ZAAWANSOWANE PROGRAMOWANIE W JĘZYKU SKRYPTOWYM

Waldemar Korłub

Aplikacje i Usługi Internetowe
KASK ETI Politechnika Gdańska

Agenda

- Zaawansowane programowanie w przeglądarce
- Bogate aplikacje klienckie
- Modelowanie przepływu sterowania: frameworki MVC w przeglądarce
  - Widoki aplikacji jako projekcje szablonów
  - Model danych
  - Kontrolery reagujące na akcje użytkownika
- Wiązanie danych
- Wstrzykiwanie zależności
- Komunikacja z aplikację serwerową

Rynek aplikacji

Aplikacje desktopowe

- Od lat na rynku systemów desktopowych aplikacje natywne tracą na znaczeniu
  - Na rzecz WWW!
- Dlaczego strona internetowa jest lepsza od natywnej aplikacji okienkowej?

Aplikacje desktopowe

- Dlaczego strona internetowa jest lepsza od natywnej aplikacji desktopowej?
  - Brak potrzeby instalacji
    - Szybszy dostęp dla nowych użytkowników
    - Dostęp z każdego miejsca
    - Nieufność do aplikacji natywnych związana z wirusami
  - Łatwe publikowanie aktualizacji
    - Wystarczy wdrożyć nową wersję na serwer
  - Współczesne przeglądarki desktopowe są *wystarczająco dobre*
  - Współczesne łącza są szybkie

Aplikacje desktopowe

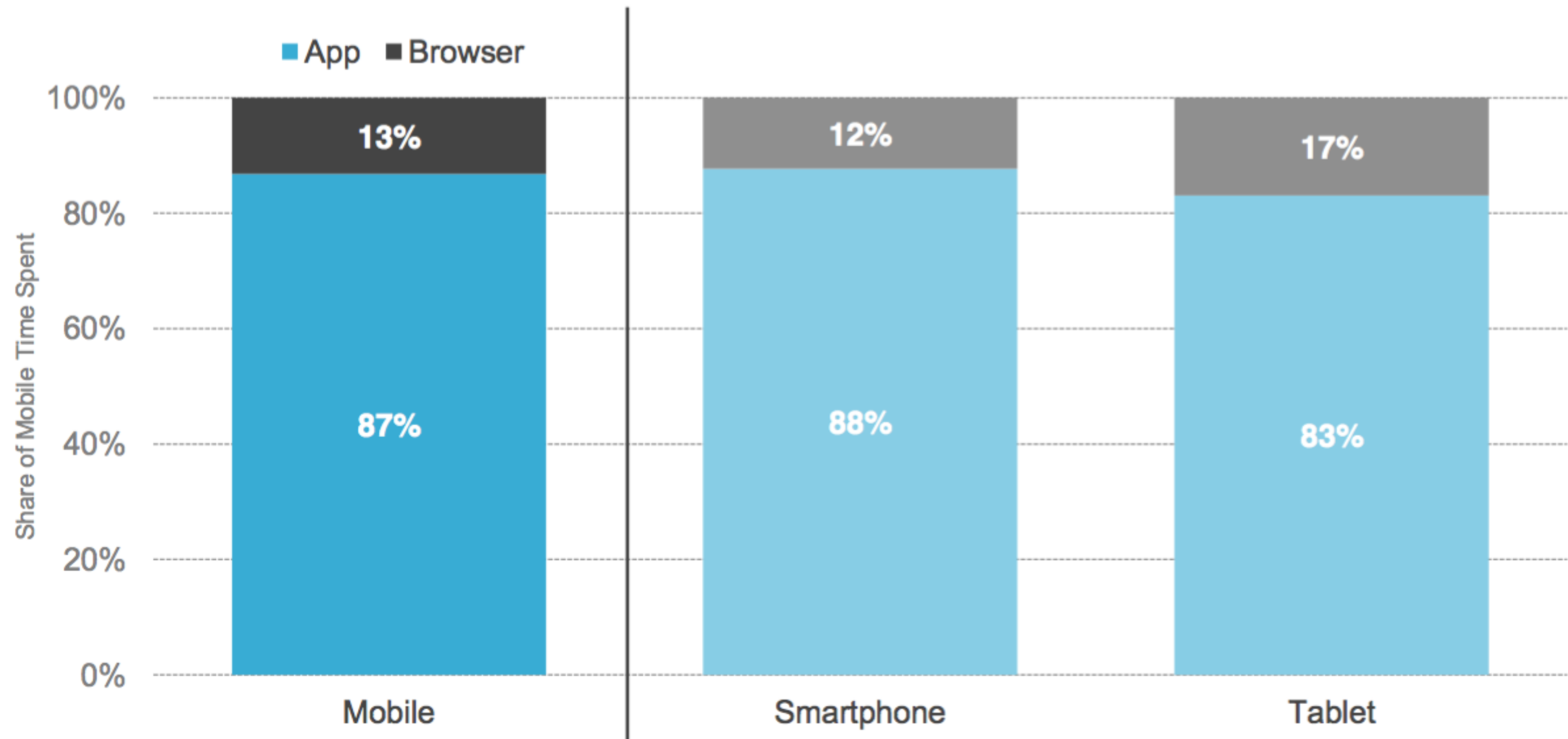
- Dlaczego strona internetowa jest lepsza od natywnej aplikacji desktopowej?
  - Swoboda w kreowaniu publikowanych treści
  - Niski próg wejścia
    - HTML, CSS, PHP…
    - Darmowe narzędzia dla deweloperów
  - Naturalny interfejs dla usług *w chmurze*

