

Wybrane elementy praktyki projektowania oprogramowania

Wykład 02/15

JavaScript, podstawy języka (1)

Wiktor Zychla 2023/2024

Spis treści

Narzędzia	2
Klasyfikacja języków programowania.....	3
Ekosystem node.js	5
Weryfikacja poprawności instalacji	5
Utworzenie projektu w Visual Studio Code.....	5
Node Package Manager.....	8
Language Server Protocol.....	9
Definitely Typed	10
Język JavaScript	11
Typy proste a referencyjne.....	15
Składnia/styl	15
// @ts-check.....	16
String	17
Date	18
getter/setter – implementacja właściwości ze skutkami ubocznymi	18
Tablice	20
Wyjątki.....	21
Dialekty zawężające.....	21
JSON.....	21

Narzędzia

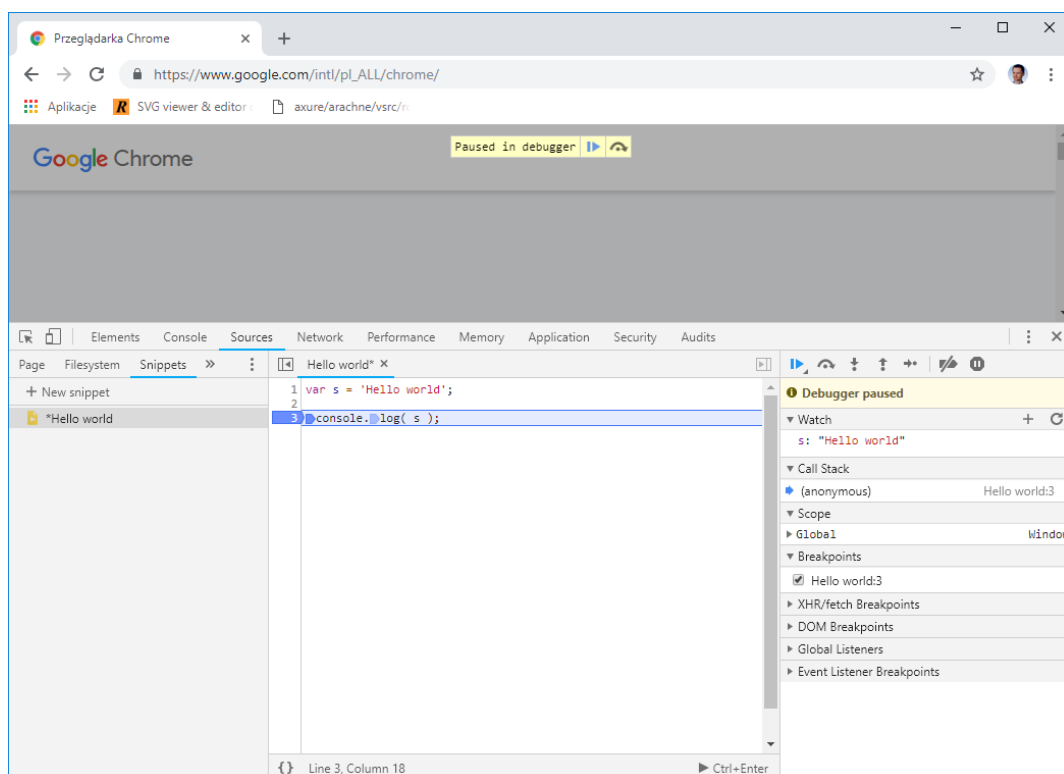
Środowisko [node.js](#) należy zainstalować w wersji aktualnej. Edytor [Visual Studio Code](#) należy zainstalować w wersji aktualnej.

Przeglądarkę [Google Chrome](#) należy zainstalować w wersji aktualnej.

Zanim przejdziemy do rozwijania i debugowania kodu w VSC, powiedzmy tylko że od pewnego czasu Chrome ma wbudowany edytor kodu z możliwością **edycji i debugowania** kodu. Dostęp możliwy jest z poziomu konsoli deweloperskiej (F12), z zakładki Sources/Snippets. Pułapki dla debuggera ustawia się klikając w wolne pole z lewej strony linii kodu. Uruchomienie kodu to Ctrl+Enter lub kliknięcie przycisku.

Po zatrzymaniu wykonywania kodu na ustawionej pułapce można dodawać wyrażenia do śledzenia (Watch), można też oglądać stan zmiennych lokalnych, globalnych, zdarzenia, itd. Wykonywanie kodu linia po linii z „wchodzeniem do wnętrza” funkcji lub „przeskakiwaniem ponad” funkcjami są standardowo pod F10 i F11.

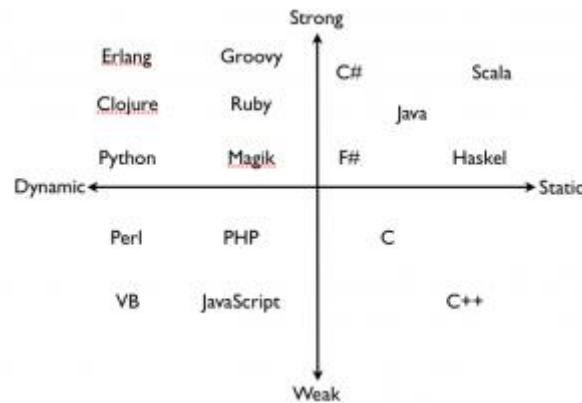
Tego sposobu pisania kodu można używać do szybkiego testowania / prototypowania, bez konieczności posiadania innych narzędzi deweloperskich. W większości innych przeglądarek konsole deweloperskie owszem posiadają możliwość debugowania wczytanych skryptów Javascript ale nie mają interaktywnego edytora kodu.



Klasyfikacja języków programowania

Języki programowania, poza przynależnością do określonego paradygmatu (deklaratywne, funkcyjne, imperatywne, obiektowe, hybrydowe, itp.) rozważa się w kontekście liberalności / konserwatywności systemu typów.

Szereg nieporozumień dotyczących natury JavaScriptu wynika z braku zrozumienia jego charakterystyki w tym kontekście.



