

Wybrane elementy praktyki projektowania oprogramowania
Wykład 02/15
JavaScript, podstawy języka (1)

Wiktor Zychła 2018/2019

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
Język JavaScript	11

Narzędzia

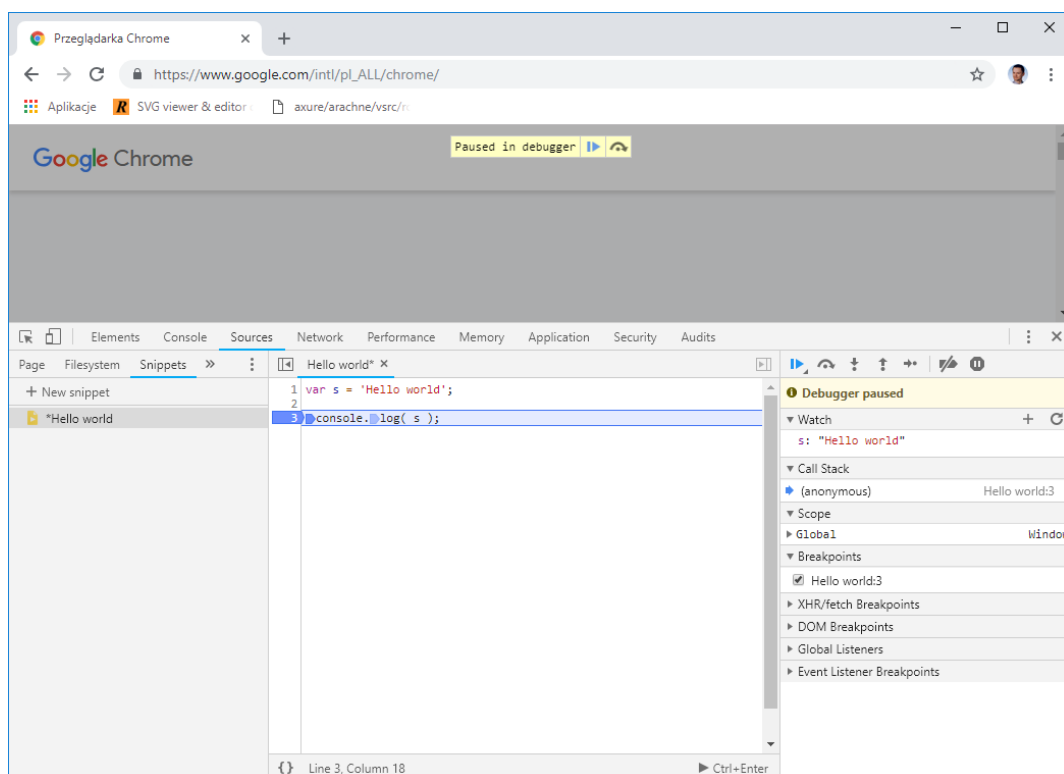
Środowisko [node.js](#) należy zainstalować w wersji 10.X. Edytor [Visual Studio Code](#) należy zainstalować w wersji aktualnej.

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.

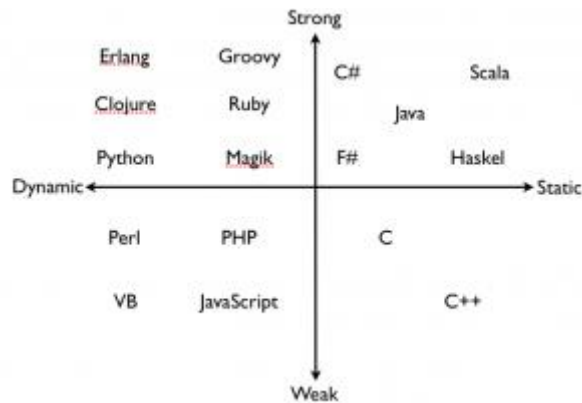
Tego sposobu pisania kodu można używać do szybkiego testowania / prototypowania, bez konieczności posiadania innych narzędzi deweloperskich. W większości innych przeglądarek konsole deweloperskie owszem posiadają możliwość debugowania wczytanych skryptów Javascript ale nie mają interaktywnego edytora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