Aplikacje desktopowe

- Coraz więcej aplikacji dedykowanych trafia do *chmury*
  - Aplikacje do prowadzenia księgowości
  - Aplikacje dla recepcji w przychodni lekarskiej
    - Jedna aplikacja będąca interfejsem dla pacjentów i pracowników
  - Aplikacje do obsługi dziekanatu
    - Moja PG :)
  - IDE w przeglądarce
- Oczywiście cały czas istnieją wymagania wymuszające wykorzystanie aplikacji natywnych
  - O nich nieco później…

Aplikacje mobilne

- Na rynku urządzeń mobilnych trend jest odwrotny



Aplikacje mobilne

- Dlaczego natywna aplikacja mobilna jest lepsza od mobilnej strony internetowej?

Aplikacje mobilne

- Dlaczego natywna aplikacja mobilna jest lepsza od mobilnej strony internetowej?
  - Mobilne przeglądarki nie są tak wydajne jak desktopowe
    - 8-rdzeniowy CPU (big.LITTLE) nie pomoże, gdy na stronie osadzony jest plik wideo i kilka reklam…
  - Mobilne strony – nawet te responsywne – nie są wygodne w obsłudze na dotykowym ekranie
    - …i odbiegają od *UX guidelines* dla systemów mobilnych

Aplikacje mobilne

- Dlaczego natywna aplikacja mobilna jest lepsza od mobilnej strony internetowej?
  - Ograniczony dostęp do natywnych możliwości urządzenia z poziomu mobilnej strony
    - Kamera, akcelerometr itd.
      - To stopniowo ulega zmianie
    - ARM NEON, RenderScript
  - Brak możliwości działania w tle
    - To również ulega zmianie

Współczesny rynek aplikacji

- Dedykowane aplikacje dla smartphonów powstają, aby zoptymalizować działanie pod kątem:
  - Wydajności
  - Interfejsu użytkownika
  - Dostępu do specyficznych możliwości platformy
    - Akcelerometr, kamera, GPS
    - Integracji z systemem

Dedykowane aplikacje desktopowe

- Dedykowane aplikacje desktopowe powstają, aby zoptymalizować działanie pod kątem:
  - Wydajności
  - Interfejsu użytkownika, łatwości wprowadzania danych
    - Klawiatura! i ewentualnie myszka…
    - Znajomo wyglądający interfejs
    - Liczba informacji widocznych na ekranie
    - Nie obsługa dotykowa i nie ogromne kafle
  - Dostępu do specyficznych możliwości platformy
    - Karta graficzna, CUDA
    - Porty dla dedykowanych urządzeń
      - np. urządzenia pomiarowe

Współczesny rynek aplikacji

- Najczęściej spotykany model:

aplikacja internetowa

+

klient dla smartphonów/tabletów

- Logika biznesowa w postaci usług sieciowych – łatwe współdzielenie

Aplikacje internetowe

- Przed erą mobilną
  - Cała mechanika witryny po stronie serwera
  - Przeglądarka wyświetla gotowe dokumenty HTML otrzymane od serwera
  - Ograniczona dynamika w przeglądarce
- Początek ery mobilnej
  - Serwer generuje dokumenty HTML dla przeglądarki, a dla aplikacji mobilnej udostępnia API (np. dane w formacie JSON)
  - Aplikacja serwerowa udostępnia swoją funkcjonalność na dwa różne sposoby – kłopotliwe, problemy z bezpieczeństwem
- A w międzyczasie…
  - Wzrost wydajności przeglądarek
  - Możliwość obsłużenia skomplikowanych interakcji