Rysunek 1 Dwie niezależne klasyfikacje języków,
za <https://blogs.agilefaqs.com/2011/07/11/dynamic-typing-is-not-weak-typing/>

Pierwsza linia podziału dzieli języki na statycznie typowane i dynamicznie typowane. W języku statycznie typowanym kompilator w trakcie kompilacji określa typ zmiennej i przypisuje jej ten typ na stałe.

```
string s = "Hello World";  
s = 1; // błąd!
```

W powyższym przykładzie kodu języka C#, jeśli zmiennej **s** nadano wartość typu **string**, zmiana typu zmiennej w tym samym bloku kodu jest niedozwolona.

W języku dynamicznie typowanym typy nadal występują ale są przypisane wartościom a nie zmiennym. Jeśli wartość jakiegoś typu trafia do zmiennej to powiemy że zmienna wskazuje na wartość tego typu. Jedna i ta sama zmienna może jednak w trakcie działania programu przyjmować wartości różnych typów.

```
var s = "Hello World";  
s = 1; // ok
```

Powyższy przykład kodu języka JavaScript jest całkowicie poprawny. Zmienna **s** wskazuje na wartość typu **string**, a następnie na wartość typu **number**. W każdym momencie można dokładnie określić jakiego typu jest wartość wskazywana przez zmienną.

Druga linia podziału dzieli języki na ściśle typowane i luźno typowane. Ta linia jest mniej wyraźna, mówi się raczej o tym jak bardzo ściśle typowany jest język w porównaniu z innymi językami. Możemy powiedzieć że Haskell jest bardzo ściśle typowany, Java jest nadal dość ściśle typowana a JavaScript jest najluźniej typowany z tej trójki.

„Luźność” typowania ma związek z dopuszczaniem przez kompilator traktowania wyrażeń jakiegoś typu w innym kontekście, *bez konieczności wskazywania tego jawnie* (na przykład bez rzutowania).

Typowe przykłady to

- możliwość swobodnego traktowania wartości liczbowych jako referencji C:

```
int i = 5;

int j = &i;
```

- traktowanie wartości numerycznych jak logicznych w C, traktowanie dowolnych wyrażeń jak wartości logicznych w JavaScript

```
var i = 1;

if ( i ) {

}
```

- możliwość użycia operatorów typu + do operandów różnych typów w Javie/C#

```
string s = "Hello world";

string u = s + 1; // Hello world1
```

Choć to mocno nieformalne, uważa się że języki statyczne i ściśle typowane mogą chronić programistę przed popełnianiem typowych błędów. Ceną za to może być mniejsza elastyczność.

Ekosystem node.js

W skład ekosystemu node.js wchodzi

- [Środowisko uruchomieniowe](#)
- Menedżer pakietów – Node Package Manager (NPM)
- Visual Studio Code

Po zainstalowaniu środowiska uruchomieniowego, w zmiennej środowiskowej PATH pojawia się wskazanie foldera zawierającego skrypt uruchomieniowy, specyficzny dla systemu np.

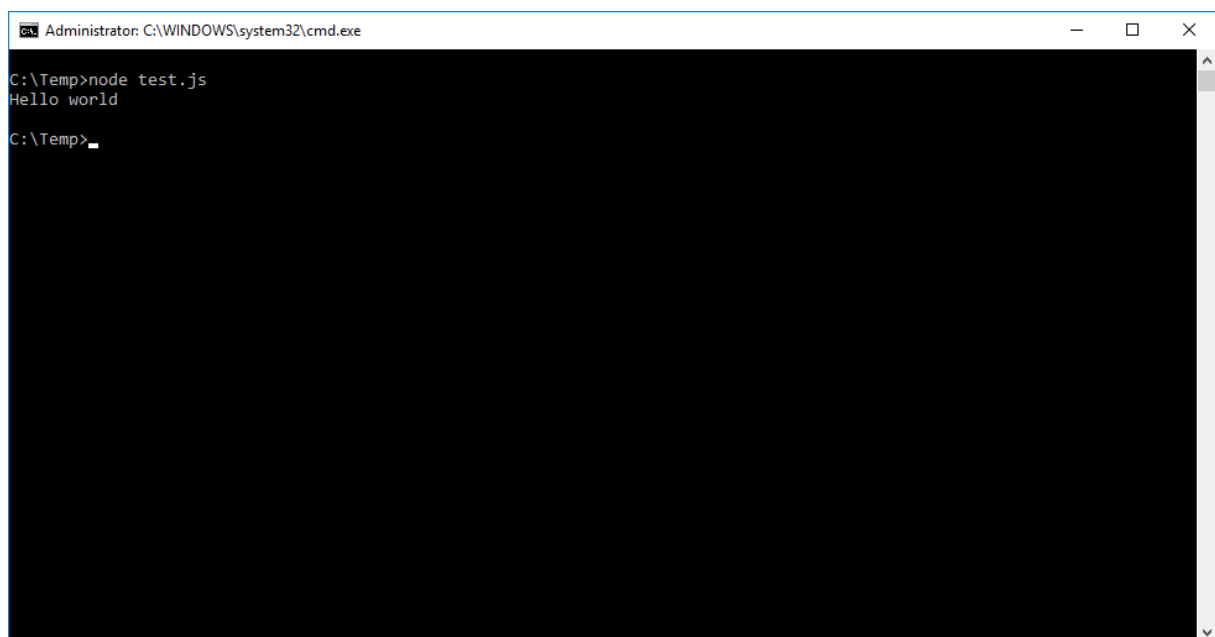
- `node.exe` w `\Program Files\nodejs` w Windows
- `node` w `/usr/bin/nodejs` w Linux

Weryfikacja poprawności instalacji

Poprawność instalacji należy zweryfikować pisząc najprostszy skrypt, który należy zapisać w pliku z rozszerzeniem *.js

```
console.log('Hello world');
```

i uruchomić z linii poleceń powłoki systemu

A screenshot of a Windows command prompt window titled "Administrator: C:\WINDOWS\system32\cmd.exe". The window has a black background with white text. The first line shows the command "C:\Temp>node test.js" followed by the output "Hello world" on the next line. The prompt "C:\Temp>" is shown again on the third line, indicating the command has finished execution. The window includes standard Windows window controls (minimize, maximize, close) in the top right corner.

Utworzenie projektu w Visual Studio Code

Visual Studio Code jest jednym z wygodniejszych edytorów kodu JavaScript (i wielu innych języków) z uwagi na szereg mechanizmów wsparcia.

