Języki interpretowane

dr inż. Andrzej Grosser

Rok akademicki 2017/18

Plan wykładów I

1. Wprowadzenie do języków skryptowych
2. Python - składnia, biblioteka standardowa
3. JavaScript - składnia, bibliotek standardowa

Języki skryptowe

- Język skryptowy jest językiem programowania, który umożliwia uruchamianie w specjalnym środowisku programów (skryptów).
- Z reguły te programy są interpretowane a nie kompilowane.
- Przeznaczeniem tych języków jest automatyzacja powtarzalnych zadań, są również stosowane jako języki osadzone w większych aplikacjach.
- Ze względu na swoje właściwości są czasem nazywane językami bardzo wysokiego poziomu lub nawet językami specyfikacji.

PHP

- Język powstał w połowie lat 90 dwudziestego wieku.
- Twórcą pierwszej wersji języka jest Rasmus Lerdorf.
- Jest językiem wieloplatformowym - implementacje można znaleźć dla różnych systemów (między innymi Windows, Linux, BSD).
- Wspiera programowanie obiektowe i strukturalne.
- Jest językiem dynamicznym z słabą kontrolą typów.
- Jego głównym zastosowaniem jest generowanie stron internetowych na przykład na podstawie danych wprowadzonych w formularz przez użytkownika.
- Jego implementacje są najczęściej interpreterami (uruchamianymi po stronie serwera).
- Najnowsza wersja języka to 7.1.

Ruby

- Został stworzony w 1995 roku przez Japończyka Yukihiro Matsumoto.
- Popularność poza Japonią zdobył dzięki frameworkowi Ruby on Rails.
- Implementacje są interpreterami.
- Jest dynamicznym językiem obiektowym (wszystko jest obiektem) z wsparciem dla programowania funkcyjnego i strukturalnego.
- Jest językiem bardzo elastycznym - przede wszystkim dzięki możliwość podmiany zachowania obiektów w czasie uruchomienia programu - nawet typów wbudowanych.
- Posiada automatyczne odśmiecanie pamięci.
- Najnowszą wersją języka jest 2.4.2.

Tcl

- Tcl (Tool Command Language) został stworzony przez Johna Outershouta.
- Jest interpretowanym językiem dynamicznym.
- Jest używany jako język rozszerzeń, skryptów powłoki itp.
- Jest ściśle zintegrowany z biblioteką graficzną Tk (Tcl/Tk).
- Umożliwia przenoszenie oprogramowania na różne platformy (Unix, Linux, Windows).
- Najnowsza wersja pochodzi z 2017 roku (8.6.7).

Perl

- Perl powstał w 1987 roku, został stworzony przez Larry'ego Walla.
- Perl jest wysokopoziomowym językiem interpretowanym.
- Jest używany przede wszystkim do przetwarzania plików tekstowych, wykonywania operacja na powłoce, a także jako język rozszerzeń.
- Jest wieloplatformowy.
- Najnowsza wersja 5.26.1 pochodzi z września 2017 roku.

Lua

- Lua została stworzona przez Roberto Ierusalimschy-ego, Luiza Henrique de Figueiredo i Waldemara Celesa w 1993.
- Lua jest szybkim, lekkim językiem skryptowym.
- Jest językiem dynamicznie typowanym.
- Jest często używana do tworzenia skryptów dla gier.
- Uruchamiana na maszynie wirtualnej z odśmiecaniem pamięci.
- Najnowsza wersja 5.3.4

Python

- Powstał we wczesnych latach 90 dwudziestego wieku jako następca języka ABC.
- Twórcą języka jest Guido van Rossum.
- Jest językiem wspierającym kilka paradygmatów programowania - między innymi obiektowy, funkcyjny, strukturalny.
- Jest dynamicznym, interpretowanym językiem programowania.
- Filozofia programowania kładzie nacisk na prostotę i czytelność programów (The Zen of Python).
- Zarządzanie pamięcią jest realizowane z wykorzystaniem automatycznego odśmiecania.
- Obecnie są w użyciu dwie linie języka: Python 2.x (najnowsza wersja 2.7.14) i Python 3.x (najnowsza wersja 3.6.3).
- Najważniejsze implementacje to CPython, IronPython, Jython i PyPy.

Cechy Pythona I

Najważniejsze cechy Pythona:

- czysta, łatwa do czytania składnia,
- możliwość introspekcji,
- intuicyjne programowanie obiektowe,
- naturalny sposób wyrażania kodu proceduralnego,
- automatyczne zarządzanie pamięcią,
- pełna modułowość, wsparcie dla hierarchicznych pakietów,
- typy danych wysokiego poziomu,
- oparty na wyjątkach system obsługi błędów,

Cechy Pythona II

- rozległa biblioteka standardowa,
- wiele dodatkowych bibliotek zewnętrznych obejmujących większość dziedzin zastosowań,
- moduły i rozszerzenia mogą być łatwo zapisywane w C, C++ (lub Javie dla Jythona, językach .NET dla IronPython),
- przenośny na różne platformy systemowe - Windows, Linux/Unix, Mac,
- możliwość osadzenia w aplikacjach jako języka skryptowego.

Interpreter Pythona

Uruchamianie trybu interaktywnego:

```
python
```

Powoduje to przejście do:

```
Python 2.7.2 (default, Jan 31 2012, 13:19:49)
[GCC 4.6.2 20120120 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Uruchamianie skryptu:

```
python nazwa_skryptu.py
```

Powoduje to kompilację i uruchomienie skryptu. Powstają plik z bajtkodem maszyny wirtualnej Pythona *.pyc

Podział aplikacji Pythona

1. Programy składają się z modułów (może być jeden)
2. Moduły zawierają instrukcje
3. Instrukcje są zbudowane z wyrażeń
4. Wyrażenia tworzą i przetwarzają obiekty

Zmienne I

- Nie określa się jawnie typu zmiennej - wnioskowanie o typie na podstawie prawej strony operatora przypisania.
- Każda zmienna musi być zdefiniowana

```
x = 4
s = "Ala ma kota"
```

- Brak przypisania wartości do zmiennej skutkuje błędem

```
>>> h
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'h' is not defined
```

Zmienne II

- Można łączyć definicje zmiennych:

```
x = y = z = 0
```

- Zmienne można redefiniować, ponowne przypisanie nie musi być do tego samego typu:

```
x = 4
x = "As"
```

Podstawowe typy danych

Najważniejsze typy danych Pythona:

- liczby
- typ logiczny
- łańcuchy znaków
- listy
- krotki
- zbiory
- słowniki

Liczby i typ logiczny

Typ	Literał	Zmienna
Całkowity	1	x = 4
Rzeczywisty	1.0	y = 2.0
Zespolony	3+1j	com = 2+2J
Logiczny	True	b = False

```
width = 20
height = 45
width * height
2 ** 100
3 * 3.75 / 1.5
9.0 / 2
1j * 1J
1j * complex(0,1)
a=1.5+0.5j
a.real
```

Łańcuchy znaków I

Łańcuchy znaków Pythona

- Przechowują dane tekstowe
- Można je definiować:

```
L = "Mielonka"
strL = 'Golonka'
napis1 = "Bardzo␣długi␣napis␣obejmujący␣wiele\n\
wierszy␣tekstu␣-␣trzeba␣jawnie␣dodawać␣\n\
znaki␣przejścia␣do␣nowego␣wiersza"
napis2 = """
Bardzo długi napisa obejmujący wiele wierszy
tekstu - nie trzeba jawnie dodawać
znaków przejścia do nowego wiersza
"""
```

Łańcuchy znaków II

- Łańcuchy znaków są indeksowane od zera

```
print(strL[0])
```

- Obliczanie długości łańcucha znaków

```
print(len(strL))
```

- Odwoływanie się do wartości od końca:

```
print(strL[len(strL)-1])
print(strL[-1])
```

- Wycinki

```
#Wycina podłańcuch od 3 znaku aż do końca
print(strL[2:len(strL)])
print(strL[2:])
```

Łańcuchy znaków III

- Operacja konkatenacji

```
wynik = strL + L
```

- Mnożenie łańcuchów:

```
wynik = 3 * strL
```

- Łańcuchy znaków są niezmienne - nie można zmienić wartości znaków składowych. Próba przypisania:

```
strL[0] = "M"
```

Zakończy się błędem:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Łańcuchy znaków IV

- Operacje możliwe do wykonania na łańcuchach znaków:

```
strL.find('Go')
strL.replace('Go','Mie')
napis = "kot,pies,rybka")
napis.split(',')
"%s␣%s" % ("Golonka", "Mielonka")
#Wersja >= 2.6
"{0}␣{1}".format("Golonka", "Mielonka")
```

Listy I

- Listy są uporządkowaną kolekcją elementów, elementy umieszcza się pomiędzy parą nawiasów kwadratowych []

```
lst = [1, 2, 3]
```

- Długość listy:

```
len(lst)
```

- Nie muszą zawierać elementów tego samego typu, mogą nawet zawierać listy zagnieżdżone:

```
lst2 = [1, "Kot", 3.0]
lst3 = [[1, "Kot"],3.0]
```

- Konkatenacja lista

```
lst2 + lst3
```

Listy II

- Podobnie jak dla łańcuchów znaków mogą być robione wycinki z list:

```
lst2[1:]
lst2[1:2]
lst[:-1]
```

- Można zmieniać elementy listy (inaczej niż dla łańcuchów znaków):

```
lst2[0] = 2
```

- Operacje specyficzne dla list

```
lst1.append([1,2,3])
lst1.pop(1)
del lst1[1]
```

Listy III

- Sprawdzanie zakresów indeksów - przekroczenie zakresu indeksów skutkuje błędem

```
lst1[10]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Wyrażenia listowe (ang. *list comprehension*)

```
lst4 = [element*2 for element in [1, 2, 3]]
```

Krotki I

- Krotki są niezmienną sekwencją elementów (nie da się zmieniać wartości elementów składowych krotki po jej utworzeniu).
- Używane najczęściej w sytuacji, gdy nie powinno się zmieniać elementów zawartych w kolekcji.
- Kolejne elementy krotki rozdziela się przecinkami:

```
t = 12345, 6789, 'witaj!'
```

- Krotki można zagnieżdżać:

```
u = t, (1, 2, 3, 4, 5)
```

Krotki II

- Liczba elementów w krotce

```
len(t)
```

- Odwołanie do elementów za pomocą indeksów, możliwe jest również robienie wycinków z krotek:

```
t[0]
t[1:3]
```

- Rozpakowanie krotki

```
x, y, z = t
```

Zbiory I

- Zbiory są nieuporządkowaną kolekcją bez powtórzeń:

```
#lst - lista, zb - zbiór
lst = ["kot","pies","kanarek","kot","kura",\
"pies","kot"]
zb = set(lst)
```

- Sprawdzanie przynależności do zbioru:

```
"kanarek" in zb
"skowronek" in zb
```

Zbiory II

- Operacje na zbiorach:

```
a = set('abcdef')
b = set('abcmz')
#Różnica zbiorów
a - b
#Suma zbiorów
a | b
#Część wspólna
a & b
#Różnica symetryczna
a ^ b
```

Słowniki I

- Słowniki są kontenerem przechowującym pary klucz - wartość. Kolejne rozdzielane przecinkami pary są zapisane pomiędzy nawiasami klamrowymi

```
telefony = {'Alek':12345,'Tadek':11111}
```

- Odwołanie do wartości pod kluczem:

```
telefony['Alek']
```

- Przypisanie wartości do nieistniejącego klucza powoduje dodanie nowej pary:

```
telefony['Robert'] = 122222
```

- Przejście po elementach słownika:

Słowniki II

```
for naz, nr in telefony.items():
    print(naz,nr)
```

Słowniki III

- Wybór listy kluczy ze słownika:

```
telefony.keys()
```

- Wybór listy wartości ze słownika:

```
telefony.values()
```

- Sprawdzenie czy klucz o podanej wartości istnieje w słowniku:

```
'Adam' in telefony
```

Pliki I

- Python dostarcza operacji dla plików
- Zapis danych do pliku:

```
f = open('dane.txt','w')
f.write('Napis')
f.close()
```

- Odczyt danych z pliku:

```
f = open('dane.txt')
calyPlik = f.read()
f.close()
```

Instrukcje I

Zbiór instrukcji Pythona:

- Instrukcja przypisania
- Wywołanie funkcji
- Instrukcja if/elsif/else
- Instrukcja for
- Instrukcja while/else
- Instrukcja pusta pass
- Instrukcja break
- Instrukcja continue
- Definicja funkcji def

Instrukcje II

- Definicja klasy class
- Instrukcja powrotu z funkcji return
- Dostęp do modułu import
- Dostęp do składowych modułu from
- Wychwytywanie wyjątków try/except/finally
- Zgłaszanie wyjątków raise
- Dostęp do zmiennych globalnych/modułowych global
- Tworzenie generatorów yield
- Usunięcie nazwy del

Instrukcja warunkowa if I

- Ogólna postać (frazy elif, else są opcjonalne):

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

Instrukcja warunkowa if II

- Przykład:

```
if x < 0:
    print("X mniejsze od zera")
elif x == 0:
    print("X równe zero")
else:
    print("X większe od zera")
```

Instrukcja pętli for

- Ogólna postać:

```
for iterating_var in sequence:
    statements(s)
```

- Przykład:

```
lst = ["eggs", "spam", 120, 124]

for elem in lst:
    print(elem)
for i in range(0,len(lst)):
    print(lst[i])
```

Instrukcja pętli while

- Postać ogólna, pętla jest wykonywana dopóki wyrażenie ma wartość logiczną prawdy:

```
while expression:
    statement(s)
```

- Przykład:

```
licznik = 0
while licznik < 10:
    licznik += 1
```

Instrukcje break, else, continue i pass I

- break przerywa wykonywanie bieżącej iteracji, wykonanie instrukcji pętli zostaje przerwane

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'równe', x, '*', n/x)
            break
    else:
        print( n, 'jest liczbą pierwszą' )
```

- continue przerywa wykonywanie bieżącej iteracji, następuje przejście do kolejnej iteracji

Instrukcje break, else, continue i pass II

- else jest wykonywane na zakończenie pętli, nie jest wykonywane w sytuacji, gdy pętla została przerwana instrukcją break
- pass jest odpowiednikiem instrukcji pustej z C++

```
while True:
    pass
```

Funkcje

- Funkcje umożliwiają grupowanie instrukcji w większą całość
- Ogólna postać

```
def nazwa(listaArg):
    instukcje
```

- Argumenty mogą mieć wartość domyślną
- Wartości zwracane przez funkcję mogą być różnych typów
- Możliwe jest definiowanie funkcji wewnątrz funkcji
- Cechą charakterystyczną wsparcie dla wywołań polimorficznych i *duck typing*

Definiowanie funkcji

```
def silnia(n):
    """Funkcja obliczająca silnię"""
    wynik = 1
    for i in range(1,n+1):
        wynik *= i
    return wynik

print(silnia(5))
print(silnia.__doc__)
print(dir(silnia))
```

Argumenty funkcji

- Argumenty funkcji są traktowane jako zmienne lokalne, dlatego zmiany ograniczają się tylko do funkcji
- Wyjątkiem od zapisanej zasady są obiekty, których zawartość może być zmieniana - listy, słowniki, obiekty klas

```
def zmianaLubNie(a,b,c):
    a = 4
    b[0] = x
    c.a = 14
    b = 4
    c = 4


liczba = 0
lista = [1, 2, 3]
obiekt = Prostokat(0)
zmianaLubNie(liczba, lista, obiekt)
```

Argumenty domyślne i nazwane funkcji