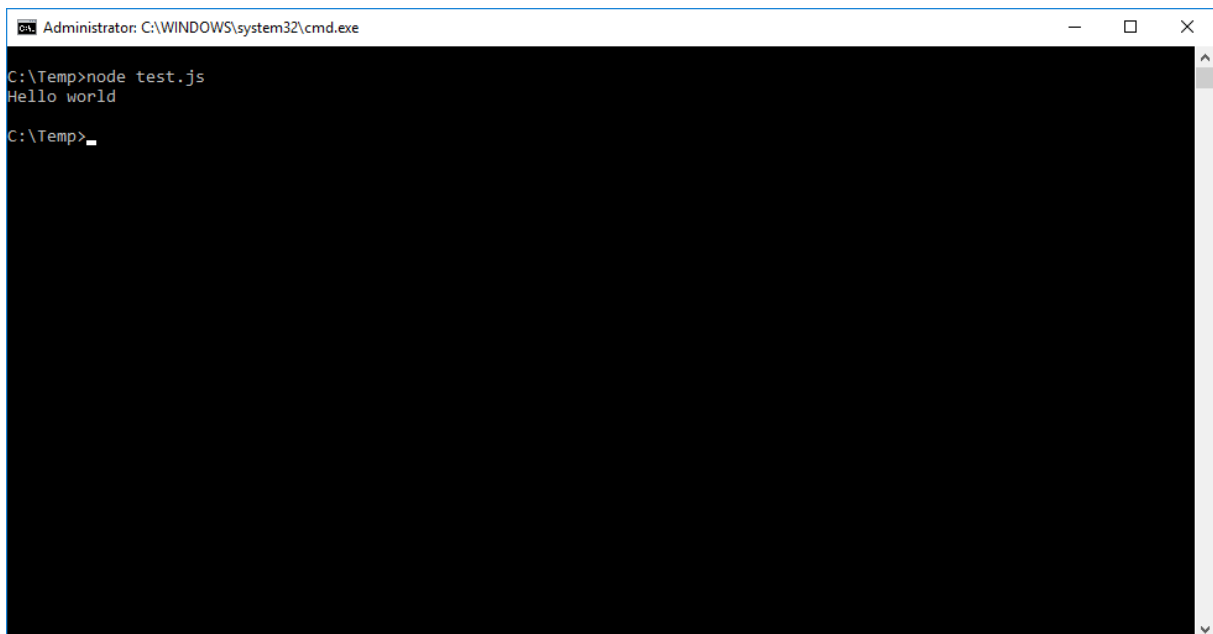
- `node.exe` w `\Program Files\nodejs` w Windows
- `node` w `/usr/bin/nodejs` w Linux

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 shows the following text:

```
C:\Temp>node test.js
Hello world
C:\Temp>
```

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

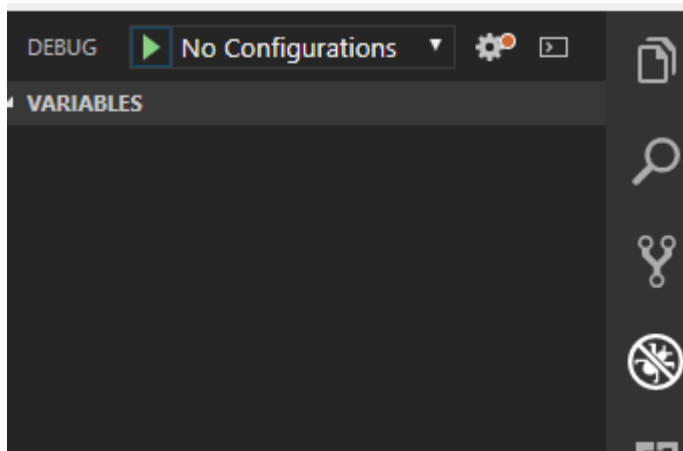
- Windows - `%APPDATA%\Roaming\Code\User\settings.json`
- Linux - `$HOME/.config/Code/User/settings.json`

W szczególności warto zwrócić uwagę m.in. na ustawienia

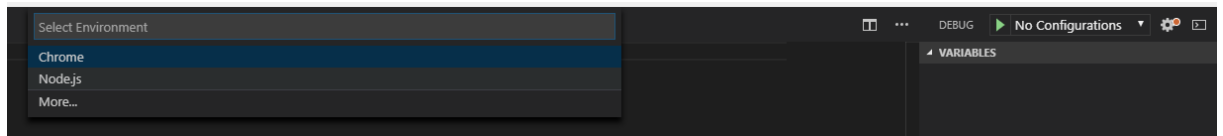
- Lokalizacji sideBara – *workbench.sideBar.location*
- Widoczności tzw. minimapy kodu – *editor.minimap.enabled*