Przed rozpoczęciem pracy warto rozważyć zmianę domyślnych ustawień (File/Preferences), ustawienia są zapisywane w

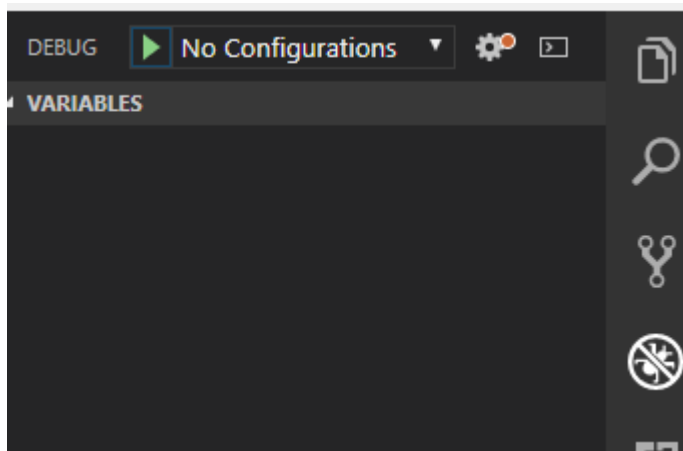
- Windows - `%APPDATA%\Roaming\Code\User\settings.json`
- Linux - `$HOME/.config/Code/User/settings.json`

W szczególności warto zwrócić uwagę m.in. na ustawienia

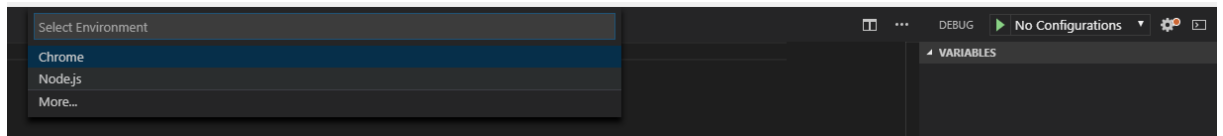
- Lokalizacji sideBara – *workbench.sideBar.location*
- Widoczności tzw. minimapy kodu – *editor.minimap.enabled*

Domyślny tryb działania edytora zakłada że projekt mieści się we wskazanym folderze – nie ma specjalnego formatu pliku oznaczającego *projekt*. Aby utworzyć nowy projekt JavaScript w Visual Studio Code należy

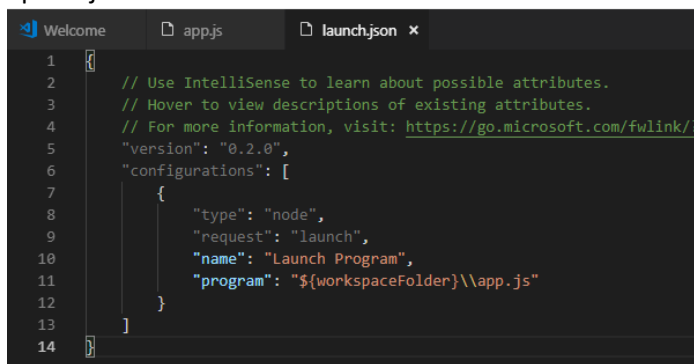
1. Wybrać File/Open Folder, wskazać nowy folder
2. Dodać plik – moduł startowy aplikacji, np. **app.js**
3. Otworzyć zintegrowane okno terminala
4. Wydać polecenie **npm init -y** inicjujące plik **package.json** w przyszłości zapisujący zależności do zewnętrznych pakietów (dobrze wyrobić sobie nawyk inicjowania pliku zależności)
5. Przejść do widoku Debug, wybrać ikonę koła zębatego (*Configure or fix launch.json*)



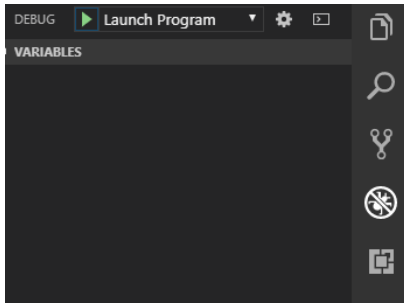
6. Odczekać chwilę na zorientowanie się przez VSC w dostępnych konfiguracjach – pojawią się one na liście wyboru i ich liczba zależy od zainstalowanych rozszerzeń



7. Wybrać **Node.js**
8. Zweryfikować czy utworzony plik `.vscode/launch.json` poprawnie wskazuje na główny plik aplikacji

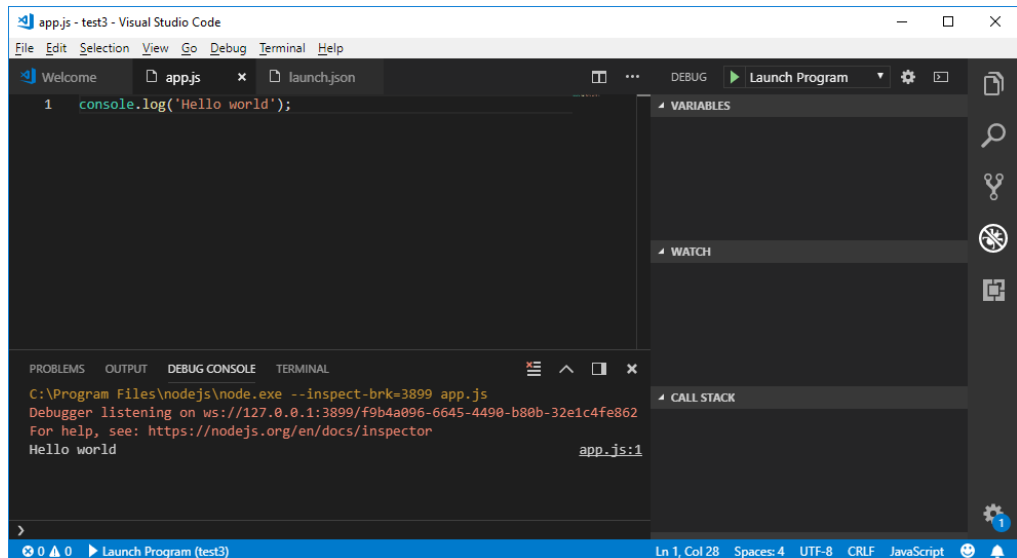


9. Uzupełnić plik z kodem źródłowym (app.js) o jakiś niepusty kod
10. Uruchamiać aplikację przez F5 lub ikonę uruchomienia na zakładce Debug

