..squared();
```

✔ ... nie jest zalecana

Polyfill


```
if (!String.prototype.repeat) {
  String.prototype.repeat = function(n) {
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Zapożyczenie metody


```
let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!
```

Wyjątki

Wyjątki


```
try {
  ...
  throw {message : "server timeout"}
  ..
} catch (e) {
  alert("File not found")
}
finally{
  zawsze();
}
```

- ✔ blok `finally` jest opcjonalny
- ✔ w `throw` może być dowolny obiekt

Obiekt Error


```
try {
  ...
  throw new Error('connection refused')
  ..
} catch (err) {
  alert(err.message)
}
```

✔ W FF `err.stack`

Obiekt Error


```
ConnError.prototype=Error;
try {
  ...
  throw new ConnError('connection refused')
  ..
} catch (err) {
  if(err instanceof ConnError){
    reconnect();
  }
  else{
    alert(err.message)
  }
}
```

JSON

Format JSON

- ✔ RFC 4627
- ✔ format dla danych w oparciu o JavaScript
- ✔ zpisuje się jako obiekt $\{\cdots\}$ lub tablica `[···]`, które zawierają
  - ✘ tekst w cudzysłowie (nie w apostrofie)
  - ✘ liczby
  - ✘ stałe `true`/`false`
  - ✘ `null`

Przykład JSON


```
{
  "name": "Aleksander",
  "surname": "Kowalski",
  "age": 35,
  "isAdmin": false
}
```

Metody `stringify` oraz `parse`

- ✔ *serializacja* —

  `JSON.stringify(value, replacer, space)`

  zamienia wartość w tekst (IE8+)
- ✔

  `JSON.parse(value, reviver)`

  zamienia tekst w obiekt JavaScript

stringify oraz parse. Przykład


```
var event = {
  title: "Conference",
  date: "Today"
};

var str = JSON.stringify(event);
alert( str ); // {"title":"Conference",
              // "date":"Today"}

event = JSON.parse(str);
```

Serializacja `toJSON`

- ✔ Przy serializacji obiektu jest wywołana metoda `toJSON`.
- ✔ Jeżeli nie ma takiej metody, to wymienia się wszystkie właściwości, oprócz funkcji

```
function Room() {
    this.number = 103;
    this.occupy = function() {};
}
event = {
  title: "Confertence",
  date: new Date(2014, 3, 6),
  room: new Room()
}; alert( JSON.stringify(event) );
```

- ✔ Zobacz

toJSON

- ✔ Iny przykład

```
function Room() {
    this.number = 103;

 this.toJSON = function() {
     return this.number;
  };

}

alert( JSON.stringify( new Room() ) );
```

- ✔ Zobacz

JSON a elementy DOM

- ✔ Elementy DOM — to są skomplikowane obiekty, które mają cykliczne referencje

```
var user = {
  name: "Aleksander",
  age: 25,
  elem: document.body
}

alert( JSON.stringify(user) ); //
```

- ✔ Zobacz

Wyłączenie właściwości w `stringify`

- ✔ W drugim argumencie (`replacer`) funkcji `stringify` można wskazać tablicę właściwości, które należy serializować

```
var user = {
  name: "Aleksander",
  age: 25,
  elem: document.body
}

alert( JSON.stringify(user,["name", "age"]));
```

- ✔ Zobacz

Funkcja jako `replacer`

- ✔ Jako `replacer` można użyć funkcji

  `function(key, value)`,

  która dla klucza `key` zwraca:

  - ✘ serializowane `value`
  - ✘ `undefined`, jeżeli nie trzeba tej właściwości serializować
- ✔ funkcja `replacer` działa rekurencyjnie

Funkcja replacer — przykład


```
var user = {
  name: "Aleksander",
  age: 25,
  elem: document.body
}
var str = JSON.stringify(user, function(key, value)
  if (key == 'elem') return undefined;
  return value;
} );

alert(str);
```

✔ Zobacz

Formatowanie serializacji

- ✔ Trzeci argument funcji `stringify` określa formatowanie
  - ✘ liczba: poziomy włożoności formatuje się podaną ilością spacji
  - ✘ tekst: na każdym poziomie wstawia się ten tekst

```
var user = {
  name: "Aleksander",
  age: 25,
  roles: {isAdmin: false, isEditor: true}
};
var str = JSON.stringify(user, "", 4);
alert(str);
```

- ✔ Zobacz

Pasrowanie

- ✔ Przykład

```
var str = '{ \
  "title":"Conference", \
  "date":"2014-03-06T00:00:00.000Z" \
  }  ';

var event = JSON.parse(str);

alert( event.date.getDate() )
```

- ✔ jaki wynik?
- ✔ Zobacz

Parsowanie intelektualne

- ✔ Drugi argument `reviver` funcji `parse` jest funkcją

  `function (key, value)`,

  która zwraca:

  - ✘ przekształcony parametr `value`
  - ✘ `undefined`, jeżeli `value` nie trzeba zmieniać

```
var event = JSON.parse(str,
 function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
})
```

- ✔ Zobacz

Parsowanie za pomocą `eval`

- ✔ Jest starą metodą (IE7-)

```
var str= '{ \
    "name": "Aleksander", \
    "age": 25 \
}';
var user = eval('(' + str + ')');
alert(user.name)
```

- ✔ niebezpieczeństwo — można przekazać dowolny skrypt
- ✔ lepiej unikać