```
def argDomyslne(a = 1, b = 2, c = 3):
    print( "a:{0} b:{1} c:{2}".format(a, b, c))

argDomyslne(4,6,8)
argDomyslne(4,6)
argDomyslne(4)
argDomyslne()
argDomyslne(a = 4)
argDomyslne(c = 8)
argDomyslne(c = 8, b = 4, a = 2)
```

Argumenty *args i **kwargs I

- Są używane dla funkcji, które przyjmują nieznaną na początku liczbę argumentów
- W przypadku *args - są zbierane argumenty z kolejnych pozycji i wstawiane do krotki args

```
def f1(*args):
    print( args)


f1(1,2,3,4,5)
```

- W przypadku **kwargs tworzony jest słownik łączący kolejne nazwy z wartościami argumentów

Argumenty *args i **kwargs II

```
def f2(**kwargs):
    print( kwargs)

f2(a = 1, b = 2, c = 3)

def f2(**kwargs):
    print( kwargs)

f2(a = 1, b = 2, c = 3)
```

- Możliwe jest łączenie obu konwencji

```
def f3(*args,**kwargs):
    print( args, kwargs)

f3(1, 2, a = 1, b = 2)
```

Argumenty *args i **kwargs III

- Nie można jednak zamieniać argumentów. Poniższa linia jest błędna:

```
f3(a = 1, b = 2, 1, 2)
```

Klasy i obiekty

```
class Prostokat:
    zmienna = 12
    def __init__(self,a):
        self.a = a
    def pole(self):
        return self.a * self.a

p = Prostokat(2)
print( p.a)
```

Atrybuty

```
#atrybut klasowy
print(Prostokat.zmienna)
p.nowazmienna = 18
p.zmienna = 1000
#to nie jest atrybut klasowy
#- tworzone jest pole obiektu!
print( p.zmienna)
#Odwołanie do zmiennej klasowej
print(Prostokat.zmienna)
```

Metody

- Metody mogą być wywoływane za pośrednictwem obiektu:

```
p.pole()
```

- Można również przekazywać obiekt jako do metod klas jako pierwszy argument wywołania:

```
Prostokat.pole(p)
```

Metody specjalne I

- Są to metody, których nazwa zaczyna się i kończy __ (dwoma podkreśleniami).
- Służą do celów specjalnych np: inicjowania, definicji operatorów, konwersji typów, reprezentacji obiektów klasy w innej postaci (np. łańcuchów tekstowych)

Metody specjalne II

```
class A:
    def __init__(self, a):
        self.a = a
    def __repr__(self):
        return 'A({0})'.format(self.a)
    def __str__(self):
        return 'Klasa A z a={0}'.format(self.a)

a = A(12)
# Odwrotne apostrofy - pod tyldą - tylko Python2
print( `a`)
print(repr(a))
print(str(a))
```

Dziedziczenie

```
class Super:
    def __init__(self, x):
        self.zmiennaSuper = x

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)
        self.zmiennaSub = y

s = Sub(1, 2)
print(s.zmiennaSuper, s.zmiennaSub)
```

Duck typing

```
class Kaczka:
    def kwacz(self):
        print("Kwaaaa")

class Wabik:
    def kwacz(self):
        print("Fiiii")

def kwakanie(obiekt):
    obiekt.kwacz()

k = Kaczka()
w = Wabik()
kwakanie(k)
kwakanie(w)
```

Moduły I

- Ułatwiają ponowne używanie kodu, mogą zawierać zmienne i funkcje.
- Pozwalają na podział nazw w celu uniknięcia ich konfliktów.
- Umożliwiają współdzielenie danych.
- Zakres modułu pokrywa się z zakresem pliku.
- Kolejne instrukcje są wykonywane tylko raz przy załadowaniu modułu.
- Zawartość modułu można przetwarzać za pomocą dwóch instrukcji i funkcja:
  - Instrukcja import - umożliwia klientowi korzystanie z modułu jako całości, dostęp do składowych przez kropkę

```
import os
os.name
```

  - Instrukcja from - umożliwia w kodzie klienckim korzystanie z wyszczególnionych nazw bezpośrednio

Moduły II

```
from sys import version_info
version_info()
```

- Funkcja reload z modułu imp (imp.reload) - umożliwia ponowne wczytanie modułu

```
import jakismodul
import imp
imp.reload(jakismodul)
```

Moduły III

```
#mojmodul.py
#coding=utf8
x = 10
print("Wywoływane wewnątrz modułu")
print(x)
def funkcja():
    print("funkcja")
print("Wywoływane wewnątrz modułu")
funkcja()

if __name__ == "__main__":
    print("Uruchomiony jako główny skrypt")
#klient.py
import mojmodul
mojmodul.funkcja()
import imp
```

Moduły IV

```
imp.reload(mojmodul)
```

Pakiety

- Moduły można organizować w hierarchiczne struktury, dane są zorganizowane w strukturach katalogowych.

```
import katalog1.katalog2.nazwaModulu
```

- Moduł jest traktowany jako pakiet, gdy zawiera plik __init__.py. Każdy katalog z pakietu musi zawierać ten plik.

Wyjątki

```
while True:
    try:
        liczba = int(raw_input("Wprowadź liczbę:"))
        break
    except ValueError:
        print("To nie jest poprawna liczba!")
```

Własne typy wyjątków

- Można tworzyć własne typy wyjątków
- Klasy wyjątków powinny w bezpośredni lub pośredni sposób dziedziczyć po klasie Exception

```
class MojWyjatek(Exception):
    def __init__(self, wartosc):
        self.wartosc = wartosc
    def __str__(self):
        return repr(self.wartosc)
try:
    raise MojWyjatek(2*2)
except MojWyjatek as e:
    print( e.wartosc)
```

Funkcje jako obiekty I

- Funkcje Pythona są obiektami.
- Mogą być przypisywane do innych zmiennych, przekazywane do innych funkcji, osadzane w strukturach danych, zwracane z funkcji.

```
def kwadrat(x):
    return x * x
def szescian(x):
    return x * x * x

fun = kwadrat
fun(12)
lst = [(kwadrat, 12), (szescian, 10)]
for (f, x) in lst:
    print (f(x))
```

Funkcje jako obiekty II

```
def make(fun):
    def wywolaj(x):
        return fun(x)
    return wywolaj

f1 = make(kwadrat)
f1(12)
f2 = make(szescian)
f2(10)
```

Funkcje jako obiekty III

- Funkcje posiadają własne atrybuty takie jak na przykład nazwa, dokumentacja domknięcie itp.

```
def func(x):
    """Funkcja podnosząca x do potęgi drugiej"""
    return x * x

print(func.__name__)
print(func.__doc__)
print(func.__closure__)
print(dir(func))
```

Funkcje jako obiekty IV

- Moduł `doctest` wyszukuje tekst, który wygląda jak interaktywna sesja Pythona a następnie wykonuje ten kod w celu zweryfikowania, że działa on dokładnie tak jak pokazano.
- Jest kilka sposobów użycia doctest:
  - Do sprawdzenia, że docstringi modułu są nadal zgodne z tym co przykłady pokazują.
  - Do wykonywania testów regresyjnych, które weryfikują czy interaktywne przykłady działają zgodnie ze spodziewanym.
  - Do zapisu samouczka dla modułu, które pokazują jak on działa (wykonywalna dokumentacja).
- Do działania wymaga trybu gadatliwego (-v).

Funkcje jako obiekty V

```
def dodawanie(x, y):
    """Dodaje do siebie x i y
    >>> dodawanie(0, 4)
    4
    >>> dodawanie(5, 4)
    9
    >>> dodawanie(5, -5)
    0
    """
    return x + y

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Funkcje jako obiekty VI

```
$ python testy.py

$ python testy.py -v
Trying:
    dodawanie(0, 4)
Expecting:
    4
ok
Trying:
    dodawanie(5, 4)
Expecting:
    9
ok
```

Funkcje jako obiekty VII

```
Trying:
    dodawanie(5, -5)
Expecting:
    0
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.dodawanie
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

Funkcje jako obiekty VIII

- Podobnie jak zwykłe obiekty, funkcje mogą posiadać dodatkowe atrybuty:

```
def funkcja():
    pass


funkcja.poziom = 1
funkcja.zabezpieczona = True
```

- Atrybuty funkcji są z reguły używane w wyspecjalizowanych dziedzinach zastosowań takich jak generatory parserów i szkielety aplikacji, gdzie dostarczają dodatkowych informacji o obiekcie funkcji.

Funkcje jako obiekty IX

- Jest też możliwe stworzenie obiektu, który się będzie wywoływał jak funkcja.
- Do tego celu należy utworzyć w klasie metodę `__call__`

```
class Przyklad:
    def __call__(self):
        print("Testowe wywołanie obiektu jako
            funkcji")

obj = Przyklad()
obj()
```

Operator lambda I

- Funkcje anonimowe w Pythonie tworzy się z wykorzystaniem wyrażenia lambda.
- Składnia wyrażenia lambda jest następująca:

  `lambda argumenty: wyrazenie`

  gdzie argumenty są listą zmiennych rozdzielonych przecinkami, zaś wyrażenie oznacza wyrażenie, które może korzystać z tych zmiennych.
- Wynikiem obliczenia wyrażenia lambda jest obiekt funkcyjny, który może być przypisany do zmiennej

```
(lambda x: x **2)(12)
fun = lambda x: x **2
fun(12)
```

Operator lambda II

- Wyrażenia lambda w Pythonie są dość ograniczone nie można w nich korzystać z niektórych instrukcji (np. pętli) - dozwolone są jedynie wyrażenia.

```
#Niepoprawne funkcje anonimowe
lambda x: for i in x: print(i)
lambda x: if x>0: 1 else -1
#I jedna poprawna
lambda x: 1 if x>0 else -1
```

- Innymi słowy wyrażenia lambda są jak funkcje z pojedynczą instrukcją powrotu (return).

Operator lambda III

- Wyrażenia lambda są użyteczne w sytuacji, gdy przekazywana funkcja ogranicza się do prostego obliczenia wyrażanie, nie jest wtedy potrzebne tworzenie pełnej definicji funkcji.

```
def wywolajFunkcje(f, x):
    return f(x)
wywolajFunkcje(lambda x: x**2, 12)
#inaczej konieczne
def kwadrat(x):
    return x**2
wywolajFunkcje(kwadrat, 12)
```

Operator lambda IV

- Są również użyteczne, gdy użycie instrukcji (np. definicji funkcji) byłoby niepoprawne składniowo:

```
lst = [lambda x: x, lambda x: x ** 2,
    lambda x: x ** 3]
print lst[1](12)
```

- Definicja literału listowego zakłada użycie wyrażeń a nie instrukcji - definicja funkcji jest, więc w tym przypadku niedozwolona i konieczne jest użycie wyrażeń lambda.

Operator lambda V

- Wyrażenia lambda można zagnieżdżać - można wewnątrz definicji wyrażenia lambda użyć innego wyrażenia lambda.
- W zagnieżdżonych wyrażeniach można wykorzystywać argumenty przekazywane do wyrażenia (wyrażeń zewnętrznych) - zagnieżdżony zasięg funkcyjny.

```
(lambda x: (lambda y: x + y))(12)(4)
#to samo co
def utworzDodaj(x):
    def fun(y):
        return x + y
    return fun
utworzDodaj(12)(4)
```

- Trzeba jednak zachować ostrożność - takie zagnieżdżone konstrukcje bardzo zaciemniają kod.

Narzędzia programowania funkcyjnego I

- Funkcja `map` pozwala na wywołanie określonej funkcji dla wszystkich elementów listy, krotki, słownika.
- Przyjmuje funkcję i listę (listy) (ogólnie coś co pozwala na przechodzenie po swoich elementach), zwraca listę.

```
#[1, 4, 9, 16]
map(lambda x: x ** 2, [1,2,3,4])
#[0, 3, 4, 3, 0]
# range(5) -> [0, 1, 2, 3, 4]
# range(4,-1,-1) -> [4, 3, 2, 1, 0]
map(operator.mul, range(5), range(4,-1,-1))
```

Narzędzia programowania funkcyjnego II

- Funkcja `filter` pozwala na odfiltrowanie z listy tylko tych elementów, dla których predykat (funkcja zwracająca wartość logiczną) zwraca wartość prawdy.
- Funkcja przyjmuje predykat i listę, zwraca listę.

```
#Tylko elementy parzyste
#[0, 2, 4, 6, 8]
filter(lambda x: x%2 == 0, range(10))
```

- Zamiast wyrażenia lambda mógłby być przekazywany obiekt funkcyjny.

Narzędzia programowania funkcyjnego III

- Funkcja `reduce()` - wywołuje dwuargumentową funkcję łącząc kolejne elementy listy od lewej do prawej, powoduje to zredukowanie listy do pojedynczej wartości.
- Przyjmuje funkcję, listę oraz opcjonalną wartość początkową.

```
#Oblicza wartość
#((((1+2)+3)+4)+5)
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
#reduce(operator.add, [1, 2, 3, 4, 5])
```

Narzędzia programowania funkcyjnego IV

- Funkcja `zip()` - zwraca listę krotek, w której każda i-ta krotka zawiera i-te elementy z sekwencji przekazywanych jako argumenty.
- Jeżeli są przekazywane listy o różnej długości to długość wyniku jest obcinana do najkrótszego argumentu.
- Z pojedynczym argumentem zwraca listę jednoelementowych krotek.

```
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
#[(1, 5), (2, 6), (3, 7), (4, 8)]
zip(x, y)
#[(1,), (2,), (3,), (4,)]
zip(x)
#[(1, 5, 1), (2, 6, 2), (3, 7, 3), (4, 8, 4)]
zip(x, y, range(1,100))
```

Narzędzia programowania funkcyjnego V

- Funkcja `partial` z modułu `functools` umożliwia częściową aplikację funkcji, tworzy specjlany obiekt, który jest pośrednikiem do wywołania funkcji.
- Pozwala na ustalenie kilku argumentów funkcji w nowym obiekcie z uproszczoną sygnaturą.

```
basetwo = partial(int, base=2)
#nie 11 111 lecz 31
basetwo('11111')
#alternatywnie
def basetwo(arg):
    return int(arg,2)
```

Dekoratory I

- Dekorator to funkcja, której głównym zadaniem jest przeźroczyste opakowanie innej funkcji lub klasy.
- Podstawowym celem tego zabiegu jest zmiana lub rozszerzenie zachowania opakowywanego obiektu.
- Dekoratory są lukrem syntaktycznym na wywołanie funkcji pobierającej funkcję.
- Dekoratory są oznaczane symbolem `@`.

Dekoratory II

```
@przyklad
def fun(x):
    return x * x * x

#jest równoważne

def fun(x):
    return x * x * x

fun = przyklad(fun)
```

Dekoratory III

```
#Funkcja pomocnik dodaje elementy do wywołania
#funkcji dekorowanej.
#Funkcja przyklad zwraca obiekt tej funkcji.
#Funkcja dekorowana jest wywołana pośrednio.
def przyklad(f):
    def pomocnik(*args,**kwargs):
        print("Wywołano " + f.__name__)
        return f(*args, **kwargs)

    return pomocnik
```

Dekoratory IV

- Dekoratory łatwiej dostrzec niż wywołanie funkcji pomocniczej, która może być w kodzie oddalona od modyfikowanej funkcji lub klasy.
- Dekoratory są aplikowane raz, gdy jest definiowana klasa lub funkcja, nie ma potrzeby dodawania dodatkowego kodu dla każdego odwołania do funkcji lub klasy, które mogą się zmienić w przyszłości.
- Użytkownik rzadziej zapomina dostosować funkcji lub klasy do wymagań API.
- Z drugiej strony dekoratory wprowadzając opakowanie mogą zmieniać typy dekorowanych obiektów - może powodować dodatkowe narzuty związane z dodatkowymi wywołaniami.

Dekoratory V

- Pojedynczą funkcję może dekorować wiele dekoratorów.
- Ma przy tym znaczenie kolejność dekorowania.