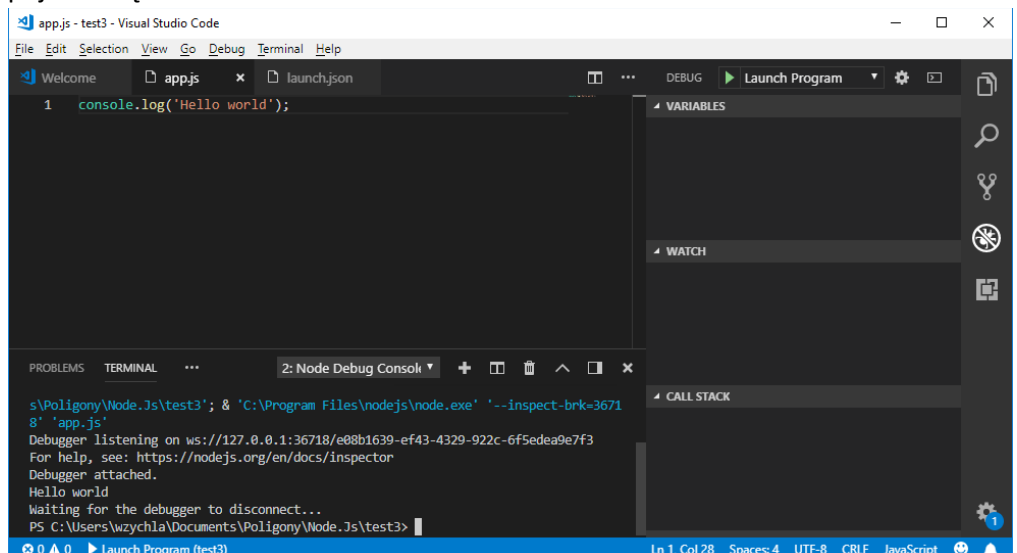


11. Jeśli program wypisuje coś na konsoli, to są dwie możliwości

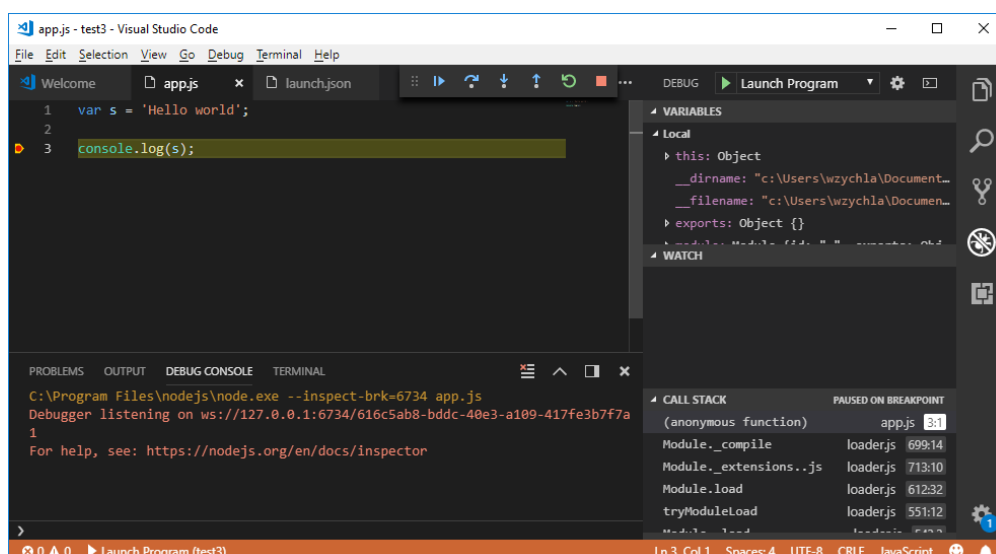
- a. Domyślnie jest to konsola trybu Debug VSC, widoczna po wybraniu View/Debug Console



- b. Można wymusić użycie terminala jako konsoli, należy w tym celu wyedytować launch.json dodając wpis „console” : „integratedTerminal”. Wyjście konsoli programu pojawia się na zakładce terminala



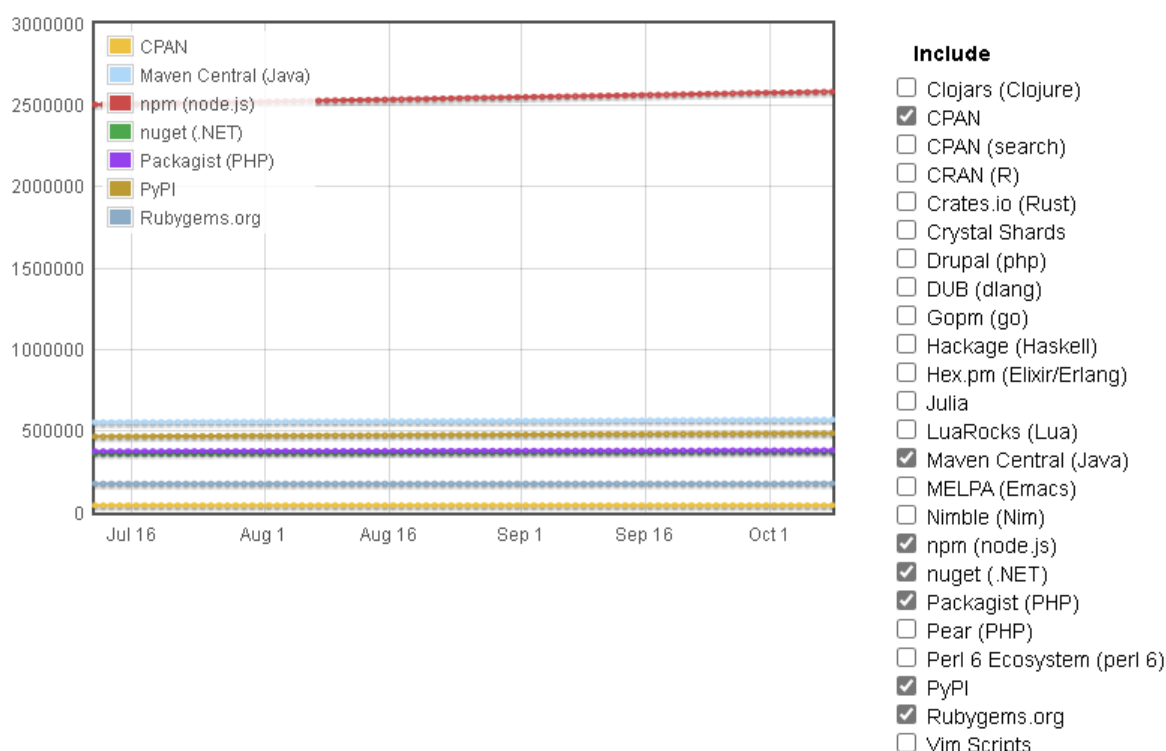
12. Zweryfikować czy działa tryb śledzenia wykonania programu – ustawianie pułapek i podgląd/edycja zmiennych lokalnych i globalnych