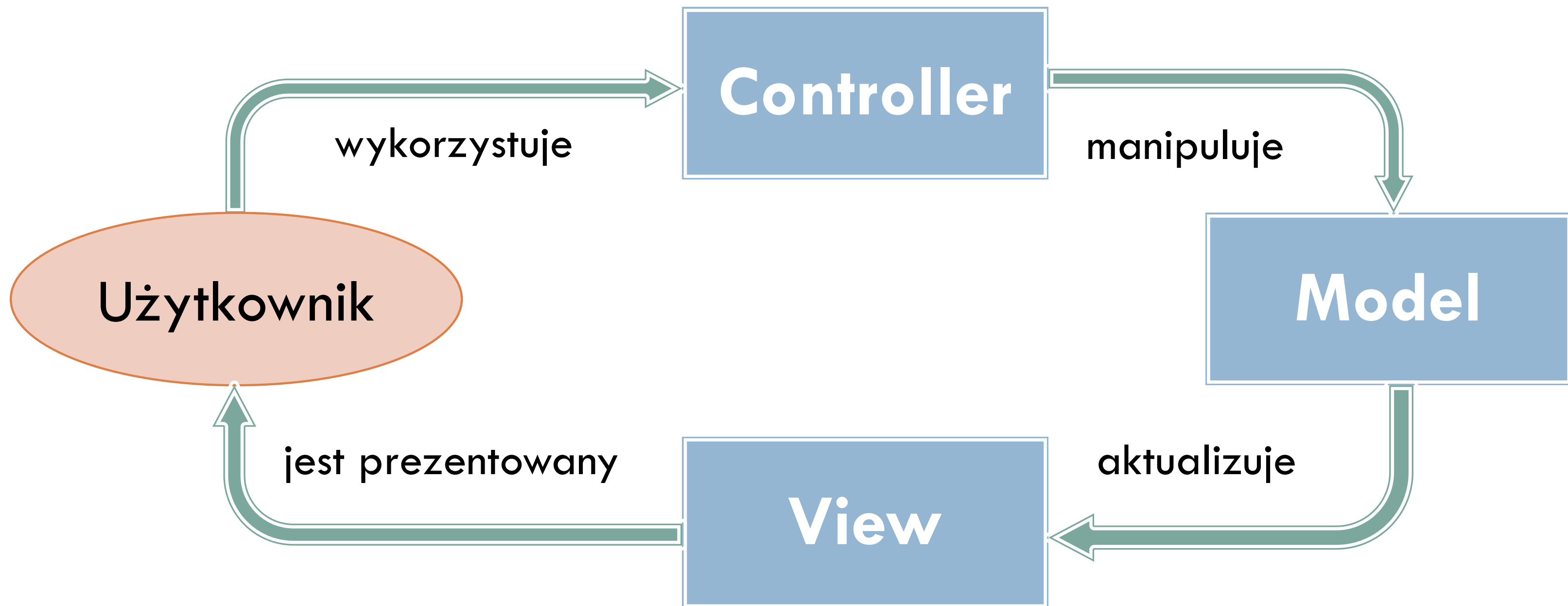
Aplikacje internetowe

- Skoro aplikacja serwerowa i tak musi udostępniać API dla klienta mobilnego, to czy klient w przeglądarce nie mógłby używać tego samego API?
- Te same punkty wejścia dla wszystkich aplikacji klienckich
  - Jednolite mechanizmy logowania i kontroli dostępu
  - Aplikacja serwerowa skupia się na logice biznesowej
  - Za interakcje z użytkownikiem odpowiedzialne aplikacje klienckie
  - Przejrzysty podział odpowiedzialności: front-end lub back-end deweloper

Aplikacje internetowe: MVC w przeglądarce

- Pojawienie się nowych frameworków, które ułatwiają budowanie złożonych interakcji w przeglądarce
- MVC: Model View Controller
  - Wzorzec architektoniczny modelujący przepływ sterowania w *interfejsie użytkownika*
  - Oryginalnie zaproponowany przez Trygve Reenskaug w latach 70. dla graficznych interfejsów użytkownika
  - Dzieli kod aplikacji na trzy części w celu oddzielenia wewnętrznych reprezentacji danych od sposobu, w jaki są one przedstawiane użytkownikowi