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

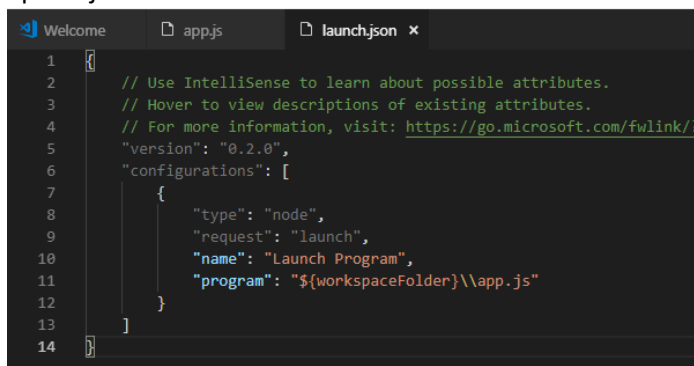
1. Wybrać File/Open Folder, wskazać nowy folder
2. Dodać plik – moduł startowy aplikacji, np. **app.js**
3. Otworzyć zintegrowane okno terminala
4. Wydać polecenie **npm init -y** inicjujące plik **package.json** w przyszłości zapisujący zależności do zewnętrznych pakietów (dobrze wyrobić sobie nawyk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



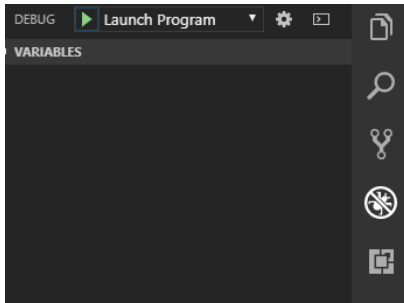
6. Odczekać chwilę na zorientowanie się przez VSC w dostępnych konfiguracjach – pojawią się one na liście wyboru i ich liczba zależy od zainstalowanych rozszerzeń



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

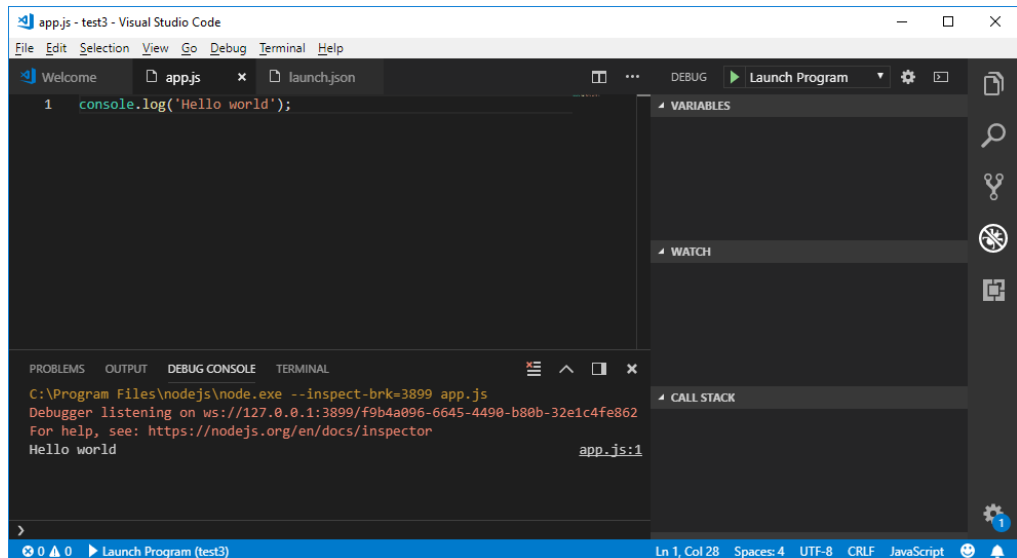


9. Uzpełnić plik z kodem źródłowym (app.js) o jakiś niepusty kod
10. Uruchamiać aplikację przez F5 lub ikonę uruchomienia na zakładce Debug