```
@foo
@bar
@spam
def funkcja(arg):
    return arg + arg

#jest to równoważne
def funkcja(arg)
    return arg + arg

funkcja = foo(bar(spam(funkcja)))
```

Dekoratory VI

- Dekorator trace, jeżeli jest włączony ślad, zapisuje informacje o wywołaniu funkcji w pliku debug.log.

```
enable_tracing = True
if enable_tracing:
    log = open("debug.log", "w")

def trace(f):
    if enable_tracing:
```

Dekoratory VII

```
        @functools.wraps(f)
        def pomocnik(*args, **kwargs):
            log.write("Wywołanie {0} z argumentami
                {1} i {2}\n"
                .format(f.__name__,args, kwargs))
            wynik = f(*args, **kwargs)
            log.write("Funkcja {0} zwróciła {1}\n"
                .format(f.__name__, wynik))
            return wynik
        return pomocnik
    else:
        return f
```

Dekoratory VIII

```
@trace
def fun(x):
    return x * x * x

fun(10)
fun(8)

#Zawartość pliku debug.log
#Wywołanie fun z argumentami (10,) i {}
#Funkcja fun zwróciła 1000
#Wywołanie fun z argumentami (8,) i {}
#Funkcja fun zwróciła 512
```

Dekoratory IX

- Dekorator memoize pozwala zapamiętać wyniki poprzednich wywołań funkcji - gdy nastąpi kolejne wywołanie z tymi argumentami funkcja nie będzie wywołana, a wynik zostanie pobrany ze słownika.

```
def memoize(obj):
    cache = obj.cache = {}

    @functools.wraps(obj)
    def memoizer(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = obj(*args, **kwargs)
        return cache[key]
    return memoizer
```

Generatory I

- Generatory są funkcjami, które umożliwiają tworzenie sekwencji wartości, które mogą być używane do iterowania.
- Funkcje tego rodzaju używają słowa kluczowego `yield` do zwrotu kolejnych potrzebnych wartości.

```
def sekwencja(n):
    while n < 10:
        yield n
        n += 1
```

- Po wywołaniu funkcji nie jest wykonywany kod, ale tworzony jest specjalny obiekt generatora.

```
x = sekwencja(0)
```

Generatory II

- Obiekt generatora jest wywołuje funkcję aż do słowa yield, gdy wywoływana jest metoda `next()` (`__next__()` w Python 3).
- Przy każdym kolejnym wywołaniu program jest wznawiany od pierwszej instrukcji po yield.

```
x.next() # 0
x.next() # 1 itd aż do 10
```

- Najczęściej obiekt generatora jest wykorzystywany w pętlach:

```
for i in sekwencja(10):
    print i * i
```

Generatory III

- Sekwencja może być potencjalnie nieskończona:

```
def sekwencja(n):
    while true:
        yield n
        n += 1

x = sekwencja(0)
#0
x.next() #x.__next__() w Python 3
#1
x.next()
```

Generatory IV

- Generatory umożliwiają funkcjom uniknięcie wykonywania całej pracy od razu, jest to szczególnie istotne, gdy tworzone listy są duże lub gdy przy tworzeniu listy dla każdego elementu jest wykonywane dużo obliczeń.
- Dodatkowo generatory stanowią prostszą alternatywę do ręcznego zapisywania stanu pomiędzy kolejnymi iteracjami - w generatorach zmienne osiągalne w zasięgu funkcji są zapisywane i przywracane automatycznie.

Generatory V

```
def fibNum(n):
    currNum = 1
    prevNum = 1
    yield 1
    if n == 1:
        return
    yield 1
    i = 2
    while i < n:
        prevNum, currNum = currNum, currNum +
            prevNum
        yield currNum
        i += 1
```

Generatory VI

```
x = fibNum(3)
# 1
next(x)
# 1
next(x)
# 2
next(x)
#Traceback (most recent call last):
#  File "<pyshell#54>", line 1, in <module>
#    next(x)
#StopIteration
next(x)
[i for i in fibNum(100)]
```

Korutyny I

- Instrukcja yield może być również wywoływana po prawej stronie operatora przypisania.
- Funkcja, która używa yield w ten sposób jest nazywana korutyną (ang. *coroutine*).
- Do tak przygotowanej funkcji wysyła się kolejne wartości przy pomocy metody `send()`.
- Należy przy tym pamiętać o wcześniejszym wywołaniu metody `next()` - sterowanie przejdzie do pierwszej instrukcji yield.

Korutyny II

```
def odbiorca():
    print "Gotów do odbioru:"
    while(True):
        elem = (yield)
        print elem


//...
o = odbiorca()
o.next()
o.send(0)
o.send(12)
o.send("Atom")
```

Korutyny III

- Korutyna zazwyczaj może być uruchamiana bez końca, chyba że zostanie jawnie zamknięta lub kończy się.
- Do zamknięcia strumienia wartości wejściowych służy metoda `close()`:

```
o.close()
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#StopIteration
o.send(43)
```

Korutyny IV

- Korutyny mogą jednocześnie przyjmować i zwracać wartości, w sytuacji gdy wartość jest dostarczana w wyrażeniu yield.

```
def potega():
    print "Gotów"
    wynik = 0
    while(True):
        elem = (yield wynik)
        wynik = 2 ** elem
```

Korutyny V

```
o = odbiorca()
#Gotów
#0
o.next()
#4096
o.send(12)
#1024
o.send(10)
```

- Wywołanie metody next() spowoduje przejście do wyrażenia yield, które zwróci 0 (wartość początkowa zmiennej wynik).
- Wartość jaka jest zwracana przy kolejnych wywołaniach send() pochodzi z następnego wyrażenia yield.

Korutyny VI

- Z koniecznością wywołania metody next() można poradzić sobie tworząc odpowiedni dekorator.

```
def corutine(fun):
    def start(*args, **kwargs):
        g = fun(*args, **kwargs)
        g.next()
        return g
    return start
```

- Dekorator automatycznie wykona niezbędne do prawidłowego działania korutyny wywołanie metody next().

Korutyny VII

```
@corutine
def odbiorca():
    print "Gotów do odbioru:"
    while(True):
        elem = (yield)
        print elem

o = odbiorca
#Nie zakończy się błędem
o.send(1212)
```

Literatura

1. M. Lutz "Learning Python" wydanie piąte, O'Reilly Media 2013
2. `https://www.python.org/doc/`

Podstawowa trójka internetowa

Podstawą każdej większej strony internetowej są:

- HTML - specyfikacja zawartości strony
- CSS - specyfikacja wyglądu strony
- JavaScript - specyfikacja zachowania strony.

JavaScript - ogólna charakterystyka

- JavaScript - wysokopoziomowy, dynamiczny, słabo typowany interpretowany język programowania.
- Zawiera wsparcie dla obiektowego i funkcyjnego stylu programowania.
- Jest wspierany przez najważniejsze przeglądarki.

Składowe JavaScript

- Rdzeń - ECMAScript - opisuje składnię i semantykę języka (nie są opisywane szczegóły związane z przeglądarką).
- DOM (Document Object Model) - API dla XML, które zostało rozszerzone do użycia z HTML. DOM opisuje stronę jako hierarchię węzłów.
- BOM (Browser Object Model) - API umożliwiające dostęp i zmiany okna przeglądarki (przesuwanie, otwieranie, zamykanie).

Historia JavaScript

- Język został stworzony przez Brendona Eicha w firmie Netscape w 1995.
- Oryginalną nazwą była Mocha, potem LiveScript a na końcu po uzyskaniu pozwolenia od firmy Sun JavaScript.
- Język jest ustandaryzowany przez ECMA (1996) - w standardzie nosi nazwę ECMAScript (innym językiem, który spełnia ten standard z rozszerzeniami jest ActionScript).
- Najnowszą edycją standardu ECMAScript jest wersja 2017 (czerwiec 2017).

Narzędzia wspierające budowę oprogramowania

- JSFiddle - aplikacja online pozwalająca na uruchamianie i testowanie kodu `http://jsfiddle.net/` (darmowe).
- Firebug - dodatek do Firefoxa - pozwala na testowanie i debugowanie kodu (darmowe).
- Aptana - oparte o Eclipse środowisko programowania aplikacji web (darmowe).
- WebStorm - środowisko programowania aplikacji web (niestety komercyjne).

Osadzenie kodu w przeglądarce

- W dokumencie HTML można umieścić skrypt JavaScript pomiędzy znacznikami <script> i </script>.

```
<script language="JavaScript">
//...
//Treść skrytpu JavaScript
//...
</script>
```

- Można się również odwoływać do zewnętrznych plików z kodem JavaScript:

```
<script src="p.js" type="text/javascript"
    language="JavaScript"></script>
```

Podstawowe elementy I

- Programy JavaScript zapisuje się używając Unicode.
- Jest rozróżniana wielkość liter - np. `A` i a są różnymi identyfikatorami.
- Białe znaki (spacja, tabulacja itp.) są z reguły pomijane w trakcie analizy leksykalnej.
- Jedynym wyjątkiem są znaki nowego wiersza, które mogą być separatorami instrukcji, gdy można jednoznacznie określić, gdzie instrukcja się kończy.
- Opcjonalnymi separatorami instrukcji są znaki średnika.

Komentarze

- Komentarze są takie jak w Javie i C++.
- Jednolinijkowe są zapisywane pomiędzy znakami // i znakiem nowego wiersza.
- Wielolinijkowe komentarze są zapisywane pomiędzy znakami /* i */.
- Na przykład:

```
//Komentarz jednlinijkowy
var x = 10;
/*Komentarz zajmujący
wiele linii*/
```

Podstawowe typy I

- Liczby

  120
  1.0
  .5
  1.2E2

- Wartości logiczne

  true
  false

- Łańcuchy znaków

  "Stała␣łańcuchowa"
  'Inna␣stała␣łańcuchowa'

Podstawowe typy II

- Data i czas

```
var d1 = new Date(2014, 1, 13);
var d2 = new Date(2014, 1, 14, 10, 0, 0);
```

- Wyrażenia regularne - składnia Perla

```
var str = "Liczby: 1, 2, 3";
var wzorzec = /d+/g;
wzorzec.test(str);
str.search(wzorzec);
str.match(wzorzec);
```

Obiekty I

- Obiekty są nieuporządkowaną kolekcją właściwości, które zawierają wartości typów podstawowych, innych obiektów lub funkcji.
- Każda z właściwości obiektu posiada swoją własną nazwę.
- Nazwy właściwości mogą być łańcuchami znaków zawierającymi spacje.
- Obiekty JavaScript (podobnie jak w Pythonie) są dynamiczne - można dodawać i usuwać z nich właściwości.

Obiekty II

```
var student = {
    nazwisko : "Kosmyk",
    wiek : 22,
    "numer indeksu" : 11212,

    daneOsobowe: function() {
        return this.nazwisko + this.wiek + this["
            numer indeksu"];
    }
}
```

Tablice I

- Tablica jest uporządkowaną kolekcją wartości.
- Tablice JavaScript są specjalnym rodzajem obiektów, nazwami właściwości są kolejne liczby całkowite (licząc od zera).
- Elementy tablicy w JavaScript mogą mieć dowolny typ (nawet różny).
- Dostęp do elementów jest realizowany za pomocą operatora indeksowania.
- Liczbę elementów tablicy można określić za pomocą własności length.

Tablice II

```
var tab1 = [1,2,3,4];
var tab2 = [1,true, "x"];
var tab3 = new Array(20);

for(var i = 0; i < tab1.length; ++i)
    tab1[i] = 12;
```

Wartości null i undefined

- Słowo null reprezentuje wartość specjalną używaną do zaznaczenia braku konkretnej wartości (jak null w Javie).
- Słowo undefined również oznacza brak wartości.
  - jest wartością niezainicjowanej zmiennej,
  - jest zwracane w sytuacji, gdy nie zostanie odszukana wartość odnosząca w tablicy,
  - jest zwracane przez funkcję, która nie ma określonego typu wartości zwracanych.
- Wartość undefined jest reprezentowana przez własność obiektu globalnego.
- Poza opisaną różnicą oba słowa mogą być używane zamiennie.

```
null == undefined; //true
null === undefined; //false
```

Obiekty globalne

- Obiekt globalny jest obiektem dostarczającym własności dostępne dla całego programu JavaScript.
- W czasie startu interpretera są tworzone własności obiektu globalnego:
  - globalne właściwości takie jak undefined, Infinity, NaN.
  - funkcje globalne - np. isNan(), parseInt().
  - konstruktory np. Date(), RegExp(), String() itd.
  - obiekty globalne jak Math i JSON.
- W kodzie najwyższego poziomu, który nie jest częścią funkcji, można użyć słowa kluczowego this do odwołania do obiektu globalnego.
- Dla aplikacji klienckich obiekt Window dostarcza obiektu globalnego object dla kodu JavaScript zawartego w oknie przeglądarki.

Zmienne

- Nazwa identyfikatora zmiennej lub funkcji musi się zaczynać od liter (w tym Unicode), podkreślenia lub znaku dolara, po którym może nastąpić sekwencja liter, cyfr, znaków podkreślenia lub dolara.
- Słowa kluczowe: break, case, catch, continue, default, debugger, delete, do, else, false, finally for, function, if, in, instanceof, new, null, return, switch, this, throw, true, try, typeof var, void, while.
- Słowa zarezerwowane: class, const, enum, export, extends, import, super.
- W trybie strict są również zarezerwowane: implements, interface, let, package, private, protected, public, static, yield.
- Niedozwolone jest używanie identyfikatorów: arguments i eval.
- ECMAScript 3 rezerwuje wszystkie słowa kluczowe Javy.

Deklaracje zmiennych

- Zmienne przed użyciem powinny być zadeklarowane.
- Do deklaracji zmiennej używa się słowa kluczowego `var`, deklaracja może być połączona z inicjowaniem zmiennej.

```
var x;
var i, j;
var y = "kot";
```

- Zmienne mogą przechowywać wartości różnych typów (nie jest do nich przypisany określony typ po definicji).

```
var x = 1.0;
x = "pies";
```

Zasięg widoczności zmiennych I

- Zasięg widoczności zmiennej zależy od miejsca definicji:
  - obiekty globalne są widoczne w całym programie
  - obiekty zdefiniowane w funkcjach (lokalne) są widoczne jedynie w ciele funkcji, w której zostały umieszczone.
- Zmienne są widoczne nawet przed swoją deklaracją, są przytwierdzone do początku funkcji (ang. *hoisting*).
- Funkcje mogą być zagnieżdżone - każda funkcja ma własny zasięg lokalny.

Zasięg widoczności zmiennych II

- JavaScript nie posiada zasięgu blokowego (w przeciwieństwie do C++)

```
var k = 0;
if(i == 10) {
   var m = 154;
   for(var l = 0; l < 5; ++l) {
      //...
   }
   //l nadal widoczne
}
//m zdefiniowane, ale nie musi być zainicjowane.
```

Wyrażenia i operatory I

- Operatory arytmetyczne:
  - jednoargumentowe operatory $+$, $-$, $++$, $--$,
  - mnożenie $*$,
  - dzielenie $/$,
  - dzielenie modulo % (działa także dla liczb zmiennoprzecinkowych),
  - dodawanie $+$ (i operator konkatenacji),
  - odejmowanie $-$,
- Operatory bitowe:
  - bitowa koniunkcja &,
  - bitowa alternatywa |,
  - bitowa alternatywa wyłączająca (xor) ^,
  - bitowe zaprzeczenie ~,
  - przesunięcie bitowe w lewo <<,
  - przesunięcie bitowe w prawo ze znakiem >> (bity z lewej strony są wypełniane bitem znaku),
  - przesunięcie bitowe w prawo >>> (bity z lewej strony zawsze zerowe),

Wyrażenia i operatory II

- Operatory relacyjne:
  - równość i nierówność - operatory ==, ===, !=, !==.
    - Operator === jest operatorem identyczności, działa jak ==, ale bardziej rygorystycznie oprócz sprawdzania wartości operandów sprawdzany jest również ich typ, nie są wykonywane żadne konwersje typów.