MVC



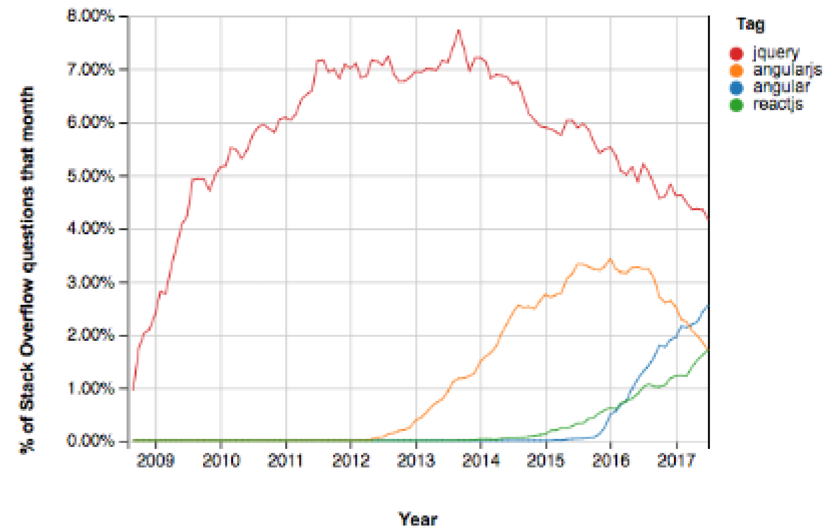
- Kontroler – aktualizuje model danych w odpowiedzi na akcje użytkownika podejmowane w interfejsie
- Model danych – dane, na których operuje aplikacja
- Widok – reprezentacja danych widoczna dla użytkownika

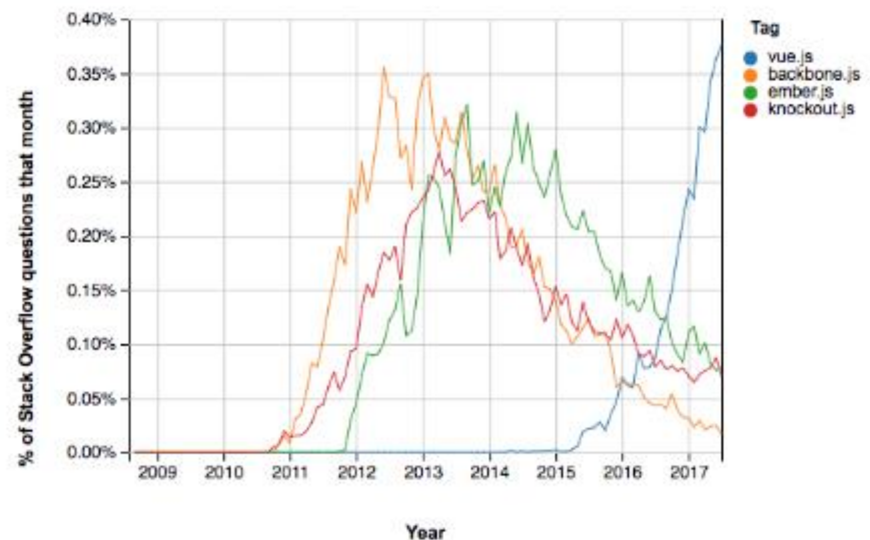
MVC w przeglądarce

- Frameworki definiują w jaki sposób zorganizować kod projektu
  - Jak podzielić logikę interakcji
  - Jak powiązać logikę z widokami
- Dostarczają funkcje narzędziowe do wykonywania typowych operacji
  - np. wywoływanie usług po stronie serwera

MVC w przeglądarce

- Backbone.js
- Knockout.js
- Ember.js
- Vue.js
- Polymer
- Angular
- React





Angular

Framework Angular

- Angular 1.x – 2009
- Angular 2.x – wrzesień 2016
  - Framework został przepisany od zera
  - Język TypeScript
  - Wersja 2.x znacznie bardziej przypomina framework React niż Angulara 1.x
    - Migracja 1.x → 2.x wymaga wielu modyfikacji
- Angular 4, 5… – 2017
  - Wersjonowanie semantyczne
  - Wersja 3.x pominięta ze względu na rozbieżność wersji poszczególnych komponentów →
  - Google planuje nowe wydanie średnio co pół roku

@angular/core	v2.3.0
@angular/compiler	v2.3.0
@angular/compiler-cli	v2.3.0
@angular/http	v2.3.0
@angular/router	**v3.3.0**