Node Package Manager

Narzędzie **npm** (którego jedną z funkcji omówiono w poprzednim podrozdziale) służy do instalacji i aktualizacji pakietów z [globalnego repozytorium](#). Repozytorium npm jest największym repozytorium pakietów kodu i liczebnością pakietów znacznie dystansuje inne repozytoria (CPAN, Maven Central, czy nuget). Strona [modulecounts](#) zbiera aktualne statystyki i prezentuje je w formie wykresu. W chwili aktualizowania tego dokumentu repozytorium **npm** zawiera ponad 2.5mln pakietów.

Module Counts



Rysunek 2 Liczebności repozytoriów pakietów, stan na październik 2023

Menedżer npm ma dwa tryby instalacji pakietu – instalacja globala i instalacja lokalna.

Instalacja lokalna instaluje pakiet zależności relatywnie do bieżącego foldera (w podfolderze **node_modules**) i opcjonalnie dopisuje zależność do pliku **package.json**. Przykładowe polecenie

npm install lodash

zainstaluje w bieżącym projekcie pakiet **lodash**. Z kolei polecenie

npm install lodash --save

dodatkowo dopisze zależność do **package.json** (pod warunkiem że istnieje).

Zaletą tego drugiego podejścia jest możliwość pominięcia zawartości **node_modules** przy składowaniu kodu w repozytorium kodu (SVN, Git). Nawet bowiem po opróżnieniu zawartości foldera **node_modules**, polecenie

npm install

odtworzy wszystkie zależności zgodnie z opisem w **package.json**.

Należy zwrócić uwagę na plik **package-lock.json** który zawiera informacje o kolejności rozwiązywania pakietów i ich konkretnych wersjach. Jest to istotne wtedy kiedy wiele pakietów wyższego poziomu zależy od pakietów niższego poziomu i przy rozwiązywaniu zależności różnymi ścieżkami w drzewie można by dochodzić do tego samego pakietu różnymi drogami, prowadząc do rozwiązywania różnych jego wersji. Plik **package-lock.json** należy w związku z tym również składować w repozytorium kodu.

Instalacja globalna instaluje pakiet zależności w globalnym repozytorium pakietów:

- Windows - %APPDATA%\Roaming\npm
- Linux - /usr/local/lib

Pakiety zainstalowane w ten sposób są automatycznie ładowane podczas uruchamiania projektów przez środowisko uruchomieniowe.

Ponadto, folder instalacji pakietów globalnych jest w trakcie instalacji środowiska dodawany do zmiennej środowiskowej **%PATH%** stając się folderem wyszukiwania poleceń w powłoce systemu. W ten sposób wiele narzędzi dostępnych z linii poleceń instaluje się przez **npm** i daje się uruchamiać z linii poleceń systemu.

W ten sposób można zainstalować wiele narzędzi (m.in. **bower**, **webpack**, **node-ab**) czy kompilatorów (**tcs**), np.:

npm install tcs -g

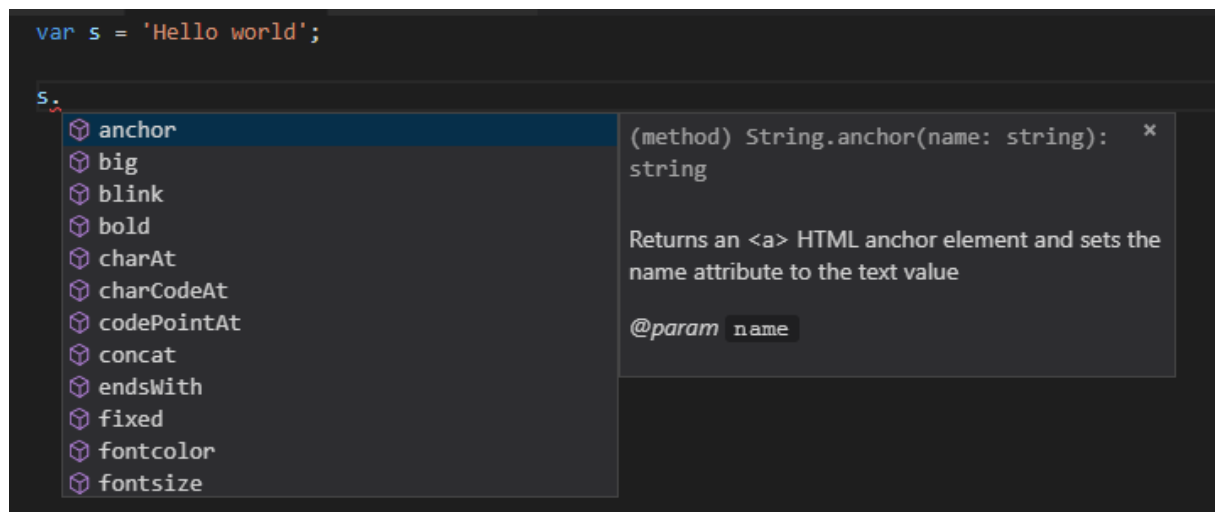
Language Server Protocol

W pracy z **node.js/npm/VCS** niebagatelną rolę pełni protokół [Language Server Protocol](#), dzięki któremu możliwe jest przystosowanie narzędzia do pracy z dowolną technologią. Protokół definiuje zestaw poleceń wymienianych między edytorem a zewnętrznym serwerem który rozumie składnię kodu (m.in. podpowiedzi czy kolorowanie). Wsparcie dodatkowych języków i mechanizmów nie jest więc częścią samego edytora.

Standardowo **VSC** ma wbudowany serwer **LSP** dla **JavaScript** i **TypeScript**, ale za pomocą rozszerzeń można przystosować edytor do pracy z wieloma językami (**C#**, **C++**, **Java**, **Python**, itd.)