```
"1" == true //true
"1" === true //false
1.0 == 1 //true
1.0 === 1 //true - ten sam typ Number
```

  - Operatory porównania - <, >, <=, >=

Wyrażenia i operatory III

- Operator `in` - zwraca prawdę w wtedy i tylko wtedy, gdy wartość zapisana po lewej stronie jest nazwą właściwości obiektu zapisanego po prawej stronie (przy czym wartość stojąca po lewej stronie musi być łańcuchem znaków lub dać się do tego łańcucha znaków skonwertować).

```
var obj = {a : 12};
"a" in obj; //true
"x" in obj; //false
```

- Operator `instanceof` - zwraca prawdę wtedy i tylko wtedy gdy obiekt stojący po lewej stronie operatora jest instancją typu zapisanego po prawej stronie:

Wyrażenia i operatory IV

```
var data = new Date(2014, 1, 1);
data instanceof Date; //true
data instanceof Number;//false
data instanceof Object;//true
```

Wyrażenia i operatory V

- Operatory logiczne:
  - koniunkcja - &&,
  - alternatywa - ||,
  - zaprzeczenie - !.
- Operatory przypisania - = i formy skrótowe (np. *=, +=, -= itd).

```
x = x + 4;
//jest równoważne
x += 4;
```

- Operator warunkowy ?:

```
arg > 0 ? 1 : -1;
```

Wyrażenia i operatory VI

- Operator `typeof` - operator jednoargumentowy, zwraca łańcuch znaków reprezentujący typ operandu.

```
typeof 2; //"number"
typeof "X"; // "string"
typeof null; //"object"
typeof undefined; //"undefined"
typeof true; //"boolean"
```

- Operator `delete` - powoduje usuwanie własności obiektu lub elementu tablicy.

```
var obj = {a : 1, b: 2};
delete obj.a;
var tab = [3,4,5];
delete tab[2];
```

Wyrażenia i operatory VII

- Operator `void` - jednoargumentowy, niezależnie od operandu zawsze zwraca wartość undefined - użyteczny, gdy należy pominąć przy wyświetlaniu wartości zwracane - na przykład bez wyświetlania w oknie przeglądarki.
- Operator przecinka - działanie takie jak w C++ - wartością zwracaną jest wartość wyrażenia z prawej strony, rzadko występuje samodzielnie - najczęściej używany w pętli for.

```
//Wartością wyrażenia zapisanego poniżej jest 3
x = 1, y = 2, z = 3;
```

Instrukcje I

- Instrukcja warunkowa - działanie takie jak w C++ i Javie - wartościowane wyrażenie w nawiasach okrągłych jeżeli jest prawdziwe to wykonywana jest instrukcja zapisana po wyrażeniu, jeżeli jest fraza else to jest wykonywana w sytuacji, gdy wyrażenie w nawiasach okrągłych jest wartościowane do fałszu.

```
if(x > 0)
    x = -x;
if(y > 0) {
  return y;
}
else {
  return -y;
}
```

Instrukcje II

- Instrukcja wyboru - składniowo podobne do instrukcji z C++ i Javy,
  - Wyrażenia etykiet są wartościowane w czasie uruchomienia.
  - Etykieta może być wyrażeniem dowolnego typu.

```
switch(dzien) {
    case "poniedziałek":
        return 1;
    case "wtorek"
        return 2;
    //...
    default:
        return -1;
}
```

Instrukcje III

- Pętla while - działanie takie jak w C++, dopóki wyrażenie zapisane w nawiasach okrągłych zwraca prawdę wykonywane jest ciało pętli:

```
var i = 0;
while(i < 100) {
    ++i;
}
```

- Pętla do while - działanie takie jak w C++, dopóki wyrażenie zapisane w nawiasach okrągłych zwraca prawdę wykonywane jest ciało pętli, pętla wykonywana jest przynajmniej raz:

```
var i = 0;
do {
    ++i;
}while(i < 100)
```

Instrukcje IV

- Pętla for - działanie takie jak w C++,
  - Najpierw wykonywana jest część inicjalizacyjna, potem wykonywany jest test, ciało pętli jeżeli test jest wartościowany do prawdy, na końcu część związana z modyfikacją zmiennych (np. liczników).
  - Iteracje są wykonywane aż do momentu, gdy test zwróci fałsz (lub pętla zostanie zakończona break, return).

```
for(var i = 0; i < 100; ++i) {
  //...
}
```

- Pętla for in = używana do przechodzenia po własnościach obiektu lub elementach tablicy.

Instrukcje V

```
for(var i = 0; i < tab.length; ++i)
    tab[i] = i;
for(var i in tab)
    tab[i] = i;
```

Instrukcje VI

- Skoki - break, continue - działanie takie jak w Javie - możliwy zapis z etykietą do której ma przejść sterowanie po osiągnięciu tej instrukcji.

```
etykieta:
for(var i = 0; i < 10; ++i) {
    for(var j = 0; j < 10; ++j) {
        for(var k = 0; k < 10; ++k)
            if(tab[k] < 0) break etykieta;
    }
}
```

- Instrukcja powrotu - return.
- Zgłoszenie wyjątku - throw.

Instrukcje VII

- Obsługa sytuacji wyjątkowych - obsługa podobna jak w Javie (ale tylko jeden blok catch)
  - Kod, w którym może wystąpić wyjątek jest objęty blokiem try.
  - Wyjątki są przechwytywane w bloku catch.
  - Niezależnie od tego czy został zgłoszony jakikolwiek wyjątek wykonywany jest blok finally.

```
try {
//Normalny kod
}
catch(e) {
//Kod obsługi wyjątku
}
finally {
//Kod kończący
}
```

Funkcje I

- Definicja funkcja rozpoczyna się od słowa kluczowego function, po którym znajduje się zapisany w nawiasach okrągłych zbiór jej argumentów (rozdzielanych przecinkami) i ciało funkcji.
- Funkcje w JavaScript nie sprawdzają, ile argumentów dostają na wejściu.
- Argumenty wywołania funkcji są przechowywane w zmiennej `arguments`.
- Kolejne argumenty można uzyskać korzystając z operatora indeksowania.

Funkcje II

```
function kwadrat(x) {
    return x * x;
}
function kwadrat2 {
    return arguments[0] * arguments[0];
}

kwadrat();
kwadrat(12);
kwadrat(10,121,12);
```

Literatura

1. D.Flanagan "JavaScript: The Definitive Guide" wydanie szóste, O'Reilly Media 2011
2. N.C. Zakas "Proffesional: JavaScript for Web Developers" wydanie trzecie, John Wiley & Sons 2012

Funkcje jako obywatele pierwszego rzędu I

- Funkcje są traktowane jako każda inna wartość:
  - Można przekazywać funkcje do innych funkcji jako argumenty,
  - Można zwracać jako wynik funkcję,
  - Można definiować funkcje lokalne (zagnieżdżone),
  - Można definiować funkcje anonimowe (nienazwane).
- Funkcje przyjmujące i zwracające funkcje to funkcje wyższego rzędu.

Funkcje jako obywatele pierwszego rzędu II

```
function przyklad(fun) {
    function lokalna() {
        return fun();
    }
    return lokalna;
}

funkcja = przyklad(function(){return 2*2});
funkcja();
```

Domknięcia I

- JavaScript używa zasięgu leksykalnego (statycznego).
- Zasięg leksykalny oznacza, że gdy funkcja jest wykonywana używa zmiennych z miejsca, w którym została zdefiniowana a nie z miejsca, w którym została wywołana.
- Dla implementacji zasięgu leksykalnego konieczne jest przechowywanie środowiska, w którym została zdefiniowana funkcja (zbiór zmiennych wpływających na zachowanie funkcji).
- Połączenie obiektu funkcji z jej zasięgiem (wystarcza zbiór wiązań zmiennych wolnych) jest nazywany domknięciem (ang. *closure*).
- Wszystkie funkcje w JavaScript są domknięciami.

Domknięcia II

```
function zwierze() {
var gatunek = "orangutan";
function wewnetrzna() {
    alert(gatunek);
}

return wewnetrzna;
}
//...
var gatunek = "goryl";
zwierz = zwierze();
zwierz();
```

Domknięcia III

- Zmienne wolne są przekazywane przez referencję.
- Jeżeli funkcja zwraca inną funkcję to przy każdym wywołaniu tworzone jest inne domknięcie.
- Daje to możliwość utworzenia zmiennych pomocniczych.

```
function przyklad() {
    var liczba = 0;
    var licznik = function() {alert(++liczba);}
    return licznik;}
zm = przyklad();
//1 2
zm(); zm();
//1 - inne domknięcie
przyklad()();
```

Literały obiektowe

- Literałem obiektowym jest nazywana lista rozdzielonych przecinkami par (nazwa właściwości, wartość) umieszczona w nawiasach klamrowych.
- Nazwami właściwości są identyfikatory lub łańcuchy znaków.
- Wartością w parze są dowolne wyrażenia JavaScript.

```
var pusty = {};
var prostokat = {a:1.0,b:2.0};
var autor = {
    imie : "Jan", "drugie imie": "Tadeusz",
    nazwisko: "Moziński",
    adres : {
        miejscowosc : "Warszawa"
    }
};
```

Tworzenie obiektów z wykorzystaniem new

- Instrukcja new tworzy i inicjuje nowy obiekt.
- Po słowie kluczowym new musi wystąpić wywołanie funkcji - konstruktora.
- Dla typów wbudowanych:

```
var obiekt = new Object();
var tablica = new Array();
var data = new Date();
var wr = new RegExp("a+");
```

Tworzenie obiektów z wykorzystaniem Object.create()

- ECMAScript 5 definiuje do tworzenia obiektów metodę `Object.create()`.
- Metoda ta tworzy nowy obiekt używając swojego parametru jako prototypu tego obiektu.
- Metoda create umożliwia przekazywanie drugiego, opcjonalnego parametru, który opisuje właściwości nowego obiektu.

```
//obj odziedziczy właściwości a i b
var obj = Object.create({a:1, b:2});
```

Programowanie obiektowe

- Wpływ na definicję nowego typu w JavaScript mają:
  - konstruktor - obiekt funkcji, który definiuje nazwę dla nowej klasy, właściwości dodane w konstruktorze są polami i metodami (w przypadku podstawiania funkcji do właściwości) klasy.
  - prototyp - część obiektu, która jest dziedziczona przez wszystkie obiekty klasy,
  - obiekt instancji - jest instancją nowego typu, właściwości przypisane do obiektu po jego utworzeniu nie są współdzielone z innymi obiektami.

Konstruktor I

- W terminologii JavaScript konstruktor to funkcja, która jest używana do inicjowania nowego obiektu.
- Konstruktor powinien być wywoływany z użyciem słowa kluczowego new (wywołany bez new utworzy właściwość obiektu globalnego).
- Każda funkcja JavaScript może być użyta jako konstruktor (poza funkcjami zwracanymi przez metodę Function.bind() - ECMAScript 5).
- Właściwość prototype konstruktora jest używana jako prototyp nowego obiektu - wszystkie obiekty są tworzone z wykorzystaniem jednego konstruktora i mają w ten sposób jeden typ.

Konstruktor II

```
function Punkt(x,y) {
    this.x = x;
    this.y = y;
    this.odl = function() {
        return Math.sqrt(this.x * this.x + this.y *
            this.y);
    }
}
//Tworzenie instancji
p1 = new Punkt(1,2)
//p2 - undefined - tylko wywołanie funkcji
p2 = Punkt(3,4)
```

Prototypy I

- Każdy obiekt JavaScript posiada dwie części:
  - wartość określającą jego zawartość,
  - prototyp.
- Wartość obiektu jest tworzona na podstawie prototypu.
- Każdy literał obiektowy posiada ten sam prototyp - `Object.prototype`.
- Prototyp może być określany z wykorzystaniem konstruktorów.
- Dostęp do prototypu - `Object.getPrototypeOf(obj)`
- Dodatkowo dostęp do prototypu (poza IE) jest możliwy z wykorzystaniem `__proto__`.

Prototypy II

```
var Prostokat = function(a, b) {
    this.a = a;
    this.b = b;
}

Prostokat.prototype.obwod = function() {
    return 2 * this.a + 2 * this.b;
}

var p = new Prostokat(10,20);
p.obwod();
```

Prototypy III

- Mechanizm tworzenia typów w oparciu o prototypy jest dynamiczny.
- Obiekt dziedziczy właściwości prototypu, nawet w sytuacji, gdy prototyp zostanie zmieniony po utworzeniu obiektu.
- Umożliwia to rozszerzanie typów poprzez dodawanie nowych metod.
- Przedstawiony mechanizm działa również dla typów wbudowanych - można rozszerzać liczby, łańcuchy, tablice itd.

Prototypy IV

```
Number.prototype.przeciwna = function() {
    return -this;
}
var x = -1;
x.przeciwna();
//Usuwa białe znaki z lewej strony
String.prototype.ltrim = function () {
    return this.replace(/^\s+/, '');
}
//x - "X    "
var x = "␣␣X␣␣";
x.ltrim();
```

Usuwanie właściwości I

- Do usuwania właściwości z obiektu służy operator `delete`.
- Operand delete musi być wyrażeniem służącym do dostępu do właściwości.
- Operator delete zwraca wartość prawdy, gdy usunięcie zakończyło się sukcesem lub nie spowodowało żadnego efektu.
- Można usuwać tylko właściwości własne obiektu.
- Operator delete zwraca wartość fałszu, gdy operacja usunięcia właściwości jest niemożliwa do przeprowadzenia.

Usuwanie właściwości II

```
obj = {x : "A", y : 12};
//Prawda
delete obj.x;
//Prawda
delete obj.x;
//Prawda
delete obj.z;
//fałsz
delete Object.prototype
```

Metody porównujące

- Operator równości JavaScript porównuje obiekty przez referencję a nie przez ich zawartość - obiekty są równe wtedy i tylko wtedy, gdy są tymi samymi obiektami.
- Porównywanie wartości poszczególnych składowych jest możliwe poprzez zdefiniowanie i późniejsze wykorzystanie metod.