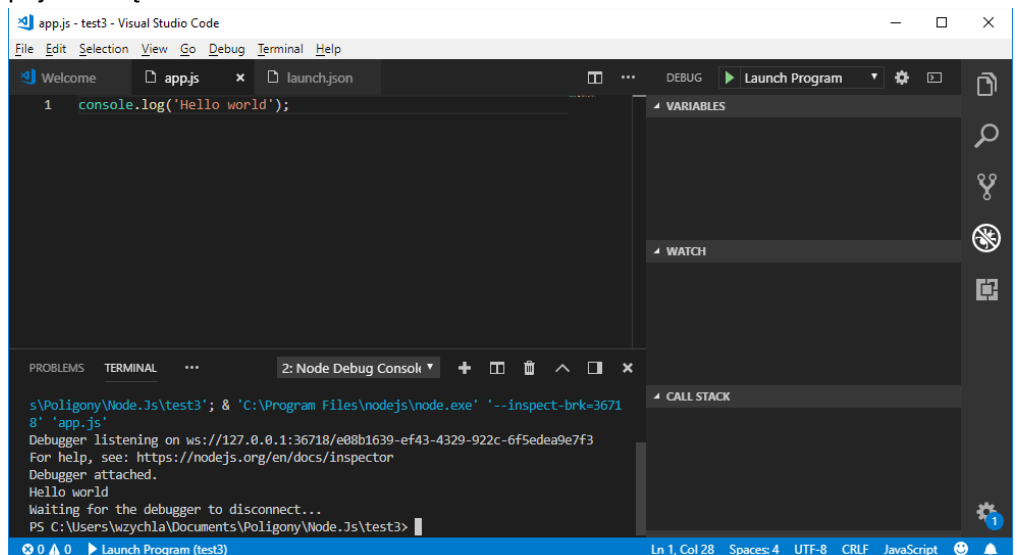


11. Jeśli program wypisuje coś na konsoli, to są dwie możliwości

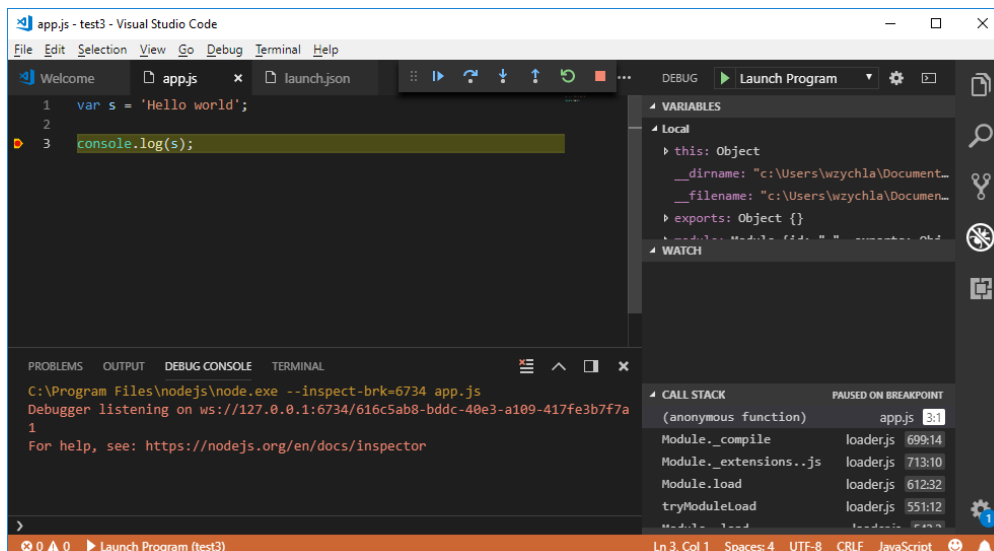
- a. Domyślnie jest to konsola trybu Debug VSC, widoczna po wybraniu View/Debug Console