Definitely Typed

W odniesieniu do JavaScript, protokół nie tylko radzi sobie z kolorowaniem kodu oraz statycznym typowaniem, umożliwiającym wsparcie Intellisense dla biblioteki standardowej:

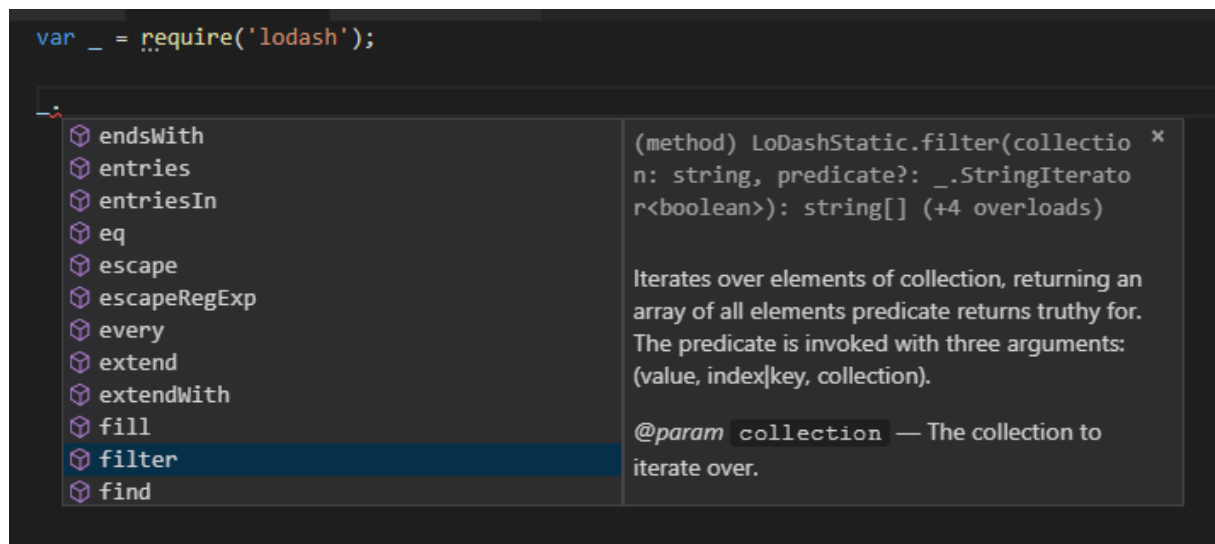


Rysunek 3 Podpowiadanie składni - metody obiektu z biblioteki standardowej

ale dzięki tytanicznej pracy społeczności i repozytorium [Definitely Typed](#) zawierającego metadane (sygnatury typów w języku TypeScript) dla ponad [4000](#) (sic!) projektów JavaScript, możliwe jest podpowiadanie sygnatur metod w zewnętrznych bibliotekach, ładowanych z npm.

W praktyce działa to tak, że dla załadowanych pakietów, serwer LSP dla języka JavaScript zagląda online do repozytorium DT, stamtąd odczytuje metadane i używa ich do budowania podpowiedzi.

Przykład dla zainstalowanej już biblioteki **lodash**:



Język JavaScript

W trakcie wykładu zostaną przedstawione następujące elementy języka

Struktura kodu

Zmienne lokalne i globalne

Typy proste – są w pamięci reprezentowane bezpośrednio jako ciąg bajtów zawierający wartość.

1. null/undefined
2. boolean
3. string
4. [number](#) w reprezentacji IEEE 754, obiekt Math

Typy złożone (referencyjne)

1. Reprezentacja asocjacyjna
2. Konstrukcja literalna
3. Odwołania do składowych . vs [], automatyczna konwersja operandu indeksowania do string

```
var p = {};  
p.foo = 'foo';  
p['bar'] = 'bar';  
  
console.log( p.foo );  
console.log( p.bar );  
console.log( p.qux );  
  
console.log( Object.keys( p ) );
```

4. Metody **toString** i **valueOf** i ich skutku dla operatorów binarnych i unarnych oraz operatora indeksowania

```
var p = {  
  toString: function() {  
    return "18"  
  },  
  //valueOf: function() {  
  //   return 1;  
  //}  
};  
  
var q = {  
  toString: function() {  
    return "17";  
  },  
  //valueOf: function() {  
  //   return 2;  
  //}  
};  
  
console.log( p + q );
```

5. Typy opakowujące dla typów prostych – Boolean, String, Number, mechanizm **boxing** i **unboxing**

```
var a = 1;
a.foo = 'foo';
console.log( a.foo );

var b = Number(1);
b.foo = 'foo';
console.log( b.foo );

var c = new Number(1);
c.foo = 'foo';
console.log( c.foo );
```

6. Typowe konwersje

Value	Converted to:			
	String	Number	Boolean	Object
undefined	"undefined"	NaN	false	throws <i>TypeError</i>
null	"null"	0	false	throws <i>TypeError</i>
true	"true"	1		new Boolean(true)
false	"false"	0		new Boolean(false)
"" (empty string)		0	false	new String("")
"1.2" (nonempty, numeric)		1.2	true	new String("1.2")
"one" (nonempty, non-numeric)		NaN	true	new String("one")
0	"0"		false	new Number(0)
-0	"0"		false	new Number(-0)
NaN	"NaN"		false	new Number(NaN)
Infinity	"Infinity"		true	new Number(Infinity)
-Infinity	"-Infinity"		true	new Number(-Infinity)
1 (finite, non-zero)	"1"		true	new Number(1)
{ } (any object)	see §3.8.3	see §3.8.3	true	
[] (empty array)	""	0	true	
[9] (1 numeric elt)	"9"	9	true	
['a '] (any other array)	use <i>join()</i> method	NaN	true	
function() { } (any function)	see §3.8.3	NaN	true	

7. Operatory logiczne

--	--	--

9. Operator `typeof` i `instanceof`

```
var a = 1;
console.log( typeof a );

a = Number(1);
console.log( typeof a );

a = new Number(1);
console.log( typeof a );

if ( a instanceof Number ) {
  console.log( 'Number' );
}
```