Wersjonowanie semantyczne

major.minor.patch

2.7.3

- Patch – poprawki błędów kompatybilne z poprzednią wersją, np. 2.7.4
- Minor - nowe funkcjonalności kompatybilnr z poprzednimi wersjami, np. 2.8.0
- Major – zmiany niekompatybilne wstecznie, np. 3.0

Wersje frameworka a aplikacja

- Jeśli chcemy mieć dostęp do nowych funkcji oraz poprawek bezpieczeństwa, musimy migrować naszą aplikację na nowsze wersje frameworka
- Patch, minor – zmiany kompatybilne wstecznie
  - Kod nasz naszej aplikacji nie wymaga modyfikacji do poprawnego działania na nowej wersji frameworka
- Major – wymaga zmian w naszym kodzie, aby dokonać migracji
  - Wymaga roboczogodzin
  - Te zmiany nie muszą być bolesne dla deweloperów
    - Autorzy Angulara nie planują przepisywać frameworka do zera *kolejny raz*
- Każde kolejne wydanie wprowadza nowe zmiany
  - Jeśli zrezygnujemy z migracji później będzie już tylko trudniej

TypeScript

Co jest nie tak z JavaScriptem?

- Dynamiczne typowanie
  - Brak weryfikacji typów na etapie wytwarzania kodu – dopiero na etapie uruchomienia w przeglądarce
  - Ograniczone wsparcie narzędziowe (IDE nie jest w stanie prawidłowo podpowiadać dostępnych metod obiektów)
    - Kontrakty funkcji i obiektów opisane w dokumentacji, a nie w kodzie źródłowym
      - Dokumentacja jest często nieaktualna, nikt nie lubi jej pisać
  - Dynamiczne typowanie jest wygodne w małych projektach
  - W dużych, wieloosobowych projektach statyczne typowanie często ułatwia pracę
    - Kontrola typów na etapie kompilacji/transpilacji, lepsze wsparcie narzędziowe

Co jest nie tak z JavaScriptem?

- Brak wsparcia dla modułów (pakietów, zasięgów) na poziomie języka
  - Idiom IIFE – immediately-invoked function expression – wprowadzanie zasięgu leksykalnego przy użyciu zasięgu funkcji
- Brak granularnej kontroli nad widocznością pól obiektów (np. private, package, protected, public)

- W efekcie: JavaScript słabo skaluje się w dużych projektach

TypeScript

- Statycznie typowany język transpilowany do języka JavaScript
  - Deweloper pracuje w TypeScripcie, przeglądarka otrzymuje zrozumiały dla siebie JavaScript
- Oferuje kompatybilność z najnowszymi wersjami JavaScriptu (ECMAScript 2016)
- Transpilacja do starszej wersji np. ECMAScript 5
  - Według wyboru dewelopera: tsc --target
  - Problem wersji standardu i wersji przeglądarek

ECMAScript

- 1997 – ECMAScript 1
- 1998 – ECMAScript 2 (editorial changes only)
- 1999 – ECMAScript 3
- ECMAScript 4 – nie doczekał się publikacji
- 2009 – ECMAScript 5 (Internet Explorer 11)
- 2011 – ECMAScript 5.1 (editorial changes only)
- 2015 – ECMAScript 6
- 2016 – ECMAScript 7

Rynek przeglądarek

- W przeszłości nowe wersje przeglądarek wydawane były raz na rok lub raz na kilka lat
  - np. Internet Explorer: 8 – 2009; 9 – 2011; 10 – 2012; 11 – 2013
- Długi cykl wydawniczy opóźnia wprowadzanie nowych specyfikacji
- Google zapoczątkowało trend wydawania nowych wersji tak często, że nikogo to już nie obchodzi
  - Google Chrome: 1.0 – 2008; 55 – 2016; 65 – 01.2018 (średnio 6-7 wydań/rok)
  - Inkrementalne zmiany, szybkie wprowadzanie nowych funkcji
  - Automatyczne instalowanie aktualizacji (przezroczyste dla użytkownika)
  - Inne przeglądarki podążają za tym trendem: Firefox 58, Opera 51, Edge 41
    - …ale nie wszystkie: Safari 11

Rynek przeglądarek

StatCounter Global Stats
Browser Market Share Worldwide from Jan 2009 to July 2017