```
Punkt.prototype.equals = function(obj) {
    if(obj == null) || (obj.constructor !== Punkt)
        return false;
    return this.x == obj.x && this.y == obj.y;
}
```

Właściwości prywatne i uprzywilejowane I

- Publiczne właściwości są deklarowane wewnątrz metod obiektu w postaci `this.nazwa`.
- Publiczne metody są definiowane z wykorzystaniem prototypu:

```
Klasa.prototype.metoda = function() {...}
```

- Prywatne zmienne są deklarowane z wykorzystaniem `var`, mogą być osiągalne jedynie z wykorzystaniem funkcji prywatnych i uprzywilejowanych (nie każda jednak taka funkcja ma dostęp - wykorzystanie domknięcia).
- Prywatne funkcje są definiowane wewnątrz konstruktora typu lub za pomocą konstrukcji:

```
var funkcja = function() {...}
```

- Właściwości statyczne są definiowane jako `Klasa.nazwa = expr;`.

Właściwości prywatne i uprzywilejowane II

```
function Kolo(r, x0, y0, kolor) {
//Pola prywatne
    var _x0 = x0;
    var _y0 = y0;
    var _r = r;
//Metoda uprzywilejowana
    this.getR = function() {
        return _r;
    }
//Pole publiczne
    this.kolor = kolor;
}
```

Właściwości prywatne i uprzywilejowane III

```
//Metoda publiczna
//Konieczne wykorzystanie metody uprzywilejowanej
//przy dostępie do zmiennych prywatnych
Kolo.prototype.pole = function() {
  return Math.PI * this.getR() * this.getR();
}
//Atrybut statyczny
Kolo.liczba = 0
kolo = new Kolo(1,2,3,"czerwony");
//Atrybut instancji
kolo.m = 43;
```

Instrukcja with I

- Dostęp do atrybutów obiektów może być skracany z wykorzystaniem instrukcji wiążącej with.
- Zazwyczaj do dostępu do składowej używa się nazwy obiektu, operatora dostępu do składowej i składową:

```
obj.x = 12;
obj.y = 10;
```

- Można jednak użyc with:

```
with(obj) {
x = 12;
y = 10;
}
```

Instrukcja with II

- W praktyce with jest kłopotliwe i może prowadzić do niejednoznaczności:

```
with(obj) {
a = b;
}
```

- Zapisaną powyżej instrukcję można interpretować jako:

```
obj.a = b;
obj.a = obj.b;
a = obj.b;
a = b;
```

- Instrukcja zaburza powiązania nazw zmiennych - spowolnienie programów.

Dziedziczenie I

- Dziedziczenie jest w JavaScript najczęściej z użyciem łańcuchowego łączenia prototypów:

```
//X <- Y <- Z
function X() {
    this.x = 10;
    this.f1 = function() {
        return "X()";    }
}
function Y() {
    this.y = 12;
    this.f2 = function() {
        return "Y()";    }
}
```

Dziedziczenie II

```
function Z() {
    this.z = 14;
    this.f3 = function() {
        return "Z()";
    }
}
//Implementacja dziedziczenia
Y.prototype = new X();
Z.prototype = new Y();
```

Tablice

- W JavaScript można przechowywać w tablicy wartości różnych typów.
- Tablice w JavaScript są dynamiczne - mogą zwiększać i zmniejszać swój rozmiar w trakcie działania programu.
- Indeksy w JavaScript są oparte na 32 bitowych liczbach.
- Indeksy nie muszą być ciągłe (jest możliwość tworzenia tzw. tablic rzadkich).
- W JavaScript tablice można traktować jako specjalny rodzaj obiektu, których właściwości są liczbami - dostęp do elementów jest jednak szybszy niż w przypadku właściwości obiektów.

Tworzenie tablic I

- Literały tablicowe - elementy rozdzielane przecinkami zapisane w nawiasach kwadratowych.

```
var pusta = [];
var tab1 = [1,2,3,4];
var tab2 = [1,false, 1.1, "x"];
var tab3 = [[1,2.0],12];
var tab4 = [{a:1,b:2},[1,2]];
var tab5 = [1,,2];
//Rozmiar tab6 3 a nie 4 - dopuszczalny jest
//zapis dodatkowego nadmiarowego przecinka
var tab6 = [,,,];
```

Tworzenie tablic II

- Tworzenie tablic z wykorzystaniem konstruktora tablicowego
  - tablica pusta:

```
var tab1 = new Array();
```

  - tablica o określonym rozmiarze - rezerwuje tylko pamięć nie są tworzone elementy ani właściwości. Rozmiarem może być zmienna:

```
var n = 10;
var tab2 = new Array(n);
```

  - jawne przekazanie elementów do konstruktora (musi być więcej niż jeden element):

```
var tab3 = new Array(1,2,3,4,5);
```

Dostęp do elementów tablic I

- Dostęp do elementów tablicy jest realizowany z wykorzystaniem operatora indeksowania.
- Dostęp do nieistniejących indeksów nie skutkuje błędem, lecz zwraca wartość undefined;

```
var tab = [1,2];
tab[10]
tab[-2]
```

- Zapis do elementu pod nieistniejącym indeksem skutkuje powstaniem nowego elementu w tablicy.

```
var tab = [1,2];
tab[10] = 100;
```

Dostęp do elementów tablic II

- Indeksem są z reguły dodatnie liczby całkowite - ale są również dopuszczalne inne wartości:
  - łańcuchy znaków - jeżeli da się go skonwertować do liczby to jest używany jako indeks, jeżeli nie jest tworzona właściwość o takiej nazwie.
  - liczba ujemna - tworzona jest właściwość o nazwie będącej łańcuchem przechowującym znakową reprezentację tej liczby.
  - liczba rzeczywista - jeżeli nie posiada części ułamkowej to jest traktowana jako indeks, w przeciwnym razie zasady są takie same jak dla liczby ujemnej.

Dostęp do elementów tablic III

```
tab = [1,2];
//tworzona jest właściwość "1.2"
tab[1.2] = 10;
//element o indeksie 100
tab["100"] = 12;
//tworzona jest właściwość "-1.2"
tab[-1.2] = 120;
//tworzona jest właściwość "-1"
tab[-1] = 12;
//dostęp do drugiego elementu
tab[1.0] = 10;
```

Rozmiar tablicy I

- Liczbę elementów w tablicy gęstej można poznać za pomocą właściwości `length`.
- W przypadku tablic rzadkich zwróci o jeden większy maksymalny możliwy indeks.
- Właściwość daje się ustawiać
  - w przypadku mniejszych wartości od bieżącego rozmiaru nadmiarowe elementy są usuwane,
  - w przypadku większych wartości od bieżącego rozmiaru zmienia się tylko wartość właściwości, nie są dostawiane żadne dodatkowe elementy.

Rozmiar tablicy II

```
//tab.length == 3
tab = [1,2,3]
//length wynosi 2, ale nie ma żadnych elementów
tab2 = [,,]
//length wynosi 5, ale nie ma żadnych elementów
tab3 = new Array(5);
//length po wykonaniu tych instrukcji wynosi 11,
//Jest tylko jeden element
tab4 = []
tab4[10] = 12;
```

Rozmiar tablicy III

```
tab5 = [1,2,3,4];
//Nadal tylko 4 elementy
tab5.length = 10;
//Teraz już tylko 3
tab5.length = 3;
//Teraz tablica pusta
tab5.length = 0;
//Odpowiednik tab5 = new Array(20);
//Nie są tworzone elementy
tab5.length = 20;
```

Iterowanie elementów tablicy I

- Najczęściej przechodzi się po elementach tablicy z wykorzystaniem pętli for w postaci:

```
var tab = [1,2,3,4];
for(var i = 0; i < tab.length; ++i) {
    console.log(tab[i]);
}
//lub wersja szybsza
var tab = [1,2,3,4];
var rozmiar = tab.length;
for(var i = 0; i < rozmiar; ++i) {
    console.log(tab[i]);
}
```

Iterowanie elementów tablicy II

- W sytuacji, gdy konieczne jest pominięcie elementów niezdefiniowanych lub nieistniejących, należy dodać dodatkowy test na te wartości (w komentarzu pominięcie również wartości null):

```
var tab = [1,,3,4];
for(var i = 0; i < tab.length; ++i) {
// if(!tab[i]) continue;
    if(tab[i] === undefined) continue;
    console.log(tab[i]);
}
```

Iterowanie elementów tablicy III

- Należy unikać pętli for - in - wynika to z tego, że będą również wyświetlane wartości odziedziczone po `Array.prototype`.
- Można je jednak odfiltrować za pomocą:

```
for(var el in tab) {
    if(!tab.hasOwnProperty(el))
        continue;
    console.log(el);
}
```

- Nie ma gwarancji, w jakim porządku będą wyświetlane elementy!

Metody tablic I

- `concat()` - tworzy i zwraca nową tablicę złożoną z połączonej pierwotnej tablicy i argumentów. Spłaszcza najbardziej zewnętrzne tablice przekazywane w ten sposób (ich elementy są wstawiane jako kolejne elementy nowej tablicy).

```
var tab = [1,2,3];
//Zwraca [1,2,1,2]
tab.concat(1,2);
//Zwraca [1,2,1,2]
tab.concat([1],[2]);
//Zwraca [1,2,1,[2]]
tab.concat([1],[[2]]);
```

Metody tablic II

- `indexOf()` i `lastIndexOf()` zwracają indeks pierwszego (lub ostatniego) wystąpienia określonej parametrem wartości (lub -1, jeżeli elementu o takiej wartości nie ma w tablicy):

```
var tab = [1,2,3,4];
tab.indexOf(2);//1
tab.indexOf(10);//-1
```

- `join()` - łączy elementy tablicy w łańcuch znaków (opcjonalny parametr - separator):

```
var tab = [1,2,3,4];
tab.join(); //'1,2,3,4'
tab.join(" "); //'1 2 3 4'
```

Metody tablic III

- `slice()` - tworzy wycinek tablicy - tablicę zawierającą elementy podane indeksami.
- `toString()` - tworzy łańcuch reprezentujący tablicę i jej elementy.
- `pop()` - usuwa ostatni element tablicy i zwraca go.
- `push()` - dodaje nowy element (elementy) na koniec tablicy i zwraca nową długość tablicy.
- `reverse()` - odwraca kolejność elementów tablicy.
- `sort()` - sortuje zawartość tablicy.

Metody tablic IV

- `every()` - zwraca prawdę, jeżeli wszystkie elementy spełniają predykat.
- `filter()` - tworzy nową tablicę zawierająca elementy, które pasują do filtra. parametr funkcję.
- `shift()` - usuwa i zwraca pierwszy element tablicy,
- `some()` - zwraca prawdę, jeżeli choć jeden element spełnia predykat.
- `unshift()` - dodaje jeden lub więcej elementów na początek tablicy, zwraca nową długość tablicy.

Metody tablic V

- `forEach()` - wywołuje określoną parametrem funkcję na każdym elemencie tablicy:

```
var tab = [1,2,3];
//tab == [1,4,9]
tab.forEach(function(x,i,t) {t[i] = x * x;});
var sum = 0
[1,2,3].forEach(function(x) {sum += x}
```

- `map()` - tworzy nową tablicę zawierającą elementy, które uzyskano aplikując przekazywaną jako argument funkcję do każdego elementu pierwotnej kolekcji:

```
var tab1 = [1,2,3];
//tab2 ==  [1,4,9]
var tab2 = tab1.map(function(x) { return x * x;}
```

Metody tablic VI

- `reduce()` i `reduceRight()` - łączą kolejne elementy tablicy z użyciem przekazywanej funkcji - tworzy się w ten sposób pojedynczą wartość. Funkcja reduce() łączy elementy do najniższego indeksu, zaś reduceRight() od najwyższego.

```
var tab = [1,2,3,4];
//suma  == 10
var suma = tab.reduce(function(x,y) {return x +
    y;}, 0);
//suma2 == 10
var suma2 = tab.reduceRight(function(x,y) {
    return x + y;},0);
```

Metody tablic VII

```
//t1 ==  [1,2,3,4]
var t1 = tab.reduce(function(x,y) {return x.
    concat(y);}, []);
//t2 == [4,3,2,1]
var t2 = tab.reduceRight(function(x,y) {return x
    .concat(y);},[]);
```

Wyrażenia regularne I

- Wyrażenie regularne jest obiektem opisującym ciąg znaków.
- Wyrażenia regularne są użyteczne przy wyszukiwaniu ciągów znaków lub zamiany ciągów znaków
- W JavaScript wyrażenia regularne opisuje typ `RegExp`.
- JavaScript implementuje składnię wyrażeń regularnych Perla (większość elementów została przejęta z tego języka).

Wyrażenia regularne II

- Wyrażenia regularne definiuje się korzystając z ukośnika (slash /) lub z konstruktora typu:

```
var wzorzec = /^A/;
var wzorzec2 = new RegExp("^A");
```

- Dopuszczalne jest użycie flag:
  - i - dopasowanie niezależne od wielkości liter,
  - g - wyszukanie wszystkich dopasowań a nie tylko jednego,
  - m - tryb wieloliniowy.

Klasy znaków

Symbol	Co dopasowuje
`[...]`	Dowolny znak w nawiasach
`[^...]`	Dowolny znak niewymieniony w nawiasach
`.`	Dowolny znak (oprócz znaku nowego wiersza)
`\w`	Znak ASCII (cyfra, litera, znak podkreślenia)
`\W`	Znak, który nie jest znakiem ASCII.
`\s`	Biały znaku Unicode.
`\S`	Dowolny znak, który nie jest białym znakiem Unicode
`\d`	Cyfra ASCII
`\D`	Dowolny znak różny od cyfry ASCII.

Powtórzenia

Symbol	Znaczenie
`{n,m}`	Dopasowanie co najmniej n razy, lecz nie więcej niż m
`{n,}`	Dopasowanie n lub więcej razy
`{n}`	Dopasowanie dokładnie n razy
`?`	Opcjonalne dopasowanie
`+`	Dopasowanie jeden lub więcej razy
`*`	Dopasowanie zero lub więcej razy.

Grupowanie i alternacja

Symbol	Znaczenie
`|`	Dopasowanie do prawego lub lewego podwyrażenia
`(...)`	Grupowanie i numerowanie - zapamiętana grupa może być później użyta
`(?:...)`	Tylko grupowanie
`\num`	Dopasowanie do tych samych znaków jak w przypadku grupy num

Dopasowanie do określonej pozycji

Symbol	Znaczenie
`^`	Dopasowanie do początku linii lub łańcucha
`$`	Dopasowanie do końca linii lub łańcucha
`\b`	Dopasowanie do granicy słowa
`\B`	Dopasowanie do pozycji, która nie jest granicą słowa
`(?= p)`	Pozytywne przeglądanie do przodu - wymagane dopasowanie do ciągu, ale ciąg nie jest konsumowany
`(?! p)`	Negatywne przeglądanie do przodu - wymagany brak dopasowania do ciągu, ciąg nie jest konsumowany

Przykłady I

```
/\d{1,4}/  //"1" "12" "1234"
/\d{4}/  //"1234"
/\s+domek\s*/ //"   domek" " domek  "
// "for" ale nie "forma", "bufor", "aforyzm"
/\bfor\b|\bif\b|\bwhile\b/
// 'a' "b"
/(['"])[^'"]*\1/
```

Metody łańcuchów współpracujące z wyrażeniami regularnymi I

- Z wyrażeniami regularnymi współpracują metody łańcuchów znakowych - `search()`, `replace()`, `match()` i `split()`.
- Metoda `search()` umożliwia odszukanie indeksu, od którego rozpoczyna się wyraz dopasowujący do danego wyrażenia regularnego (lub -1, jeżeli takiego nie znajdzie):

```
"MOLEK".search(/[AO]lek/i); //1
```

Metody łańcuchów współpracujące z wyrażeniami regularnymi II

- Metoda `replace()` zwróci łańcuch, w którym zastąpi wyraz dopasowujący do określonego wyrażenia regularnego ciągiem przekazanym jako drugi argument (dozowolone jest użycie znaku $, za którym następuje cyfra):

```
var tekst = "a b c d e";
//Zwróci "a, b, c, d, e"
tekst.replace(/(\s+)/g, ',$1);
```

Metody łańcuchów współpracujące z wyrażeniami regularnymi III

- Metoda `match()` - umożliwia przekazanie jako argument wyrażenia regularnego i zwraca tablicę zawierającą wyniki dopasowania (rozróżnia globalne i nieglobalne przeszukiwanie).