- 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



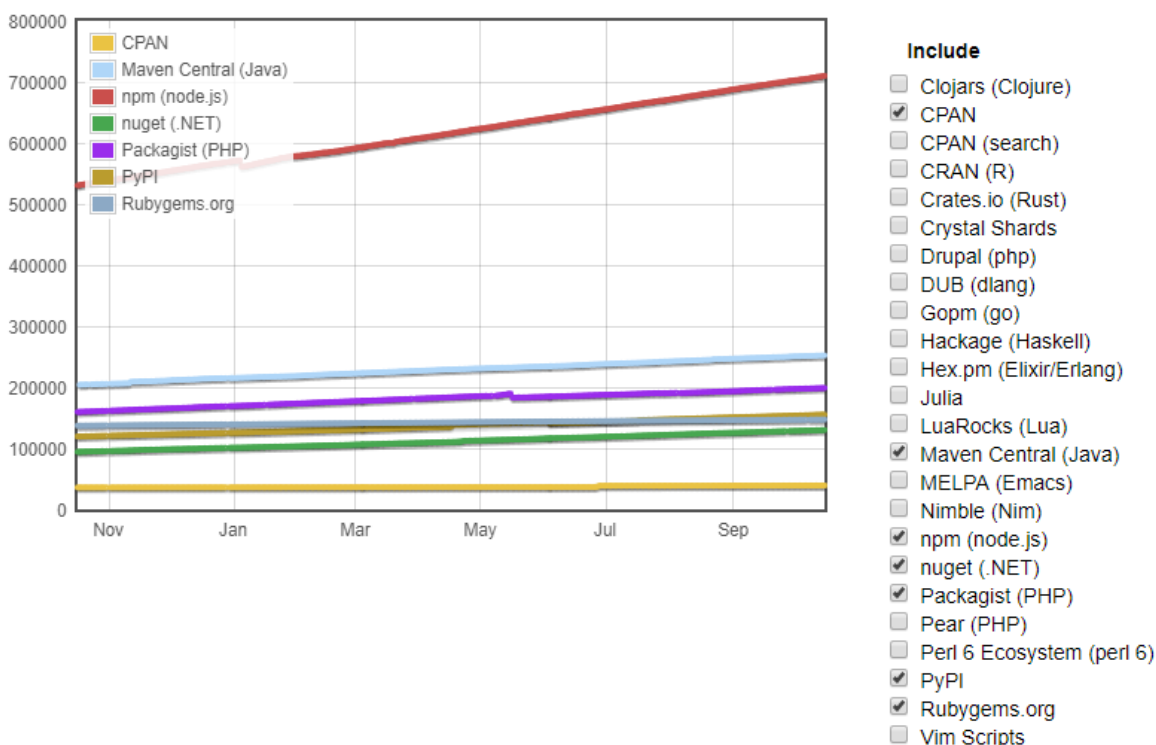
12. Zweryfikować czy działa tryb śledzenia wykonania programu – ustawianie pułapek i podgląd/edycja zmiennych lokalnych i globalnych



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 pisania tego dokumentu repozytorium npm zawiera ponad 700 tysięcy pakietów.

Module Counts



Rysunek 2 Liczebności repozytoriów pakietów, stan na październik 2018

Menedżer npm ma dwa tryby instalacji pakietu – instalacja globalna 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

dotkownie 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.)

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

```
var s = 'Hello world';
```

s.

- anchor
- big
- blink
- bold
- charAt
- charCodeAt
- codePointAt
- concat
- endsWith
- fixed
- fontcolor
- fontSize

(method) String.anchor(name: string): string

Returns an <a> HTML anchor element and sets the name attribute to the text value

@param name

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**:

```
var _ = require('lodash');
```

_.

- endsWith
- entries
- entriesIn
- eq
- escape
- escapeRegExp
- every
- extend
- extendWith
- fill
- filter
- find

(method) LoDashStatic.filter(collection: string, predicate?: _.StringIterator<boolean>): string[] (+4 overloads)

Iterates over elements of collection, returning an array of all elements predicate returns truthy for. The predicate is invoked with three arguments: (value, index|key, collection).

@param collection — The collection to iterate over.

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. 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 );
```

4. Typy opakujące dla typów prostych – Boolean, String, Number, mechanizm **boxing** i **unboxing**
5. Odwołania do składowych . vs [], automatyczna konwersja operandu indeksowania do string
6. Konwersje między typami

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. [Wat?](#)

```
var p = {};
console.log( p );           // Object
console.log( !p );         // false
console.log( !p+[] );     // 'false'
console.log( [] );        // []
console.log( +[] );       // 0
console.log( (!p+[])[+[]] ); // 'f'
```

A

```
(![]+[]) [+[]]+(![]+[]) [+!+[]]+(![]+[]) [+!+[]]+(+!+[])+(![]+[]) [+!+[]]+(+!+[]);
```

??

8. Operator [typeof](#) i [instanceof](#)

9. Operator [==](#) i [===](#)

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

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