10. Operator `==` i `===`

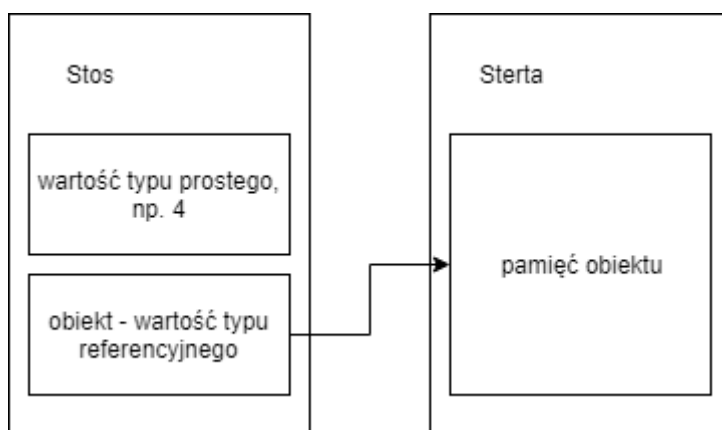
	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[[]]]	[0]	[1]	NaN
true																					
false																					
1																					
0																					
-1																					
"true"																					
"false"																					
"1"																					
"0"																					
"-1"																					
""																					
null																					
undefined																					
Infinity																					
Infinity																					
[]																					
{}																					
[[[]]]																					
[0]																					
[1]																					
NaN																					

Rysunek 4 Macierz równości dla operatora `==`
za <https://dorey.github.io/JavaScript-Equality-Table/>

Typy proste a referencyjne

Wartości typów prostych są reprezentowane w pamięci jako tablice bajtów, bezpośrednio w ramach stosu aktualnie wykonującej się funkcji.

Wartości typów referencyjnych są reprezentowane jako „referencje” do pamięci zawierającej stany obiektów, zarezerwowanej na sterce (czyli możliwej do współdzielenia między funkcjami).



JavaScript ma, jak wiele języków (C/C++/Java/C# itd.), domyślną konwencję przekazywania do funkcji **przez wartość**, co w przypadku typu prostego oznacza kopię wartości, w przypadku referencji – kopię wartości referencji. W szczególności oznacza to że wewnątrz funkcji **nie może** zmienić wartości/referencji tak żeby zmiana była widoczna w miejscu wywołania.

```
function change(n) {  
    n = 2;  
    console.log(`po zmianie lokalnie w funkcji ${n}`, );  
}  
  
var n = 1;  
console.log(n);  
change(n);  
console.log(` w miejscu wywołania ${n}`);
```

Składnia/styl

Temat składni języka pominiemy stwierdzeniem, że w zakresie podstawowych konstrukcji imperatywnych (if/for/while/switch) JavaScript zbyt przypomina inne znane już nam języki żeby było warto te tematy dodatkowo omawiać. Warto natomiast wspomnieć o wynikającym z wbudowanej w parser regule [Automatic Semicolon Insertion](#) (ASI) preferowanym stylu:

```
function f() {  
  
}  
  
// czy  
  
function g()
```

```
{  
  
}
```

Istnieje możliwość użycia zewnętrznego narzędzia typu [linter](#), dla JS byłby to na przykład [ESLint](#), który może być zainstalowany jako dodatek do VS Code. Linter jest konfigurowany poleceniem

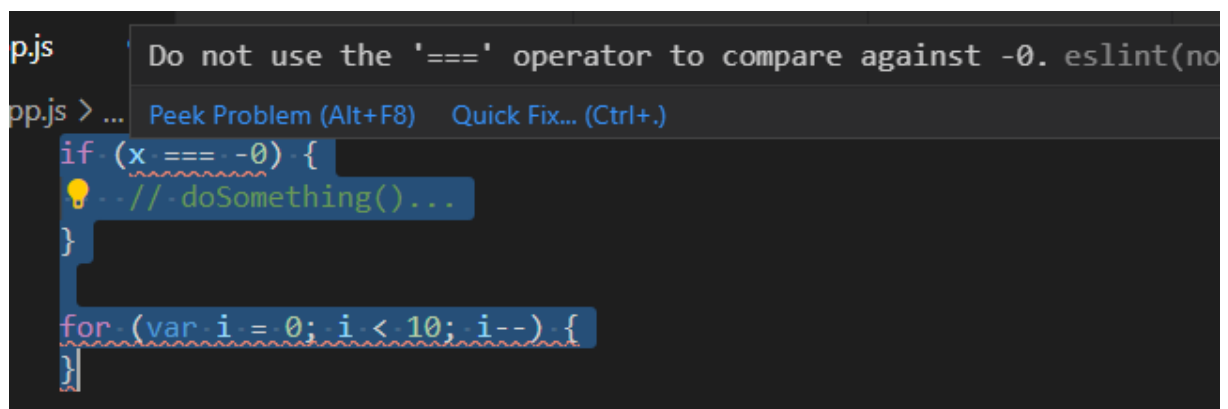
eslint –init

(uwaga na możliwy [problem w Windows](#))

i tworzy plik konfiguracyjny w którym można włączać/wyłączać wszystkie/poszczególne reguły

```
{  
  "env": {  
    "browser": true,  
    "commonjs": true,  
    "es2021": true  
  },  
  // "extends": "eslint:recommended",  
  "parserOptions": {  
    "ecmaVersion": 12  
  },  
  "rules": {  
  }  
}
```

Przykładowa walidacja



```
p.js  
pp.js > ...  
if (x === -0) {  
  // doSomething()...  
}  
for (var i = 0; i < 10; i--) {  
}
```

// @ts-check

Oba języki, JavaScript i TypeScript są współcześnie tak mocno powiązane, że dodatkową kontrolę poprawności kodu można osiągnąć przy pomocy dyrektyw sterujących TypeScript, umieszczonych w kodzie JavaScript.

Konkretnie, dyrektywa **@ts-check** w kodzie JavaScript uruchamia analizę kodu specyficzną dla TypeScript i pomaga unikać pewnej liczby błędów w kodzie JavaScript

```
// @ts-check
function doWork( a ) {
    return 123;
}

var result = doWork();

// w linii 8 sygnalizacja błędu wywołania metody toLowerCase
// na wyniku funkcji, który TS wynioskował jako 'number'
console.log( result.toLowerCase() );

// w linii 12 sygnalizacja błędu TS zmiany typu zmiennej
var x = 1;
x = 'foo';
```

Istnieje również możliwość uzyskania podglądu wnioskowanych typów zmiennych, parametrów czy wartości zwracanych z funkcji – służą do tego ustawienia z grupy **Inlay**

```
JS app.js > ...
1 // @ts-check
2 function doWork( a : any ) : number {
3     return 123;
4 }
5
6 var result : number = doWork();
7
8 // w linii 8 sygnalizacja błędu wywołania metody toLowerCase
9 // na wyniku funkcji, który TS wynioskował jako 'number'
10 console.log( result.toLowerCase() );
11
12 // w linii 12 sygnalizacja błędu TS zmiany typu zmiennej
13 var x : number = 1;
14 x = 'foo';
```

String

Ogranicznikami napisów są tradycyjnie `"` i `'''`. W dialekcie ES6 dodano ``` jako ogranicznik szablonów (niżej).