```
tekst = "abc@gmail.com"
//[ 'abc@gmail.com', 'abc', 'gmail.com',
// index: 0, input: 'abc@gmail.com' ]
tekst.match(/(\w+)@(\S+)/)
//[ 'abc@gmail.com' ]
tekst.match(/(\w+)@(\S+)/g)
```

Metody łańcuchów współpracujące z wyrażeniami regularnymi IV

- Metoda `split()` umożliwia podział łańcucha znaków względem granicy określonej wyrażeniem regularnym (zwraca tablicę łańcuchów znaków):

```
var tekst = "1234 456   789 11223";
//[ '1234', '456', '789', '11223' ]
tekst.split(/\s+/);
```

Metody wyrażeń regularnych I

- Typ RegExp definiuje dwie metody pozwalające na wykonywanie operacji z wykorzystaniem wzorców - `exec()`, `test()`.
- Działanie metody `exec()` - działa podobnie jak `match()`, ale zwraca tablicę znaków o takiej samej postaci - ustawiane są właściwości `index` i `lastIndex`.
  - W przypadku wyszukiwania globalnego, w kolejnej iteracji dopasowanie zaczyna się od pozycji `lastIndex`.
- Metoda `test()` pozwala na sprawdzenie czy dane wyrażenie regularne dopasowuje do jakiegoś fragmentu łańcucha znaków (zwraca wartość logiczną).

Metody wyrażeń regularnych II

```
var tekst = "JavaScript␣jest␣bardziej␣zakręcony\
od␣języka␣Java";
var wzorzec = /Java/g;
var wynik;
while((wynik = wzorzec.exec(tekst)) != null) {
    console.log(wynik[0]);
}

wzorzec.test(tekst); //true
```

Nowości w JavaScript 6 I

- Zasięg blokowy
  - ECMAScript 5

```
var tab = [];
for (var i = 0; i < 5; ++i) {
   tab[i] = function () {
      console.log("Numer" + i);
   };
}
//Numer 4
tab[2]();
```

Nowości w JavaScript 6 II

- ECMAScript 6

```
var tab = [];
for (var i = 0; i < 5; ++i) {
  let nr = i;
  tab[i] = function () {
    console.log("Numer" + nr);
  };
}
//Numer 2
tab[2]();
```

Nowości w JavaScript 6 III

- Stałe

```
const EULER = 0.572
//Nie powiedzie się
EULER = 0.5
```

- Parametry domyślne funkcji

```
function fun(x,y) {
    if(x === undefined) x = 1;
    if(y === undefined) y = 2;
    return x + y;
}
```

Nowości w JavaScript 6 IV

```
function fun(x,y) {
    x = x || 1;
    y = y || 2;
}

function fun(x = 1, y = 2) {
    return x + y;
}
```

Nowości w JavaScript 6 V

- Nowy typ Symbol - wartości, które program może tworzyć i wykorzystywać jako klucze bez ryzyka wystąpienia kolizji - wartości są unikalne.

```
var s1 = Symbol();
var s2 = Symbol('x');
var s3 = Symbol('y');
typeof Symbol(); // "symbol"
Symbol('x') === Symbol('x'); //false
var wlasciwosc = Symbol('nowaWlasciwosc');
var obj = {};
obj[wlasciwosc] = 10;
```

Nowości w JavaScript 6 VI

- Operator rozwijania (spread)

```
function f(x, y, z) { }
var args = [0, 1, 2];
//ES5
f.apply(null, args);
//ES6
f(...args);
```

Nowości w JavaScript 6 VII

```
var tab = [1, 2, 3, 4];
// Zawartość tablicy tab staje się elementami
   tab2
var tab2 = [0, ...tab, 5];
```

Nowości w JavaScript 6 VIII

- Łańcuchy szablonowe

```
//Odwrotny apostrof
`Bez interpolacji zmiennej`
`Zajmuje
 wiele linii`
var imie = "Janusz"
//Z interpolacją zmiennej
`Witaj ${imie}`
```

Nowości w JavaScript 6 IX

- Iteratory i for..of

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur };
      }
    }
  }
}
```

Nowości w JavaScript 6 X

```
for (let n of fibonacci) {
  if (n > 1000)
    break;
  console.log(n);
}
```

Nowości w JavaScript 6 XI

- Generatory - to funkcje, które zwracają iterator. Funkcja generatorowa zatrzymuje wykonanie po każdej instrukcji yield.

```
function *utworz(tab) {
    for (let i = 0; i < tab.length; ++i) {
        yield tab[i];
    }
}

let itr = utworz([5, 4]);

console.log(it.next());
console.log(it.next());

console.log(it.next());
```

Nowości w JavaScript 6 XII

- Skrócony zapis funkcji anonimowych (ang. arrow function)

```
var kwadrat = x => x * x;
var prostokat = (a, b) => a * b;

tab.forEach(v => {
        if (v % 5 === 0)
                wynik.push(v) })
```

Nowości w JavaScript 6 XIII

- Słowniki i zbiory

```
let s = new Set();
s.add(1);
s.add(2);

let m = new Map();
m.set("nazwisko", "Nowak");
m.set("rok", 2016);

console.log(m.get("nazwisko"));
console.log(m.get("rok"));
```

Nowości w JavaScript 6 XIV

- Słabe słowniki i zbiory

```
let s = new Set(),
    klucz = {};

s.add(klucz);
console.log(s.size);
//niszczenie obiektu
klucz = null;

console.log(s.size);

//Odzyskiwanie klucza
klucz = [...s][0]
```

Nowości w JavaScript 6 XV

```
let s2 = new WeakSet(), klucz2 = {};

s2.add(klucz2);
s2.has(klucz2);

klucz2 = null;

s2.has(klucz2);
```

Nowości w JavaScript 6 XVI

- Klasy

```
class Punkt {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    dlugosc() {
        return Math.sqrt(x * x + y * y);
    }
}
```

Nowości w JavaScript 6 XVII

- Dziedziczenie

```
class Figura {
    constructor(x,y) {
        //...
    }
}
class Kwadrat extends Figura {
    constructor(x, y, a) {
        super(x,y);
        //...
    }
}
```

Nowości w JavaScript 6 XVIII

- Moduły

```
//mod1.js
 export function f(x, y) {
     return x * y
 }
 //main.js
 import * as m from "mod1"
 console.log("Wynik:" + m.f(4,5))
```

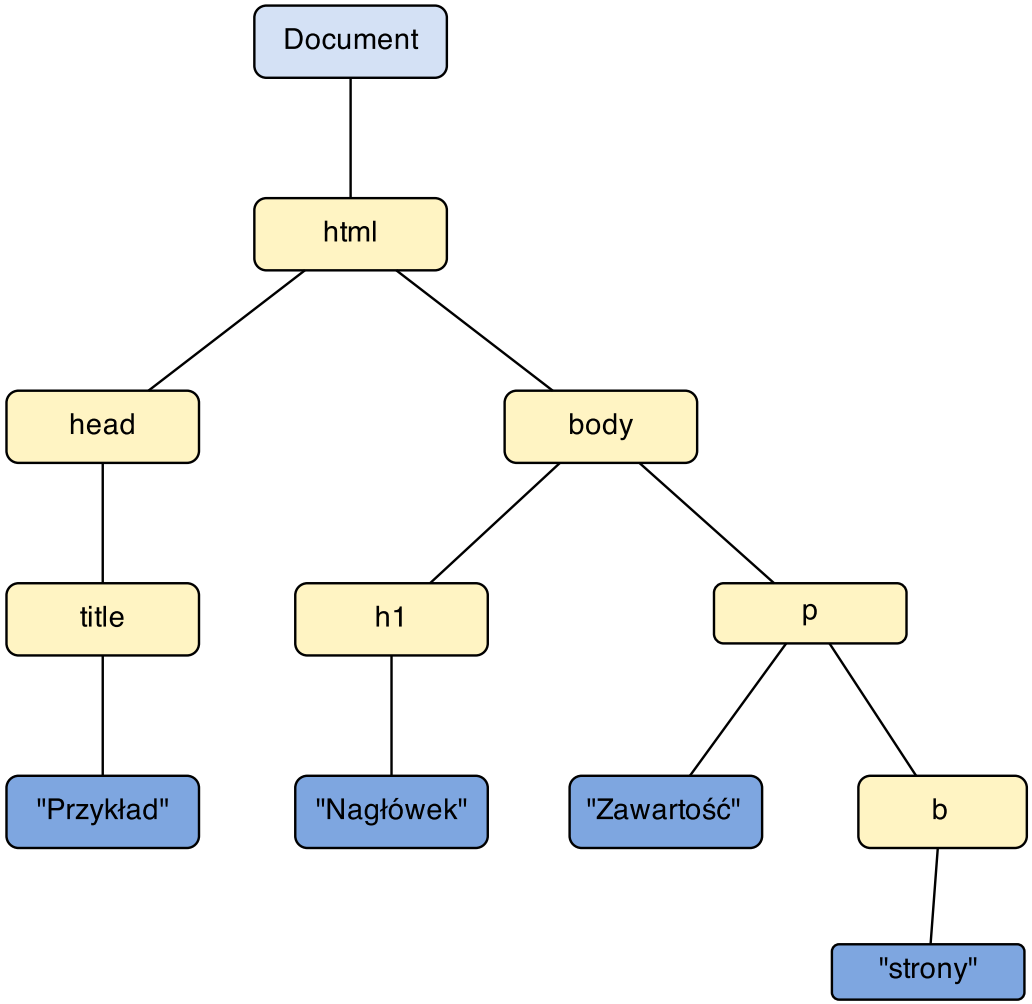
Document Object Model I

- Document Object Model (w skrócie DOM) jest podstawowym API do reprezentacji i modyfikacji zawartości dokumentów HTML i XML.
- Zagnieżdżona struktura dokumentów HTML lub XML jest reprezentowana w DOM za pomocą drzewa obiektów.
- Drzewo reprezentuje dokument w postaci węzłów przedstawiających tagi HTML (np <body>,<b>) i węzłów reprezentujących tekst.
- Korzeniem całego drzewa jest dokument jako całość.

Document Object Model II

```
<html>
    <head>
        <title>Przykład</title>
    </head>
    <body>
        <h1>Nagłówek</h1>
        <p>Zawartość <b>strony</b> </p>
    </body>
</html>
```

Document Object Model III



Rodzaje węzłów DOM

- Dokument - `Document`
  - `HTMLDocument`
- Dane znakowe - `CharacterData`
  - `Text` - tekst,
  - `Comment` - komentarz.
- Elementy - `Element`
  - `HTMLElement`
    - `HTMLHeadElement`
    - `HTMLBodyElement`
    - `HTMLTableElement`
    - ...
- Atrybuty - `Attr` - reprezentują atrybuty elementów

Atrybuty węzłów - trawersacja DOM I

Najważniejszymi atrybutami węzłów są:

- `parentNode` - zawiera węzeł rodzicielski (lub null dla węzłów, które nie mają rodzica),
- `childNodes` - tablica tylko do odczytu zawierająca reprezentacje węzłów potomnych,
- `firstChild` i `lastChild` - pierwszy i ostatni potomek węzła (lub null jeśli węzeł nie posiada dzieci),
- `nextSibling`, `previousSibling` - następne i poprzednie rodzeństwo węzła (węzły potomne tego samego węzła są rodzeństwem). Porządek jest określony występowaniem w dokumencie.

Atrybuty węzłów - trawersacja DOM II

- `nodeType` -rodzaj węzła
  - Document - 9
  - Element - 1
  - Text - 3
  - Comment - 8
  - DocumentFragment - 11
- `nodeValue` - zawartość węzła tekstowego lub komentarza (Text i Comment),
- `nodeName` - nazwa tagu elementu zapisana wielkimi literami.

Atrybuty elementów - trawersacja DOM

- Elementy przedstawiają tagi HTML - przedstawione właściwości umożliwiają przechodzenie po drzewie dokumentów bez odwiedzania węzłów tekstowych i komentarzy.
- Najważniejsze właściwości z tej grupy:
  - `children` - jest to `NodeList`, ale zawiera tylko obiekty Element.
  - `firstElementChild` i `lastElementChild` - podobne jak dla węzła, ale tylko elementy.
  - `nextElementSibling` i `previousElementSibling` - rodzeństwo, które jest elementem,
  - `childElementCount` - liczba elementów potomnych (zwraca to samo co `children.length`).

Wybór elementów dokumentu I

- Do obiektu Document można odwołać się poprzez zmienną globalną `document`.
- Przekształcanie elementów jest możliwe dopiero po uzyskaniu odpowiedniego elementu.
- Dostęp do elementów dokumentów jest możliwy na wiele sposobów:
  - za pomocą atrybutu `id`,
  - za pomocą atrybuty `name`,
  - za pomocą wyspecyfikowanej nazwy tagu,
  - za pomocą wyspecyfikowanej klasy arkusza styli,
  - za pomocą dopasowania do wyspecyfikowanego selektora CSS.

Wybór elementów dokumentu II

- Wszystkie elementy dokumentu HTML posiadają atrybut `id`.
- Wartość atrybutu id musi być unikalna wewnątrz dokumentu.
- Elementy wybiera się za pomocą funkcji `getElementById()` obiektu `Document`.

```
//Na przykład w funkcji obsługi zdarzenia
var x=document.getElementById("mojNaglowek");
alert(x.innerHTML);
//Fragment kodu HTML
<h1 id="mojNaglowek">Nagłówek</h1>
```

Wybór elementów dokumentu III

- Atrybut `name` był używany do nazywania elementów fomularzy, które były wykorzystywane przy wysyłaniu danych na serwer.
- Atrybut name przypisuje nazwę dla elementu, w przeciwieństwie jednak do id nie musi być unikalny.
- Wybór elementu na pomocą atrybutu name jest możliwy po zastosowaniu funkcji `getElementsByName()`, jest ona metodą `HTMLDocument` - jest więc specyficzna dla dokumentów HTML.
- Wartością zwracaną przez wymienioną wcześniej metodą jest obiekt `NodeList`, który zachowuje się jak tablica tylko do odczytu obiektów Element.

```
alert(document.getElementsByName("nazwa")[0].
    value);
```

Wybór elementów dokumentu IV

- Możliwy jest wybór wszystkich elementów o wyspecyfikowanym typie (lub nazwie tagu) za pomocą metody `getElementsByTagName()`.

```
var paragy = document.getElementsByTagName("p");
```

- Podobnie jak wcześniejsza metoda zwraca obiekt `NodeList`.
- Dla dokumentów HTML nie są rozróżniane przez metodę wielkości liter.
- Elementy również posiadają metodę `getElementsByTagName()`, ale w ich przypadku są zwracane elementy, które są potomkami tego elementu (są umieszczone wewnątrz).

Wybór elementów dokumentu V

- Atrybut `class` HTML is listą zawierającą identyfikatory rozdzielone spacjami.
- Atrybut pozwala na definicję zbioru powiązanych ze sobą elementów dokumentu.
- Atrybut ten jest wykorzystywany w połączeniu z arkuszami styli.
- HTML5 definiuje metodę `getElementByClassName()` umożliwiającą wybór elementów dokumentu, które mają ten sam atrybut `class`.
- Metoda zwraca obiekt `NodeList`.
- Metoda może być używana przez elementy - umożliwiają wybór elementów potomnych, które posiadają tą samą wartość atrybutu `class`.