70%
56%
42%
28%
14%
0%

Jan 2010
Jan 2011
Jan 2012
Jan 2013
Jan 2014
Jan 2015
Jan 2016
Jan 2017

Chrome
IE
Firefox
Safari
Opera
Android
UC Browser
Nokia
Samsung Internet
Other (dotted)

Problem z Internet Explorerem 11

- Kolejnego wydania Internet Explorera (IE 12) nie będzie
  - Microsoft rozwija przeglądarkę Edge
- Edge jest dostępny tylko dla Windows 10
  - Brak back-portu dla Windows 7
- Windows 7 cały czas funkcjonuje na wielu komputerach
  - Szczególnie na rynku korporacyjnym
- Wielu klientów korporacyjnych wymaga kompatybilności z IE 11

TypeScript

- Umożliwia korzystanie z istniejących bibliotek JavaScriptowych
  - Poprawny kod JavaScript jest też poprawnym kodem TypeScript
    - *Declaration Files* – zawierają deklaracje typów dla bibliotek zaimplementowanych w JavaScriptcie, np. http://definitelytyped.org/
- Wygenerowane pliki .js są (w miarę) czytelne dla dewelopera
  - Strategia wyjścia
- Wersja 1.0: kwiecień 2014

TypeScript – podstawowe użycie

- Instalacja:
  $ npm install -g typescript
- Transpilacja pliku:
  $ tsc file.ts
- Obserwowanie zmian:
  $ tsc --watch file.ts
- Popularne IDE oferują integrację z narzędziami

Przegląd składni

Typy proste

- Boolean:

```
let isDone: boolean = false;
```

  - Po transpilacji:

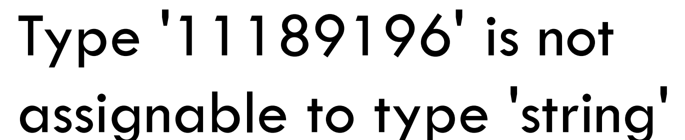
```
var isDone = false;
```

- Number:

```
let age: number = 42;
let color: number = 0xf00dcc;
```

- String:

```
let color: string = "blue";
color = 'red';
color = 0xaabbcc;
```



Typy proste

- String – szablony:

```
let username: string = `world`;
let greeting: string = `Hello, ${ username }!`;
```

  - Po transpilacji:

```
var username = "world";
var greeting = "Hello, " + username + "!";
```

- Tablice:

```
let list1: number[] = [42, 36, 28];
let list2: Array<number> = [27, 45, 19];
```

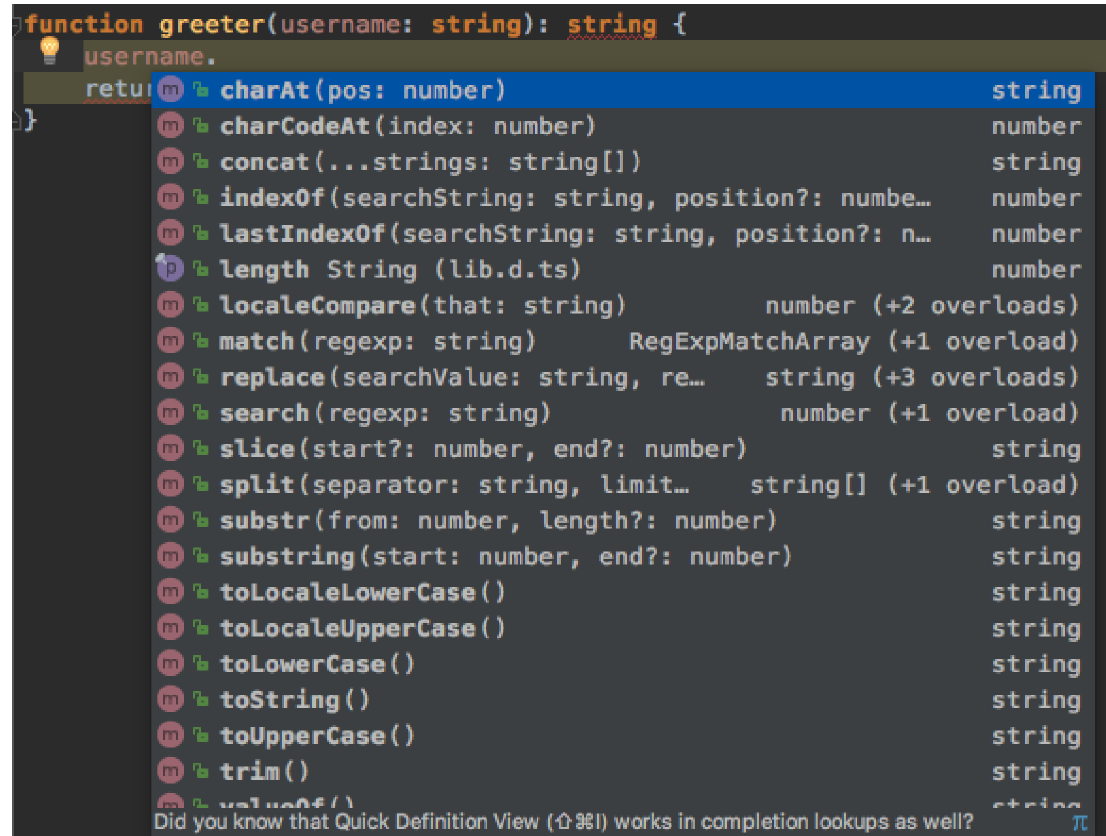
  - Wszystkie elementy tablicy tego samego typu