Jeśli chodzi o same napisy, to warto zapoznać się z biblioteką standardową i znać podstawowe metody napisów (m.in. **toLowerCase/toUpper/replace/trim/indexOf/startsWith/endsWith**).

Do formatowania zawartości napisów w Javascript używa się rozszerzenia języka dodającego mechanizm szablonów do literałów (tzw. [string templates](#)):

```
var n = 1, m = 2;
var string = `szablon z wartościami ${n}, ${m} i ${n+m}`;

console.log( string );
```

Date

Javascript w bibliotece standardowej posiada wzorowany na Javie interfejs operacji na datach. W tym przypadku również warto przestudiować dokumentację dostępnych metod oraz zapoznać się z tym jak określono standard [ISO 8601](https://www.iso.org/standard/52865.html) i jak konwertować daty w javascript z/na taki właśnie format.

Użycie jedynie takiego formatu przy wymianie danych gwarantuje interoperacyjność z innymi technologiami.



Rysunek 5 Za <https://xkcd.com/1179/>

getter/setter – implementacja właściwości ze skutkami ubocznymi

Na wzór wielu języków programowania, JavaScript posiada lukier syntaktyczny do definiowania właściwości (properties), wymagający określenia akcesorów [get](#) i [set](#).

```
var foo = {
  _i : 0,
  get bar() {
    return foo._i;
  },
  set bar(i) {
    foo._i = i;
  }
}

console.log( foo.bar );

foo.bar = 5;
```

```
console.log( foo.bar );
```

Możliwe jest ogólniejsze podejście, użycie **Object.defineProperty** i tak zwanego [Property Descriptora](#), za pomocą którego można dodać do obiektu dowolną wartość (pole, właściwość, funkcję):

```
var foo = {
  _i : 0,
  get bar() {
    return foo._i;
  },
  set bar(i) {
    foo._i = i;
  }
}

Object.defineProperty( foo, 'qux', {
  get : function() {
    return 17;
  }
});

Object.defineProperty( foo, 'baz', {
  value : function() {
    return 34;
  }
});

console.log( foo.qux );
console.log( foo.baz() );
```

O ile do istniejącego obiektu można dodawać pola/metody za pomocą zwykłej składni (operatora `.` lub `[]`) o tyle dodanie do istniejącego obiektu właściwości (get/set) jest możliwe tylko w wyżej zademonstrowany sposób.

W trakcie wykładu powiemy o dodatkowych atrybutach deskryptora (**writable**, **enumerable**, **configurable**).

Tablice

Tablice to obiekty o indeksach numerycznych, zoptymalizowane pod kątem szybkości dostępu oraz zużycia pamięci. W przeciwieństwie do wielu języków, Javascript nigdy nie wyrzuca wyjątków typu „out of bounds” ponieważ dostęp do komórki tablicy która nie posiada wartości zwraca wartość **undefined**.

Z uwagi na omówione wcześniej dynamiczne konwersje między typami, mylące może być więc zastosowanie idiomatycznej konstrukcji dla klauzuli **if**:

```
var a = [];  
  
a[100] = 1;  
  
if ( a[100] ) {  
    console.log( 'jest element' );  
} else {  
    console.log( 'nie ma elementu' );  
}
```

ponieważ takie podejście niepoprawnie rozpozna sytuację gdy elementem tablicy o wskazanym indeksie jest cokolwiek konwertowalnego do **false**:

```
var a = [];  
  
a[100] = 0;  
  
if ( a[100] ) {  
    console.log( 'jest element' );  
} else {  
    console.log( 'nie ma elementu' );  
}
```

Jest to jeden z popełnianych przez nowicjuszy błędów i pierwszy przypadek w którym można zauważyć że zasadne jest posiadanie zarówno **null** jak i **undefined** w języku.

Poprawne jest testowanie za pomocą

```
if ( a[100] !== undefined )
```

lub

```
if ( typeof a[100] !== 'undefined' )
```

(przypominamy że operator **typeof** zwraca wartości literalne).

Na wykładzie omówimy również podstawowy interfejs komunikacyjny tablic, wprowadzając wcześniej [składnię typu lambda](#) dla funkcji (o funkcjach będzie mowa w kolejnej sekcji)

- [slice](#)
- [splice](#)
- [filter](#)
- [map](#)
- [reduce](#)
- [join](#)

Omówimy też **enumerowanie** tablic za pomocą konstrukcji [for-of](#) i [for-in](#). Na kolejnym wykładzie dowiemy się jak umożliwić enumerowanie zawartości dowolnego obiektu za pomocą for-of.

Wyjątki

Na wykładzie omówimy mechanizm [wyjątków](#).

```
try {  
    throw new Error('wyjątek');  
}  
catch ( e ) {  
    console.log( e.message );  
}
```

Dialekty zawężające

Oryginalne sformułowanie składni Javascript może w przypadkach brzegowych prowadzić do nieścisłości. Stąd w wersji 5 języka dodano możliwość opcjonalnego włączenia ograniczenia składni do tzw. [trybu ścisłego](#) (Strict mode). W trakcie wykładu zobaczymy przykłady.

Innym dialektem którego istnienie warto odnotować (aczkolwiek niekoniecznie posługiwać się nim na co dzień) jest [asm.js](#). Ten wzmiankowany już na wykładzie dialekt jest używany zwykle w sytuacjach gdy Javascript powstaje jako kompilat języka wyższego poziomu i ma na celu dodanie optymalizacji wydajnościowych. Asm.js jest domyślnie dostępny we współczesnych przeglądarkach.

JSON

[Javascript Object Notation](#) to interoperacyjny format wymiany danych o składni wywiedzionej z Javascript. Javascript posiada wsparcie dla JSON, na wykładzie zademonstrujemy funkcje **JSON.stringify** i **JSON.parse**.