Wybór elementów dokumentu VI

```
//Znajdź wszystkie elementy posiadające atrybut
   header
var header = document.getElementByClassName("
   header");
//Znajdź wszystkich potomków elementu o id log,
//którzy posiadają nazwę klasy error.
var log = document.getElementById("log");
var err = log.getElementByClassName("error");
```

Wybór elementów dokumentu VII

- Arkusze stylów posiadają selektory, które umożliwiają opis elementów lub zbioru elementów wewnątrz dokumentu. Np:
  - Element o id równym log - `#log`,
  - Dowolny element `<div>` - `div`,
  - Dowolny element z atrybutem error - `.error`
  - Selektory mogą być nawet łączone - `div.warning`.
- Metodą umożliwiającą wybór elementów za pomocą selektorów jest - `querySelectorAll()`.
- Metoda przyjmuje łańcuch znaków zawierający selektor CSS i zwraca `NodeList`.

```
var dops = document.querySelectorAll("div.note, div.alert");
```

Wybór elementów dokumentu VIII

- Wybór pierwszego pasującego elementu do selektora jest również możliwy za pomocą `querySelector()`.
- Jeżeli łańcuch znaków zawiera niepoprawny selektor to zwracany jest wyjątek.
- W celu dopasowania identyfikatora lub selektora, który nie jest składniowo poprawny z CSS (np zawiera dwukropek) znak musi być poprzedzony odwrotnym ukośnikiem.
- Znak odwrotnego ukośnika ma w JavaScript znaczenie specjalne, dlatego musi być wprowadzony podwójnie raz jako łańcuch JavaScript zaś drugi raz jako dla argumentu funkcji querySelector.

Wybór elementów dokumentu IX

```
<div id="foo\bar"></div>
<div id="foo:bar"></div>

<script>
  // "#fooar"
  console.log('#foo\bar');
  //Nie dopasowuje do niczego
  document.querySelector('#foo\bar')

  // "#foo\bar"
  console.log('#foo\\bar');
  // "#foo\\bar"
  console.log('#foo\\\\bar');
```

Wybór elementów dokumentu X

```
    //Dopasowuje do pierwszego diva
    document.querySelector('#foo\\\\bar');

    //Nie dopasowuje do niczego
    document.querySelector('#foo:bar');
    //Dopasowuje drugiego diva
    document.querySelector('#foo\\:bar');
</script>
```

Dostęp do atrybutów HTML elementów I

- Każdy obiekt HTMLElement, który reprezentuje elementy dokumentu HTML definiuje właściwości, które odzwierciedlają atrybuty HTML tego tagu.
- HTMLElement definiuje uniwersalne właściwości takie jak `id`.
- Są również specyficzne dla elementów atrybuty - np. dla obrazka `src`.

```
var obraz = document.getElementById("obrazek");
var obrazUrl = obraz.src;
```

Dostęp do atrybutów HTML elementów II

- Właściwości elementów, mimo że HTML nie rozróżnia wielkości liter w atrybutach, zapisuje się małymi literami (wymagania składni JavaScript).
- Atrybuty, które są słowami kluczowymi JavaScript poprzedza się przedrostkiem `html` - np dla for - htmlFor.
  - wyjątkiem dla tej reguły jest `className` dla atrybutu `class`.
- Typ właściwości zależy od tego co przechowuje najczęściej są to łańcuchy znaków (ale np. dla obsługi zdarzeń Function lub null).

Odczyt/zapis atrybutów niestandardowych I

- Typ Element definiuje także funkcje `getAttribute()` i `setAttribute()` - są one użyteczne:
  - do odczytu i ustawiania niestandardowych atrybutów HTML,
  - do odczytu i zapisu atrybutów dokumentów XML.
- Wartości atrybutów są zawsze traktowane jako łańcuchy znaków:

```
var dlg = parseInt(obraz.getAttribute("width"));
obrazek.setAttibute("class", "miniaturka");
```

- Metody te mogą być również używane dla standardowych atrybutów - nazwy są zapisywane normalnie nawet dla atrybutów będących słowami kluczowymi JavaScript.

Odczyt/zapis atrybutów niestandardowych II

- Element definuje dodatkowe dwie metody:
  - `hasAttribute()` - pozwala sprawdzić czy podany atrybut istnieje,
  - `removeAttribute()` - umożliwia usunięcie atrybutu.
- Dokumenty XML, które dołączają atrybuty z innej przestrzeni nazw używają metod `getAttributeNS()`, `setAttibuteNS()`, `hasAttributeNS()` i `removeAttributeNS()` - przejmują dwa parametry:
  - pierwszy określaja przestrzeń nazw,
  - drugi określa atrybut w tej przestrzeni nazw.

Zawartość elementu I

W zależności do potrzeb zawartość elementu (content) może być definiowana jako:

- łańcuch HTML - łańcuch zawierający tagi HTML np. `Tekst z <b>tagami</b> html`,
- czysty tekst - czysty łańcuch tekstowy, bez tagów HTML np. dla powyższego łańcucha będzie to `Tekst z tagami html`,
- węzeł typu Text - Element zawiera jako potomka tego rodzaju węzeł.

Zawartość elementu II

- Odczyt właściwości `innerHTML` elementu pozwala poznać zawartość w postaci HTML.
- Zapis tej właściwości wywołuje parser HTML przeglądarki i zastępuje zawartość elementu nowym.
- Właściwość `innerHTML` może być używana w dokumentach HTML i XML.
- Element posiada również właściwość `outerHTML` - oprócz zawartości dołączone są również otwierające i zamykające tagi elementu.

Zawartość elementu III

- Czysty tekst można uzyskać za pomocą właściwości `textContent`.
- Właściwość umożliwia uzyskanie zawartości tekstowej węzła i jego potomków (konkatenacja).
- W IE jest właściwość jest dostępna od wersji 9.

```
document.getElementsByTagName("BUTTON")[0].
    textContent;
```

Modyfikacja dokumentu

- Oprócz przechodzenia dokumentu i zmiany właściwości elementów i węzłów tekstowych możliwa jest zmiana dokumentu na poziomie indywidualnych węzłów.
- Dokument definiuje kilka metod do tworzenia obiektów elementów i tekstowych a węzeł (Node) definiuje metody do wstawiania, usuwania i zamiany węzłów w drzewie dokumentu.

Tworzenie węzłów

- Do tworzenia węzłów elementów służy metoda dokumentu `createElement()`. Przyjmuje ona tag elementu (wielkość liter nie ma znaczenia w przypadku HTML, ma zaś znaczenie w XML).

```
var elem = document.createElement('script');
```

- W podobny sposób są tworzone węzły tekstowe, do tego celu służy metoda `createTextNode()`. Przekazywany argument jest zawartością tego węzła.

```
var wzl = document.createTextNode("Zawartość");
```

- Do tworzenia komentarzy służy metoda `createComment()`.

Wstawianie węzłów

- Po utworzeniu nowego węzła należy go wstawić do dokumentu z wykorzystaniem metod węzła - `appendChild()` lub `insertBefore()`.
- Metoda appendChild() jest wywoływana dla elementu, do którego należy wstawić zawartość - wstawiony węzeł będzie ostatnim potomkiem tego elementu (lastChild).
- Metoda `insertBefore()` pobiera dwa argumenty, pierwszym jest węzeł, który będzie wstawiany, zaś drugim argumentem jest węzeł, przed którym będzie on wstawiony (metoda jest wywoływana dla rodzica tego węzła).
  - Metoda może przyjmować jako drugi argument null - zachowuje się wtedy jak appendChild().

Usuwanie i zastępowanie węzłów

- Metoda `removeChild()` usuwa węzeł z drzewa dokumentu.
- Metoda musi być wywołana w węźle rodzicielskim danego węzła:

```
node.parentNode.removeChild(node);
```

- Metoda `replaceChild()` zastępuje jeden węzeł innym.
- Metoda musi być wywoływana dla węzła rodzicielskiego węzła zastępowanego:

```
node.parentNode.replaceChild(
      document.createTextNode("Zawartość"),
                                          node);
```

Obiekt Window I

- Obiekt Window reprezentuje okno przeglądarki,odnosi się do niego z wykorzystaniem identyfikatora window.
- Obiekt Window jest obiektem globalnym.
- Obiekt Window definiuje szereg właściwości i metod pozwalających na:
  - używanie timerów,
  - zmianę lokalizacji przeglądanej strony,
  - przeglądanie historii,
  - obsługę błędów,
  - uzyskiwanie informacji o ekranie,
  - wyświetlanie prostych okien dialogowych,
  - pracę z wieloma oknami i ramkami.

Timery I

- Metody obiektu Window - `setTimeout()` i `setInterval()` pozwalają na zarejestrowanie funkcji wywoływanych raz lub wielokrotnie po określonym przedziale czasu.
- Metoda `setTimeout()` wywołuje wyszczególnioną funkcję po upływie określonej liczby milisekund. Metoda zwraca wartość, która może być wykorzystana do anulowania wykonania zaplanowanej funkcji (metoda `clearTimeout()`.
- Metoda `setInterval()` działa podobnie jak wymieniona wcześniej `setTimeout()` z wyjątkiem tego, że stale powtarza wywołanie określonej funkcji.

Timery II

```
var myVar = setInterval(mojTimer,1000);

function mojTimer()
{
    var d = new Date();
    var t = d.toLocaleTimeString();
    document.getElementById("demo").innerHTML=t;
}

function zatrzymajTimer()
{
    clearInterval(myVar);
}
```

Właściwość location I

- Właściwość `location` (obiekt `Location` odnosi się do adresu dokumentu wyświetlanego obiektu w oknie przeglądarki.
- Obiekt `location` posiada właściwości i metody:
  - `href` - przechowuje pełny adres,
  - `hash` - przechowuje część związaną z identyfikatorem fragmentu (#)
  - `host` - przechowuje nazwę hosta i port,
  - `hostname` - przechowuje adres hosta,
  - `port` - przechowuje port,
  - `search` - przechowuje fragment adresu za znakiem zapytania (część zapytania,
  - `assign()` - ładuje nowy dokument,
  - `reload()` - przeładowuje bieżący dokument,
  - `replace()` - zastępuje bieżący dokument nowym.

Historia przeglądania I

- Historię przeglądarki reprezentuje właściwość `history` obiektu Window.
- Obiekt History jest listą dokumentów i stanów dokumentów.
- Skrypty nie mają bezpośredniego dostępu do przechowywanych adresów.
- Metody `back()` i `forward()` - działają jak przycisk przeglądarki <- i ->.
- Metoda `go()` pozwala podać ile stron do przodu (lub do tyłu przy ujemnych liczbach) należy przesunąć się w historii przeglądarki.

Informacje o przeglądarce i ekranie I

- Informacje o przeglądarce przechowuje właściwość `navigator` obiektu Window.
- Obiekt typu `Navigator` posiada właściwości:
  - `appName` - przechowuje pełną nazwę przeglądarki,
  - `appVersion` - przechowuje informacje o wersji przeglądarki,
  - `userAgent` - przechowuje łańcuch jaki przeglądarka wysyła w nagłówku `USER-AGENT`.
  - `platform` - łańcuch identyfikujący system operacyjny, na którym przeglądarka jest uruchomiona.

Informacje o przeglądarce i ekranie II

- Właściwości ekranu (rozdzielczość, liczba kolorów) reprezentuje atrybut `screen` obiektu Window.
- Właściwości obiektu `Screen` reprezentujące wysokość i szerokość ekranu w pikselach to odpowiednio `height` i `height`.
- Właściwości `availWidth` i `availHeight` reprezentują wielkość ekranu pomniejszoną o piksele które zajmuje interfejs np. pasek zadań.
- Głębia koloru jest reprezentowana przez `colorDepth`.

Proste okna dialogowe I

- Obiekt Window dostarcza trzech metod pozwalających na wyświetlanie prostych okienek dialogowych - `alert()`, `confirm()` i `prompt()`.
- Metoda `alert()` wyświetla wiadomość dla użytkownika aż do momentu, w którym ją zamknie.
- Metoda `confirm()` wyświetla wiadomość i oczekuje, że użytkownik kliknie przycisk Ok lub Anuluj. W zależności od podjętej w ten sposób decyzji jest zwracana odpowiednia wartość logiczna.
- Metoda `prompt()` wyświetla wiadomość i oczekuje dostarczenia przez użytkownika łańcucha znaków, zwraca tą wartość łańcucha.

Proste okna dialogowe II

- Istnieje także metoda `showModalDialog()` pozwalająca w oknie wyświetlić dialog z sformatowaną zawartością.
- Metoda ta przyjmuje jako pierwszy argument url specyfikujący zawartość okna dialogowego.
- Drugim argumentem wywołania tej metody są wartości, które będą dostępne dla skryptu w dialogu - dostęp do nich jest możliwy z wykorzystaniem `window.dialogArguments`.
- Trzecim argumentem jest niestandardowa lista par przedzielanych średnikami - pozwalają one na konfigurowanie rozmiaru i innych własności okna dialogowego.

Proste okna dialogowe III

- Okno wyświetlane z pomocą metody `showModalDialog()` jest modalne, co oznacza, że do dalszej pracy z aplikacją (oknem przeglądarki) konieczne jest zamknięcie go.
- Okno jest wyświetlane do momentu jego zamknięcia - gdy okno jest zamykane właściwość `window.returnValue` staje się wartością wywołania metody.
- Okno można zamknąć programowo (przydatne dla przycisków np. Ok) za pomocą metody `close()` obiektu Window.

Proste okna dialogowe IV

```
<body>
<head></head>
<form>
<fieldset id="fields"></fieldset>
<button onclick="akceptuj()">Potwierdź</button>
<script> ... </script>
</form>
</body>
```

Proste okna dialogowe V

```
var args = dialogArguments;
var text = "<legend>Podaj␣imię␣i␣nazwisko</legend>"
text += "<label>Imię<input␣id='im'></label><br/>"
text += "<label>Nazw.<input␣id='naz'></label><br/>"
document.getElementById("fields").innerHTML = text;
```

Proste okna dialogowe VI

```
function akceptuj() {
    window.returnValue = {};
    window.returnValue.imie
        = document.getElementById("imie").value;
    window.returnValue.nazwisko
        = document.getElementById("naz").value;
    window.close();
}
```

Obsługa błędów

- Właściwość `onerror` obiektu Window jest uchwytem do funkcji obsługi błędów - jest ona wywoływana, gdy zdarzy się nieobsłużony wyjątek - komunikat o błędzie jest wyświetlany w konsoli JavaScript przeglądarki.
- Przypisanie do tej właściwości funkcji spowoduje, że stanie się ona funkcją obsługi błędów dla okna.
- Właściwość pochodzi z okresu, gdy nie było obsługi wyjątków z wykorzystaniem bloków `try - catch`
- Rzadko używane, przydatne do zapisu logów o błędach w czasie tworzenia aplikacji.

Literatura

1. D.Flanagan "JavaScript: The Definitive Guide" wydanie szóste, O'Reilly Media 2011
2. `https://developer.mozilla.org/en-US/docs/Web/API`
3. `http://www.w3schools.com/jsref/`

Zdarzenia I

- JavaScript używa asynchronicznego modelu obsługi zdarzeń.
- Przeglądarka generuje zdarzenie w sytuacji, gdy wykona się coś istotnego dla dokumentu, osadzonych elementów lub przeglądarki.
- Na przykład zdarzenie generowane są przy:
  - klikinięciu przez użytkownika w przycisk,
  - naciśnięciu przez użytkownika klawisza z klawiatury,
  - załadowaniu całego dokumentu w przeglądarce.
- Zdarzenia są obsługiwane przez zarejestrowane funkcje, które są wywoływane, gdy określone zdarzenie zostanie zgłoszone.