Typy proste

- Krotki:
  ```
  let x: [string, number] = ["Waldemar", 22];
  ```
- Typy wyliczeniowe:
  ```
  enum Color {Red, Green, Blue};
  let c: Color = Color.Green;
  ```
- Typ Any:
  ```
  let zmienna: any = 42;
  zmienna = "Imiesław";
  zmienna = false;
  ```
- Typ Void:
  - Tylko null lub undefined
  - Używany w deklaracjach typów zwracanych funkcji

Funkcje z deklaracjami typów


- ```
  function greeter(username: string): string {
      return "Hello, " + username;
  }
  ```
- Podpowiadanie składni na podstawie typu →


  ```

Funkcje bez deklaracji typów – JavaScript

- **function** greeter(username) {
    return "Hello, " + username;
  }
- Podpowiadanie składni →

```
function greeter(username) {
    username.
    retur
}
constructor (Object)                                            Function
hasOwnProperty([PropertyKey] propertyName) (Ob…                 boolean
isPrototypeOf([Object] o) (Object)                              boolean
propertyIsEnumerable([PropertyKey] propertyNam…                 boolean
prototype (Object)                                              Object
toLocaleString([string|string[], optional] loca…                string
toSource() (Object)                                             string
toString() (Object)                                             string
unwatch([string] prop) (Object)                                 string
valueOf() (Object)                                              *
watch([string] prop, [Function] handler) (Objec…                string
username (test.ts)                                              string
$1 (RegExp)                                                     string
$2 (RegExp)                                                     string
$3 (RegExp)                                                     string
$4 (RegExp)                                                     string
$5 (RegExp)                                                     string
__defineGetter__([string] propertyName, func) (Object)
__defineSetter__([string] propertyName, func) (Object)
__DIR__ (Nashorn.js)                                            string
Not all variants are shown, please type more letters to see the rest
```

Parametry funkcji

- Standardowo wszystkie parametry funkcji są wymagane
  - Jak jest w przypadku JavaScriptu?
- Parametry opcjonalne można oznaczyć znakiem ?:

```
function greeter(username?: string): string {
    if(username) {
        return "Hello, " + username;
    } else {
        return "Hello!"
    }
}
```

Parametry funkcji

- Wartości domyślne definiowane na liście argumentów:

```
function greeter(username: string = "world"): string
{
        return "Hello, " + username;
}
```

Zmienna liczba argumentów


```
function greeter(firstName: string,
                 ...otherNames: string[]): string {
    return "Hello, "+username+" "+otherNames.join(", ");
}

//wywołanie:
greeter("world", "Waldemar", "Michał");
```

Programowanie obiektowe

- Jest to styl programowania bazujący na pojęciu klasy i obiektu
  - A jak jest w JavaScriptcie?
- Upraszcza proces projektowania, tworzenia i testowania systemów informatycznych
- Każdy element świata rzeczywistego w programie odzwierciedlony jest w postaci klasy, która łączy stan opisywanego przedmiotu z jego zachowaniem.

Klasa

- Definiuje pewien wycinek tego świata za pomocą jego własności
- W określaniu przynależności elementu do klasy przede wszystkim istotne jest, że element posiada daną własność
- Klasa stanowi definicję, czyli tylko określa pewne elementy, ale sama w świecie elementów formalnie nie istnieje
- Klasę definiuje jej nazwa a nie zbiór własności

Obiekt

- Jest to konkretny element świata, ale widziany jako należący do pewnej klasy, stąd nazywany jest instancją klasy
- Dla obiektu własności istotne z punktu widzenia klasy powinny mieć już konkretne wartości
- Pole obiektu to własność klasy z określoną konkretną wartością i przypisaną do konkretnego obiektu

Metody klas

- Jest odpowiednikiem czynności, którą obiekt danej klasy może wykonać
- z punktu widzenia programisty zawiera w sobie kod, który zadziała w momencie jej wywołania.
- Jest odpowiednikiem procedury z programowania strukturalnego.
- Metoda może pobierać parametry oraz zwracać jakiś wynik.

Konstruktor

- Metoda wywoływana automatycznie w momencie tworzenia instancji klasy
- Konstruktor służy do wstępnego ustawienia pól, wykonania jakiś czynności inicjalnych itp.
- Jak do każdej innej metody, tak samo i do konstruktora można przekazać parametry

Przykład klasy