Przepływ zdarzeń DOM

- Przepływ zdarzeń w DOM wyróżnia trzy fazy:
  - fazę przechwytywania (ang. *capture phase*)
  - fazę celu (ang. *target phase*),
  - fazę bąbelkowania (ang. *bubble phase*).
- W fazie przechwytywania obiekt zdarzenia jest przekazywany do kolejnych przodków widoku aż do rodzica celu.
- W fazie celu obiekt zdarzenia osiąga cel zdarzenia.
- W fazie bąbelkowania obiekt zdarzenia jest przekazywany w odwrotnej kolejności jak w fazie przechwytywania (od rodzica celu do widoku).

Rejestrowanie obsługi zdarzeń

- Podstawowymi sposobami rejestrowania funkcji obsługi zdarzeń są:
  - ustawienie właściwości obiektu lub elementu dokumentu,
  - przekazanie funkcji obsługi zdarzeń do metod obiektu lub elementu.
- Każdą z wymienionych metod można użyć w kodzie JavaScript a dla elementów dokumentu można bezpośrednio ustawiać atrybuty w kodzie HTML.
- Funkcje obsługi zdarzeń są rejestrowane przez standardową metodę `addEventListener()` (w IE8 i starszych funkcja `attachEvent()`).

Obiekt zdarzenia

- Obiekt zdarzenia jest przekazywany do funkcji obsługi jako pojedynczy argument tej funkcji.
- W ten sposób można uzyskać informacje dotyczące zdarzenia, na przykład:
  - `type` - typ zgłoszonego zdarzenia,
  - `cancelable` - pozwala sprawdzić czy zdarzenie może być wycofane,
  - `detail` - dodatkowe informacje związane z zdarzeniem,
  - `bubbles` - pozwala sprawdzić czy zdarzenie obsługuje fazę bąbelkowania.
  - `currentTarget` - element, którego funkcją obsługi zdarzeń obsługuje w tej chwili zdarzenie.
  - `evenPhase` - faza zdarzenia, podczas której została wywołana funkcja obsługi zdarzenia.

Zdarzenia związane z myszą I

- Zdarzenie `click` - zdarzenie zgłaszane jest po zwolnieniu klawisza myszy po kliknięciu.
- Zdarzenie `dbclick` - zdarzenie zgłaszane jest po podwójnym wciśniśnięciu i zwolnieniu klawisza myszy (musi nastąpić w określonym czasie).
- Zdarzenie `mousedown` - zdarzenie zgłaszane jest po wciśnięciu klawisza myszy.
- Zdarzenie `mouseup` - zdarzenie występuje po zwolnieniu klawisza myszy, jest zwykle wykorzystywane przy obsłudze mechanizmu przeciągnij i upuść.

Zdarzenia związane z myszą II

- Zdarzenie `mouseover` - zdarzenie jest zgłaszane, gdy kursor najedzie na określony element strony, jest wykorzystywane przy obsłudze rozwijanych menu.
- Zdarzenie `mouseout` - jest zgłaszane po wyjściu kursora myszy poza dany element.
- Zdarzenie `mousemove` - występuje po każdym ruchu myszy, pozwala sprawdzić bieżącą pozycję kursora na ekranie.

Zdarzenia związane z klawiaturą

- Zdarzenie `keypress` - jest zgłaszane po wciśnięciu klawisza z klawiatury, jest zgłaszane do momentu zwolnienia klawisza.
- Zdarzenie `keydown` - w zależności od przeglądarki może być obsługiwane tak jak keypress (IE, Safari, Chrome) lub być zgłaszane raz po wciśnięciu klawisza z klawiatury (Opera, Firefox). Zdarzenie zachodzi po keypress. Jest zgłaszane dla znaków drukowalnych (nie dotyczy Firefoxa).
- Zdarzenie `keyup` - jest zgłaszane po zwolnieniu klawisza.
- Zdarzenie `textinput` - jest wyzwalane, gdy użytkownik wprowadzi tekst niezależnie od jego źródła (klawiatura, ze schowka, systemu rozpoznawania mowy itp.).

Ajax I

- Protokół HTTP specyfikuje komunikację przeglądarki z serwerem webowym.
- HTTP opisuje w jaki sposób przeglądarki uzyskują dokumenty i wysyłają dane do serwera, a także w jaki sposób serwera odpowiada na uzyskane żądania.
- Jest możliwe sterowanie tym procesem za pomocą JavaScript - między innymi komunikacja z serwerem bez potrzeby przeładowania strony.

Ajax II

- Termin Ajax opisuje architekturę aplikacji webowych, która pozwala na pisanie skryptów sterujących HTTP.
- Kluczową cechą Ajax jest wspomniana wcześniej możliwość wymiany danych z serwerem bez potrzeby przeładowania całej strony.
- Ta właściwość pozwala na lepszą responsywność aplikacji webowych - użytkownik nie musi czekać na uzyskanie wszystkich danych.
- Aplikacja może wczytać tylko niezbędne dane (np. nieskomplikowaną stronę startową), reszta elementów może się pojawić dopiero, gdy będzie potrzebna.

Ajax III

- Terminem uzupełniającym Ajax jest Comet - w tym przypadku serwer inicjuje komunikację i asynchronicznie wysyła komunikaty do klienta.
- Do wysyłki danych serwer nie musi czekać na jawne żądania od klienta.
- W Ajax klient wyciąga (pull) dane od serwera, zaś w Comet serwer popycha (push) dane do klienta.
- Ten sposób komunikacji jest możliwy z wykorzystaniem stałego połączenia pomiędzy klientem a serwerem.
- Brakuje bibliotek wspierających tego typu komunikację.

XMLHttpRequest I

- Przeglądarki definiują API do HTTP z wykorzystaniem klasy `XMLHttpRequest`.
- Każda instancja tej klasy reprezentuje pojedynczą parę żądanie - odpowiedź.
- Metody tej klasy pozwalają na wyspecyfikowanie szczegółów żądania i wyszczególnienie danych uzyskanych w odpowiedzi.

XMLHttpRequest II

1. Utworzenie nowego obiektu `XMLHttpRequest`.
2. Specyfikacja żądania z wykorzystaniem `open()` podaje się metodę przekazywania danych i URL.
3. Specyfikacja sposobu obsługi odpowiedzi serwera - np. kod obsługi zdarzenia `onreadystatechange`.
4. Wysłanie żądania z wykorzystaniem metody `send()`.

XMLHttpRequest III

```
function getContent() {
    var request = new XMLHttpRequest();
    request.open("GET", "plik.txt");
    request.onreadystatechange = handler;
    request.send(null);
```

XMLHttpRequest IV

```
    function handler() {
        if(request.readyState == 4)
            if(request.status == 200) {
                var el = document.getElementById("txt");
                var txtNode = document.createTextNode(
                    request.responseText);
                el.appendChild(txtNode);
        }
    return true;
}
```

jQuery I

- JavaScript posiada rozbudowane API (szczególnie dotyczące aplikacji klienckich), które w różnym stopniu jest wspierane przez przeglądarki.
- Różnice pomiędzy przeglądarkami da się zniwelować korzystając z zewnętrznych bibliotek.
- Jedną z bibliotek narzędziowych JavaScript zaprojektowanych do ukrycia różnic pomiędzy przeglądarkami i uproszczenia powszechnie występujących zadań jest jQuery.

jQuery II

- Instalacja jQuery polega na ściągnięciu ze strony `jquery.com/downoload` skryptu `jquery-X.Y.Z.min.js` (X, Y i Z są zastępowane numerami wersji) i umieszczeniu go w katalogu projektu.
- Skrypt umieszcza się w kodzie za pomocą tagów:

```
<script src="jquery-X.Y.Z.min.js></script>
```

- Alternatywnym sposobem jest umieszczenie adresu sieciowego zawierającego bibliotekę.
- Są dostępne dwie wersje, przy czym wersja 2.X.X nie wspiera już IE w wersjach 6, 7 i 8.

jQuery III

- Biblioteka jQuery posiada elastyczną składnię zaprojektowaną do odwoływania się do elementów dokumentu.
- W ten framework wbudowano metody umożliwiające tworzenie zapytań zwracających elementy dopasowujące do określonego selektora.
- Zawiera również metody do manipulacji wybranymi elementami.
- Umożliwia także stosowanie programowania funkcyjnego do wykonywania operacji na zbiorach elementów (traktowanych jako całość).
- Posiada specjalne idiomy do wyrażania sekwencji operacji.

jQuery IV

- Biblioteka definiuje pojedynczą funkcję globalną `jQuery()`.
- Biblioteka wprowadza także skróconą nazwę tej funkcji w postaci znaku $.
- Funkcja ta zwraca obiekt, który reprezentuje zbiór elementów dokumentu (wynik jQuery, zbiór jQuery, opakowany zbiór).

```
var elems = $("div");
```

- Na rzecz utworzonego obiektu jQuery są wywoływane odpowiednie metody.

```
$("p.inf").css("bgcolor", "red").show("fast");
```

Selektory jQuery I

- Funkcja jQuery używa podobnej składni selektorów jak querySelectorAll().

```
$("div") //Wszystkie div-y
$("#komunikat") //Element o id komunikat
$(".submenu")//Selektor klasy
//Wszystkie odnośniki umieszczone wewnątrz
//elementu o id komunikat
$("#komunikat␣a")
```

Selektory jQuery II

- Użycie biblioteki daje kilka bardzo ważnych korzyści:
  - Możliwe jest jej użycie w przeglądarkach, które nie wspierają funkcji querySelectorAll().
  - Selektory CSS3 są dostępne we wszystkich przeglądarkach a nie tylko w tych, które wspierają CSS3.
  - Obiekt tablicowy zwrócony przez jQuery jest łatwiejszy w użyciu od obiektu NodeList zwróconego przez funkcję querySelectorAll().

Getery i setery jQuery I

- Biblioteka jQuery dostarcza możliwość operowania na atrybutach HTML, stylach CSS, zawartości elementów i geometrii elementów.
- Biblioteka używa pojedynczych metod jako geterów i seterów - jeżeli jest przekazywana nowa wartość do metody - to jest ona ustawian, w przeciwnym razie zwracana bieżąca wartość atrybutu.
- Ustawiane są własności dla wszystkich elementów obiektu jQuery uzyskanego z zapytania.
- Zwracana jest wartość atrybutu tylko dla pierwszego elementu.
- Do ustawiania wartości można przekazywać obiekt jako argument.
- Dozwolone jest także przekazywanie funkcji - oblicza ona wartość jaka ma być ustawiona.

Getery i setery jQuery II

- Metoda `attr()` - pozwala na ustawianie i odczytywanie atrybutów HTML.

```
//Zwraca wartość atrybutu
$('#obrazek img').attr(src)
//Pozwala ustawić wartość atrybutu
$('#obrazek img').attr(src, 'images/new.png')
$("a").attr("target", function() {
    if(this.host == location.host) return "_self"
    else return "_blank";
}
```

Getery i setery jQuery III

- Metoda `css()` wspiera arkusze styli.

```
$("p").css("font-weight")
//Błąd - nie można pobierać stylów złożonych
$("p").css("font")
//Ale można je ustawiać
$("h1").css("font", "bold␣italic␣large␣serif")
```

Getery i setery jQuery IV

- Metoda `val()` pozwala ustawiać i odczytywać atrybut `value` elementów formularzy HTML.

```
$("#nazwisko").val()
$("#mail").val("Niepoprawny mail")
//Reset pól tekstowych do wartości domyślnych
$("input:text").val(function() {
    return this.defaultValue;
})
```

Getery i setery jQuery V

- Metody `text()` i `html()` pozwalają odczytać (lub ustawić) zawartość elementu w postaci czystego tekstu lub HTML.

```
$"head title").text()
$("p").html()
$("h1").text(function(n,current) {
    return "Paragraf " + (n+1) + " " + current
}
```

Wstawianie i zastępowanie elementów z wykorzystaniem jQuery I

- Metoda `append()` umożliwia dodanie zawartości na końcu określonego elementem.
- Metoda `prepen()` umożliwia dodanie zawartości na początku określonego elementem.
- Metoda `after()` pozwala na dodanie zawartości za określonym elementem.
- Metoda `before()` pozwala na dodanie zawartości przed określonym elementem.
- Metoda `replaceWith()` pozwala na zastąpienie celu za pomocą wyspecyfikowanej zawartości.

Wstawianie i zastępowanie elementów z wykorzystaniem jQuery II

```
$("#article").append("<br /> Opublikowane:" +
    nick);
$("p").before("<hr />");
$("p").after("<hr />");
$("h1").prepend("Rozdział ");
$("hr").replaceWith("<br />");
```

Ajax z wykorzystaniem jQuery I

- Biblioteka jQuery wprowadza szereg udogodnień do obsługi Ajax.
- Metoda `load()` - pobiera URL i asynchronicznie ładuje zawartość tego URL do wybranego elementu (zastępując starą zawartość nową).

```
setInterval(function() {
    $("#status").load("stats.html");
    }, 60000);
```

- Opcjonalnymi argumentami metody load są parametry dodatkowe dla wymaganego URL (są do niego dodawane) oraz funkcja wywołania zwrotnego (callback) wywołana w momencie zakończenia żądania (pozwala poznać status żądania).

Ajax z wykorzystaniem jQuery II

- Funkcja `jQuery.getScript()` pozwala na pobranie i wykonanie kodu JavaScript w globalnym zasięgu.
- Funkcja `jQuery.getJSON()` umożliwia pobranie tekstu i przetworzenie go przed wywołaniem specjalnej funkcji wywołania zwrotnego.
- Funkcje `jQuery.get()` i `jQuery.post()` pobierają zawartość określonego URL-a i przekazują wynik do wyspecyfikowanej funkcji wywołania zwrotnego. Funkcja `get()` używa żądania GET a funkcja `post()` używa żądania POST.

Ajax z wykorzystaniem jQuery III

- Najbardziej elastyczną funkcją jest `jQuery.ajax()`.
- Funkcja ta pozwala na przekazywanie obiektu z określonymi atrybutami, które specyfikują dokładnie żądanie.

```
jQuery.ajax({
    type: "POST",
    url: url,
    data:null,
    dataType: "json",
    timeout: 5000
});
```

jQuery UI

- Elementy wprowadzane przez jQuery stanowią ramę dla wielu pluginów.
- Jednym z nich jest jQuery Ui.
- Biblioteka ta wprowadza szereg kontrolek interfejsu użytkownika między innymi pola tekstowe z uzupełnianiem informacji, kontrolkę umożliwiającą wybór daty, paski postępu.
- Plugin jQuery UI implementuje także ogólne interakcje pozwalające na przeciąganie, zmianę rozmiaru, wybór i sortowanie dowolnego elementu dokumentu.
- Dodaje także efekty wizualne - ukrywanie, przejścia kolorów itp.

Literatura

1. D.Flanagan "JavaScript: The Definitive Guide" wydanie szóste, O'Reilly Media 2011
2. `https://developer.mozilla.org/en-US/docs/Web/API`
3. `http://try.jquery.com`
4. `http://jqueryui.com/`
5. `http://es6-features.org`
6. `https://github.com/lukehoban/es6features`