```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
let powitanie = greeter.greet();
```

Dziedziczenie

- Możliwe jest tworzenie hierarchii klas umożliwiającej specjalizowanie opisu danego elementu rzeczywistości.
- Klasa dziedzicząca przejmuje wszystkie własności i metody klasy, po której dziedziczy, o ile ich poziom widoczności jest inny niż private.
- Klasa dziedzicząca może jednak przesłaniać metody klasy nadrzędnej dostarczając własnej implementacji.

Dziedziczenie – klasa bazowa


```
class Animal {
    public name: string;

    public constructor(theName: string) {
        this.name = theName; }

    public move(distanceInMeters: number) {
        console.log(
            `${this.name} moved ${distanceInMeters}m.`);
    }
}
```

Klasa potomna


```
class Snake extends Animal {
    constructor(name: string) { super(name); }

    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
```

Inna klasa potomna


```
class Horse extends Animal {
    constructor(name: string) {
        super(name);
    }

    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}
```

Poziomy dostępu

- Public – domyślny
  - Pola i metody widoczne dla innych klas
- Protected
  - Widoczne tylko w obrębie hierarchii dziedziczenia
- Private
  - Dostępne tylko w obrębie danej klasy

Akcesory

- Nie zaleca się udostępniania bezpośredniego dostępu do pól klasy

```
class Animal {
    private _name: string;

    get name(): string {
        return this._name;
    }

    set name(newName: string) {
        this._name = newName;
    }
}
```

Klasy abstrakcyjne

- Możemy sobie wyobrazić klasę, która nie ma i nie powinna mieć żadnych wystąpień a pomimo to sama w sobie powinna istnieć – np. Ssak.
- W metodologii obiektowej rolę takich klas przyjmują klasy abstrakcyjne.
- Nie może istnieć żaden obiekt typu klasy abstrakcyjnej.
- W klasie abstrakcyjnej możemy (ale nie musimy) zadeklarować jedną lub więcej metod jako abstrakcyjne.
- Metoda abstrakcyjna nie posiada "ciała" (kodu źródłowego), ale wymaga, aby każda klasa potomna albo tą metodę przesłoniła, wypełniając ją kodem, albo sama była abstrakcyjna.

Przykład klasy abstrakcyjnej


```
abstract class Animal {

    abstract makeSound(): void;

    move(): void {
        console.log("move");
    }
}
```

Interfejsy

- Opisuje kontrakt, który powinny spełniać obiekty
  - Oczekiwane pola
  - Dostępne metody
- Nie tworzymy instancji interfejsu lecz obiekty, które ten interfejs implementują

Interfejs i jego implementacja


```
interface Person {
    firstName: string;
    lastName: string;

    email(message: string): string;
}

class Student implements Person {
    firstName: string;
    lastName: string;

    email(message: string): string {
        return this.firstName + "." + this.lastName +
                "@student.pg.edu.pl";
    }
}
```

Interfejs i jego implementacja


```
let p: Person = {firstName: "Imiesław", lastName: "Nazwiskowy", email(): function() /*...*/};

let p2: Person = new Person(); //błąd kompilacji

let p3: Person = new Student("Nazwisław", "Imiesłowy");
```

Filary programowania obiektowego

- Enkapsulacja (hermetyzacja)
  - Gromadzenie pól w obrębie klas i określanie poziomu dostępu do nich
- Dziedziczenie
  - Wykorzystywanie implementacji klas nadrzędnych w klasach potomnych
- Polimorfizm
  - Posługiwanie się instancjami klas potomnych jak instancjami klas nadrzędnych

Pytania?